TOAD-1 System Architecture Reference Manual

special purpose computer design, manufacturing, and sales

XKL LLC
8420 154th Avenue NE
Redmond, Washington 98052
(425) 869-9050 FAX: (425) 861-7863

Printed copies of this manual often omit Chapter 4, the description of the older processors.

Part Number 50103–00001

Revision 03

January 13, 1999

Restricted Rights Legend. Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227–7013 for Department of Defense agencies, and subparagraphs (c)(1) and (c)(2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227–19 for other agencies.

Warranty

The information in this publication is subject to change without notice. The information contained herein should not be construed as a commitment by XKL LLC.

XKL LLC makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

XKL LLC shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

The information presented here is derived in part from *DECsystem–10 DECSYSTEM–20 Processor Reference Manual* by Digital Equipment Corporation, Marlboro, Massachusetts, July 1980 (and previous editions), with revisions provided by a Technical Newsletter dated June 1982. This material is used here under license from Digital Equipment Corporation.

Notice:

The TOAD-1 System has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the (U.S.) FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

Instruction times, operating speeds, and the like are included here for reference only; they are not to be taken as specifications.

This manuscript was prepared using character recognition software developed by Ibuki, Inc, Los Altos, California, and editing and text formatting facilities of the DECSYSTEM–20. The final version was prepared using the LaTeX text formatting program and the PostScript document description language.

PostScript is trademark of Adobe Systems, Inc.

The following are trademarks of Digital Equipment Corporation: CI, DEC, DECnet, DECUS, DECsystem–10, DECSYSTEM–20, DDT, HSC, HSC–50, MASSBUS, PDP, PDP–10, TOPS–10, TOPS–20, TOPS–20AN, UETP, and digital

Preface

This manual explains the machine language programming and operation, for both instructional and reference purposes, of the PDP–10 central processors used in the TOAD-1 System, the DECsystem–10 and the DECSYSTEM–20. Basically, the manual defines in detail how each processor functions, exactly what its instructions do, how it handles data, what its control and status information mean, and what programming techniques and procedures must be employed to utilize it effectively. The programming is given in machine language, in that it uses only the basic instruction and device mnemonics and symbolic addressing defined by the assembler. The treatment relies on neither any other software nor any of the more sophisticated features of the assembler; moreover the manual is completely self–contained: no prior knowledge of the assembler is required.

The text of the manual is devoted entirely to functional description and programming. Chapter 1 discusses the general characteristics of the system, defines the formats of the words used for numbers and instructions, and explains the conventions needed to program the system and understand the examples given in the text.

Chapter 2 covers all operations regularly available to the user. It includes a general discussion of user programming. Chapter 2 also defines the in–out instructions, even though they are available to the user only in special circumstances; the discussion of the use of in–out instructions for handling the peripheral equipment is not included. For completeness, individual instruction descriptions do include special effects unrelated to user programming, but the detailed treatment of such effects is left for the discussion of system operations.

Subsequent chapters describe the system operation features that provide the means by which a system programmer can create software to manage a system that has many simultanous users. These features, such as a priority interrupt system and a memory management system, are implemented differently in the different processors. Chapter 3 discusses these features in the XKL-1 processor and TOAD-1 System. Chapter 4 discusses these features in the earlier processors: the KL10, the KS10, the KI10, and the KA10.

The first three appendices contain the basic reference tables for the programmer: word formats, instruction and internal device mnemonics, ASCII code, bit assignments showing conditions and status, and a shorthand presentation of instruction actions in symbolic form.

Although specific knowledge of Macro, the assembler, is not required to read this manual, this information will usually be used in the context of assembly language programs. So, for the convenience of Macro users, and in an effort to standardize usage, symbolic definitions relating to the TOAD-1 System hardware are made at various places in the text. These definitions appear in `typewriter font` and they are collected in the Macro source file TD1DEF.MAC and in the universal file TD1DEF.UNV.

Caution

Every effort has been expended to ensure that this manual presents a complete description of the architecture of the TOAD-1 System, the XKL-1 processor, and the several PDP–10 compatible processors. If there is anything you cannot find, *please do not make assumptions* — write to:

TOAD-1 System Architecture Committee
XKL LLC
8420 154th Avenue NE
Redmond, Washington 98052

In some instances the result of an operation using particular operands or given in particular circumstances is indicated as being "indeterminate." This means simply that no guarantee is made of what that result will be. If you experiment and find a result to your liking, you are hereby warned that, if you use the operation, your program may well not be compatible with any other processor, with any other model of your processor, with the same model of your processor at some other installation, or even with your own processor running at some other time with a different version of the microcode or Monitor.

Revision History

This revision history is provided for two purposes. First, a reader who has seen an earlier version of this manual can quickly scan for the areas that have changed. Second, although this history includes many items that are trivial, some items reflect engineering and architectural decisions which may be of interest to readers.

31 August 1993 – 11 January 1994

- This revision list has been created and added to the manual.
- The Preface now explains TD1DEF.MAC.
- Each figure now displays the source file name from which it is derived.
- An accurate drawing of the CPU data paths, HRMF-TD1CPU, figure 1.2, has been provided. Section 1.1.1 has changed accordingly but it is still preliminary.
- Minor correction to figure 1.10.
- The description of BLT and examples of its use have been changed to more clearly explain the effects of BLT when extended addressing is used.
- Omitted the word "preferred" in describing "JFCL 0," as a no–op.
- Corrected the description of the PC flags for the XKL-1 processor: no "Public" flag and no "Previous Context Public" flag in Exec mode.
- Changed the specification of SFM, XJRSTF, and XPCW. In Exec mode, SFM and XPCW store CAC, PAC, and PCS in the right half of the flag word. In Exec mode, XJRSTF and XPCW set CAC, PAC, and PCS from the right half of the flag word.
- HALTRM added (to JRST) for the XKL-1 processor.
- Changed the specification of Arithmetic and Stack Overflow trapping. Provided an 8–word trap data vector in the UPT and EPT for each kind of trap. Combined the User Trap/No–Trap MUUO new PC words; combined the Executive Trap/No–Trap MUUO new PC words. "Trap MUUOs" no longer exist. Split 2.9.6 into 2.9.6.1 and 2.9.6.2.
- The description of byte pointers has been rewritten. One–word globals apply in all sections of extended processors.

- The description of LDB now states explicity that when S is zero, LDB clears the AC.
- Changed the specification of the LUUO trap location in section 0 of the XKL-1 processor to make it match the KL/KS.
- Changed the specification of the MUUO block in the UPT: the first two words now look like an exec–style double word saved PC (with CAC, PAC, and PCS stored in the right half of the flag word). The next two words are the instruction image and E. RDUBR data is no longer stored in the MUUO block. Eliminated "Trap MUUOs". Only Executive and User MUUOs exist now. Added a subsection to 2.16 for LUUOs.
- In Section 3.1.3, added description of Need DC and System Active.
- In Section 3.1.4, the bus address word format changed. (This was announced on the disclaimer page at the front of the 8/31/93 edition.) The 4–bit slot number, formerly adjacent to "D", has been moved right two bits. The two–bit gap between "D" and the slot number is reserved for expansion of the slot number field. Corresponding changes to the immediate page pointer format in 3.6.1.4 and in 3.6.1.2 have been made.
- "Non-existent memory trap" has been replaced by "page trap with a page–fail word indicating a bus timeout."
- No microcode implementation of the material described in 3.2.2.3, Console Micro–command Mode, has yet been attempted. This material is highly susceptible to change. The Disable command has been deleted: the function is accomplished via Enable with a null password.
- An implementation of BOOT is partly complete. There are many more commands than those mentioned in 3.2.4, but they will be described in a different document (or perhaps in an appendix to this document).
- In 3.2.5, the section in which the BOOT ROM is addressed is 10 (octal).
- The old subsection 3.2.6 has been removed. There is no BOOT RAM.
- In 3.2.6 (formerly 3.2.7), various locations in NVRAM have been assigned.
- Subsection 3.3.7 describing the Interrupt register has been added.
- In 3.3.8 (the former 3.3.7), SIMIRD instruction has been added.
- In 3.4.3, the cache diagnostic instructions data formats have been changed to account for the movement of the slot number field.
- In 3.6.1, figure 3.2 has been changed to reflect the MUUO block in the UPT, the Executive and User MUUO new PCs, and the User and Executive trap vector blocks. (These are in the UPT except the Executive trap vector blocks are in the EPT.)
- Table 3.2 (in 3.6.1.6) has been updated with additional (reordered and renamed) page–failure codes.
- The data formats for the Pager Diagnostic instructions have been changed to reflect the shifted position of the slot number field in the bus address word.
- The SYSID instruction has been added to 3.6.2. The data formats for WREBR, WRUBR, etc. have been changed to reflect the shifted position of the slot number field in the bus address word.

- In 3.6.2., changed format of RDCTX/WRCTX to put the CAC and PAC fields in bits 18–23, to conform with SFM.
- The description of pager–disabled mode and system initialization in 3.6.2.1 has been enhanced. Even while the pager is disabled, traps are possible. Therefore, it is mandatory to set up a vestigial EPT/UPT to catch them. (Generally, that is done by BOOT.) The starting address of BOOT, 10003000 has been documented.
- Time Base locations in MemA have been assigned. The syncopated clock is documented in a footnote.
- In 3.8.2, WCTRLF and RCTRLF have been added.
- In 3.11, many changes have been made to the description of the XRH-1 Mass–Storage Interface Processor. The Communications Region has been developed. The format of the Mass–Storage Control Block has been changed.
- In 3.12, Status Read from Address 1 has changed considerably. Packet snoop registers have been introduced. The boundary between the control registers and the data register was moved. Control register assignments were revised. The Message Control Block format has been updated.
- Appendix A.2.1 has been updated to reflect the additional instructions SYSID, WCTRLF, RCTRLF, and SIMIRD. The spelling of RDTIME was corrected.
- Appendix A.3 has been changed to reflect the addition of the instructions mentioned above and HALTRM. The spelling of RDTIME was corrected.
- The index has been enhanced.
- The year has been changed in the copyright notice on page ii.

12 January 1994 – 8 October 1994

- XJRST has been documented. It is implemented in the KL.
- The EXTEND instruction has been added to the index.
- The names of the former `MS%xxx` symbols, defined for the XRH-1, have been changed to `MX%xxx`. This eliminates a confict with some `MONSYM` names for the `MSTR` JSYS. Likewise `.MSxxx` and `MS.xxx` have been changed to `.MXxxx` and `MX.xxx`, respectively.
- Additional symbolic names for MemA locations were defined.
- Request to Send has been added to the signals controled via WCTRLF. The Need DC and System Active Light control bits have moved.
- The description of the virtual memory space created by Boot for programs that it loads has been revised and expanded. Boot does not create a CST; programs loaded by Boot are uncached until they create a CST for themselves.
- In the description of the cache, explicit reference is made to the need for the CST to exist and to specify that a page is cacheable in order for data to be cached

- The description of the CST (in 3.6.1.3) has been corrected: the CST must be aligned on a page boundary.
- The location of the sense and status bytes in the MSCB have been exchanged. `MX%SS` and `MX%SN` have been changed accordingly.
- The wording in the description of SFM has been changed. The semantics are not changed.
- A new privileged instruction, LDLPN, has been defined. Bits 9–35 of C(E) are interpreted as a PAW; the PAW is converted to an LPN, and the LPN is stored in AC. If the conversion fails, a page–failure trap occurs with page–fail code PF.NLP.
- Symbolic names have changed. EPT locations formerly `EP%xxx` have been renamed `EP.xxx`. This is to follow a general monitor convention that field names and bit names include "%" but location names within structures include ".". Also affected: `UP.SSO`.
- The Keep–Alive Trap Control Block has been added at EPT locations 50–53. Keep–Alive monitoring is turned on and off via WCTRLF. The Keep-Alive "timer" is reset by WCTRLF.
- Wrote specification for CMOVE and CMOVEM instructions, analogs of PMOVE and PMOVEM, which look for the data in the cache before trying memory. [This specification was removed on 10/19/94.]
- Changed WRCSB to allow bit 35 of the bus address word to specify the cachability of the CST. The is required to remove circularity in defining the cachability of pages. (This is not implemented in the prototype processor: CST, SPT, map pages, EPT and UPT are all uncached in the prototype.) [However, see 9/14/94.]
- In the description of the XRH-1, clarified that system error report, `MX.CSX`, returns an explanatory byte in *Status*.
- Assigned names to bits in the interval timer.
- Added to the definition of trapping. A trap to executive mode loads the PC flags from the New Flags halfword; it loads CAC and PAC in the machine context from bits 18–23 of the same word; it loads PCS from the old PC. A trap to user mode loads PC flags from the New Flags halfword; the right half of that word is ignored: machine context is unchanged.
- Added to the definition of MUUOs. An MUUO will store the present machine context (CAC, PAC, and PCS) in the right half of the flags word in the MUUO block in the UPT. The new machine context will set PCU according to the state of User in the old PC flags. No other PC flags are set. The new machine context will have PCS set to the section number specified in the old PC.
- Added to the definition of PXCT. Made explict mention of the three quantities that define the previous context: PCU, PCS, and PAC.
- Reserve MSCB fields for the XRH-1. Change Buffer Capacity field to Byte Count.
- Defined formats 2–7 in the MSCB as permuations of 32 bit/36 bit mode and cache look and cache load. [This scheme was abandoned in favor of an explicit command to turn on caching for a particular target.]

- Figure 3.2 has been changed. In UPT, 0–420 and 600–777 are marked as "Available to Software." The MUUO handling has been revised to make the MUUO blocks (executive and user) identical in format to the trap blocks. The MUUO writeup in chapter 2 and figure 2.3 have also changed.
- In Table 3.2, page–failure codes PF.OFF and PF.NLP have been changed to show $H=1$. With this change, codes in the range 1-27 all have $H=1$, and codes 0 (no failure) and 40–65 have $H=0$.
- 4/4/94. Added a sentence to MAP explaining that the result when $E = 1,,0$ is that of virtual page 1000 (and not the meaningless mapping of AC 0).
- 4/16/94. Added a sentence to XJRSTF explaining that, in exec mode, this instruction restores CAC, PAC, and PCS from the right half of the word addressed by E.
- 4/22/94. Clarified warning text regarding byte pointers. Stated also that a byte pointer is interpreted in the context of the section from which it is read.
- 4/25/94. In Table 3.2, page–failure code 0, no failure, is now marked as reserved for software use. At XGCCHK, TOPS–20 simulates a page–failure with code 0 to force a garbage collection.
- 5/3/94. The SWPIA instruction does not clear the "modified" bit in the cache lines. To the extent that this is necessary, do it via the DWRCSH instruction.
- 5/5/94. Additional values were defined for XRH-1 system error report.
- 5/8/94. A footnote has been added to report that the KL10 fails to provide the correct result in ADJBP when AC initially contains 400000,,0.
- 5/12/94. Defined unused fields in the data supplied by the program to DRDCSH as ignored by hardware; defined fields returned by DRDCSH as zero. The corresponding changes have been made in the descriptions of DRDPTB and DWRPTB.
- 5/19/94. The appendix "Processor Compatibility" has been moved to Appendix C and contains some new material.
- 5/20/94. Added a definition, **AM%CAP**, the capacity of MemA in words, 8192.
- 5/24/94. Added a paragraph to XCT describing the effect of executing an instruction in a different section. Also, added a footnote regarding XCT of a trap instruction, JSYS, or MUUO in a section other than the PC section.
- 5/29/94. In 2.9, the numeric opcodes for JFFO and JFCL have been corrected.
- 6/7/94. In the description of the XRH-1 and MSCBs, the name of Command Block Status 0 when returned by the XRH-1 has been changed from "SCSI Command Complete" to "SCSI Command was Performed." This is to emphasize that the command has been attempted and that the success or failure of the command is indicated by the contents of the status field. See also changes to Command Block Status 3, "SCSI Error Status Report", in which the XRH-1 reports that the SCSI Bus and/or protocol failed, as distinct from a report from a specific device.
- 6/12/94. MSCBs to read and write the DRAM have been defined. These are intended to diagnose the DRAM and the path between system memory and the XRH-1.

- 6/20/94. Clarified that the SPT contains entries in the format of a Page Address Word.
- 6/27/94. Corrected the definition of WRSPB in TD1DEF.MAC.
- 6/27/94. The spelling of WCTRLF and RCTRLF in the table in Appendix A (A.2.1, AC field decodes for APR0, APR1, APR2, and APR3) has been corrected.
- 6/27/94. Added RDCFG instruction to read per-slot device and memory configuration information in a way that keeps the monitor independent of the implementation.
- 6/30/94. Added new codes to MSCB for the System Error Report. Added XRH-1 device status register 2, the BAW of the most recent system bus error. Added four error flags to the status word 0.
- 7/01/94. Declared that silly combinations of bits in WRPI are undefined.
- 7/03/94. Shuffled the location of the various flags in WCTRLF and RCTRLF to make them easier to microcode.
- 7/05/94. Shuffled bits in the right half of the interrupt register.
- 7/05/94. Rewrote Section 3.4.9 “Special Considerations” regarding interrupts. Removed references to “trap instructions” as not pertinent to the XKL-1 processor.
- 7/05/94. Added WRTIME to initialize the timebase in an implementation–independent manner.
- 7/17/94. Symbolic names have been added for the offsets within the UPT that address the LUUO, Executive MUUO, and User MUUO blocks. Symbolic names have been added for the offsets within the Trap 1, Trap 2, and Trap 3 trap vector blocks. Symbolic names have been added for the offsets within the EMUUO and UMUUO blocks; the same names apply to the offsets within the trap vector blocks. Symbolic names have been added for the six UPT offsets associated with page–failure traps.
- 7/17/94. Some controversy has arisen regarding MUUOs and traps.

  On MUUO, will PCS be set to the PC section of the MUUO or will it be set to the section from which the MUUO was fetched? The former is easier, the latter is more analogous with how XCT performs. (The question arises only when a XCT in one section targets an MUUO in another section.)

  Can we microcode the machine so that all the information pertaining to a trapping instruction can be saved in a trap block? That would mean preserving the opcode, AC, and *E* during the execution of every instruction so they could be saved in the trap block before trapping. If that is done, we would not need TRAP 1 and TRAP 2 flags anymore. Otherwise, we can not save that info in the trap block, so we might as well go back to having trap instructions instead of trap vector blocks.

  As of 7/17/94 the manual calls for the more difficult implementation. [However, see 7/22/94 and 10/4/94.]
- 7/21/94. The locations of the LEDs controlled by WCTRLF have been described.
- 7/22/94. Redefined trap blocks, MUUO blocks, and page–failure block. In all cases, 8 words have been reserved for the block, the last four of which are essentially an XPCW block, i.e., a double word in which to store the old flags, context, and PC and a double word from which to

load new flags, partial new context, and the new PC. (The partial new context is composed of CAC and PAC. PCS is set by the processor to a value that is still controversial.) [See 10/4/94]

The trap blocks no longer contain an image of the trap instruction.

The MUUO block image of the the MUUO now puts the opcode and AC field in the left half of the word.

An illustration of the page–failure block has been added.

Figure 3.2, TOPS-20 Process Table Configuration, now omits details that are recorded elsewhere in the text. Added cross–references in the figure.

- 7/22/94. Described new XRH-1 functionality. Device Control word 0 now includes a Bus Reset bit and a field in which to specify the number of the affected bus.

- 7/28/94. A value of zero in the Executive Base Register is invalid. A value of zero in the User Base Register is invalid. On initialization, the EBR and UBR are zero.

  When the Executive Base Register is invalid, executive traps (arithmetic, PDLOV, Trap 3) are disabled; all other implicit references to the EBR (e.g., interrupts, Enable Paging, etc.) will halt the processor.

  When the User Base Register is invalid, user traps are disabled; all other implicit references to the UBR (e.g., page–failure, MUUOs, etc.) will halt the processor.

  The UBR should be set up via WRUBR before the EBR is set up.

- 7/28/94. The symbol formerly `AM%CAP`, the capacity of AMEM (number of words), has been renamed `AM.CAP`.

- 8/3/94. The Keep–Alive timer has been assigned its own opcode, WRKPA, an immediate operation to set the value of the time period. Keep–Alive facilities in WCTRLF have been expunged (and the diagram was updated 5/2/95). The locations `AM%KPV` and `AM%KPI` have been removed and `AM%KPA` has been added.

- 8/31/94. Page–failure traps and codes have been changed.

  A new bit, N, meaningful only when H=0, has been introduced. (It overlaps B, valid only when H=1.) When N is 1, the second page–failure word is not determinate. This code is used in codes 2, 5, and 6, which are now marked H=0, N=1. These were codes for hardware-detected programming errors: illegal indirect, pager is off, and LDLPN failure). As these are programming faults, they are now reported as "soft" failures.

  Address Failure and Illegal Address (codes 1 and 3, respectively) have been recategorized as "soft" failures, H=0, N=0.

  When H=1, the failure is hard; e.g., a parity error, bus timeout, bus busy, etc. These failures, mostly unexpected by the software, are not generally a user–related fault. Hence, these trap through the EPT instead of the UPT (same locations though). One further difference is that, if the PI system was on at the time of the trap, it is turned off and bit 13 of the saved flags and context word will be set to 1. XJRSTF in exec mode will restore PI on from this flag bit, if set. [However, some of these changes were reversed on 9/27/94.]

  The page–failure code field (`PF%FLC`) has been moved right four bits (to 12–17) for the sake of being able to read it in octal.

- 9/9/94. An explanation of how $E + 1$ is calculated when the in–section component of E is 777777 has been added to the explanation of DMOVE. Reference to that explanation has been added to other instructions that have double word and multi–word operands.

- 9/13/94. Symbolic names have been added for bits and fields in the CST word. The bits are `CST%WB`, `CST%CB`, and `CST%MB`; the fields are `CST%SC` (state code) and `CST%AG` (age field of state code).
- 9/13/94. Corrected the description of the XNI-1 Control Registers, addresses 0 and 1. The description incorrectly referred to bits 1–4 as containing a slot number; in bus address word format (since 8/31/93), the slot number is in bits 3–6.
- 9/14/94. A new scheme for setting the cacheability of references made by the page refill microcode has been developed. The microcode page refill procedure makes refrence to memory in terms of bus addresses, not virtual addresses. In virtual references, the pager entry determines the cacheability of the reference from data in the CST. Because the refill is a physical reference, the CST data is not immediately available. Logically, it is sufficient to have only a special mechanism to define the cacheability of the CST. However, for performance reasons, we also provide special mechanisms for accessing the SPT, EPT, and UPT:
  - In WRCSB, bit 0 (`CS%CSH`) of the data word, if set, means the CST is cacheable.
  - In WRSPB, bit 0 (`SP%CSH`) of the data word, if set, means the SPT is cacheable.
  - In WREBR, bit 9 (`PG%CSH`) of the data word, if set, means the EPT is cacheable.
  - In WRUBR, bit 9 (`UB%CSH`) of the data word, if set, means the UPT is cacheable.
- 9/15/94. The XRH-1 format codes have been interchanged and augmented.
- 9/19/94. The contents of E must be zero at the start of any cache Sweep All instruction; C(E) may be changed by a sweep all instruction that is interrupted. [Withdrawn 3/13/95.]
- 9/26/94. In immediate pointers (also in Page Address Words), we have defined that zero in bits 5–7 means "in–memory" and non–zero means not in–memory. We allow bit 8 to be used by software. In not–in–memory pointers, mentions in this manual of "bits 4–35" being available to software have been corrected: bits 8–35 are available subject to bits 5–7 not all being zero.
- 9/27/94. In an MUUO, bit 35 of `UP.UOP` (previously undefined) will be set to 1 if the EA Calculation for the MUUO resulted in a global address. Thus, the program that responds to the MUUO can know whether or not the MUUO specified a global address.
- 9/27/94. Rescinded a portion of the change announced 8/31/94. In hard page–failures, the condition of the PI system (on or off) will be reported in bit 0 of the `UP.PFF` word. The bits called B, N, and Y in the page–failure word and in the MAP word have been removed.
- 9/28/94. Added page–failure code `PF.ZPC`, Zero PC. Marked the three Write Not Allowed codes with $V = 1$. Changed the Cache Line Scrambled definition to set $B = 0$ and to delete mention of Y. Added new page–failure bit `PF.RTP`, recursive trap, a modifier to hard failure codes.
- 10/3/94. Changed page–failure codes per new microcode specification. Added codes distinguishing cache data/tag parity errors physical/virtual. Deleted the code for MemA parity error; the condition causes a microcode halt/reboot.
- 10/4/94. Defined that MUUOs shall set PCS to the PC section from which the MUUO is executed. This is compatible with what the KL10 does. Nothwithstanding the failure to be analogous with how XCT computes addresses local to a target instruction in a different section, this method is simple, easy to remember, and implementable.

- 10/6/94. In the description of the use of the *A* field in PXCT in the XKL-1 processor, the text has been updated to mention the use that XBLT makes of bits 11 and 12.
- 10/8/94. Split instruction index from main index in preparation for reissue of the hardcopy version.

18 October 1994 – 7 July 1995

- 10/18/94. Editorial corrections to the manual published 10/8/94: broken reference to figure 1.2; removed bits *N*, *B*, and *Y* from the description of the first MAP word and the first page–failure word (this completes the change started 9/27/94).
- 10/18/94. Added a new instruction, XJRSTP, JRST 11,. *E* points to a three word block. The first word contains the flags and context; the second the new PC; the right half of the third word provides data for WRPI. This instruction provides atomic restoration of PI, PC, flags, and previous context, for DDT.
- 10/19/94. Instructions CMOVE and CMOVEM have been deleted. The definitions of PMOVE and PMOVEM have been changed to reflect that they use the cache when making references to memory pages that are defined as cacheable in the CST.
- 10/19/94. Changes have been made to DRDCSH and DRDPTB. These instructions no longer cause parity-error traps when parity errors occur. Instead, parity information is reported as part of the returned data.
- 10/19/94. Changed the name of page–failure code `PF.TTM` to be `PF.P2M`: "Pager Two Tags Matched". Added new page–failure codes for "Write Not Allowed by CST" and "Two Cache Tags Matched".
- 11/2/94. The sense of the error bit in XNI-1 device status address 0 has been inverted: the bit is 0 to denote an error.
- 11/3/94. Corrected the Global Index Register figure in Appendix A. Bits 1-5 need not be zero. An index register used in a global indirect word is always global: bits 6–35 are used by the hardware; bits 0–5 are ignored. In Chapter 1, added explanatory words to the description of Global Indexing and Local Indexing.
- 11/7/94. Revised the description of the XRH-1 and the CPU's communication with it. Defined additional status bits in reading device status address 0. Defined techniques for reading SRAM and DRAM. Defined Target Blocking, Target Blocking Control commands, and Target is Blocked status. Defined a format bit as controlling the interpretation of `.MXDBA` as a Command List Address or Data Buffer Address. Deleted use of Message In bytes. Added subsubsections on Error Reporting, Error Handling, Long Transfers, and Unaligned Transfers.
- 11/8/94. Removed "I/O Instructions" from the list to which PXCT does not apply: there are no I/O instructions. Added UMOVE and UMOVEM to the list to which PXCT does not apply.
- 11/9/94. Added a warning in WRPI regarding setting the "write bad parity bits" while the PI system is on. (See 11/23/94.)
- 11/9/94. More XRH-1 changes: the Byte Count field returns the residual count, i.e., the count of bytes allocated for the command but not used for data. The description of the contents and handling of Report Asynchronous or Error Status MSCBs has been changed and elaborated.

- 11/9/94. Network interface Device Status Address 0 format has been changed. When other device subtype fields were expanded, this device was overlooked. The subtype has expanded from 3 bits to 6 bits. The Hardware Revision and Microcode Version fields have been shifted to the right by 3 bits, obliterating three formerly unused bits at 18–20.
- 11/11/94. Added bits in page–failure word. `PF%VMA` means the second word is a virtual address; `PF%PMA` means the second word is a bus address word. `PF%DIA`, available for hard failures only, means that additional, implementation–specific diagnostic information has been stored (at an as–yet undefined location). "Implementation–specific" means that the format of this information is not specified as part of the architecture. (A revision of the page–failure codes, coalescing most hard failure codes, is in the works but not yet implemented.)
- 11/16/94. Data format changes have been made in RDAPR and WRAPR. Added two flags, `AP%SHT` and `AP%INT`, by which the console requests system shutdown and kernel DDT (an unsolicited breakpoint), respectively. Moved the NVRAM Battery Low flag to bit 28 of RDAPR.
- 11/16/94. Described XRH-1 changes. Renamed error status `.MXMSC` to `.MXIES`, internal error status. Documented the ASC field corresponding to these codes.
- 11/17/94. Although MAP does not reference memory, it interprets its effective address as a memory address. Therefore, PXCT henceforth shall treat MAP as a memory reference instruction.
- 11/23/94. Page–failure `PF.ZPC` now reports the BAW of the word from which the zero PC was fetched.
- 11/23/94. Data for diagnostic read/write cache/pager have been rearranged. The "write bad parity bits" have been removed from WRPI.
- 11/23/94. CLRPT may now be executed under PXCT for the purpose of clearing user entries in the pager.
- 11/28/94. Provided new diagrams, bits, text for DRDCSH, DWRCSH, DRDPTB, DWRPTB. Write bad parity tag/data was removed from the diagram for WRPI.
- 11/30/94. The descriptions of SKIP, TDN, and TSN have been rewritten to emphasize that, although the instruction is overtly a no–op, the instruction reads memory and may cause side effects from the read. Likewise, the side effects of MOVES have been emphasized. A paragraph at the start of Chapter 2 has been added to explain that the instructions are described in terms of their overt effects and that side effects not visible to the user (e.g., pager refils, CST updates, changes to the cache contents) are to be expected.
- 12/2/94. Changed APRID and SYSID data formats. APRID data has expanded to 3 words. Device Status Read directed to the XKL-1 processor at addresses 0, 1, and 2 now return the APRID data. Device Status Read directed to the XKL-1 at addresses 3–7 now return the data in MemA locations 323–327, respectively.
- 12/3/94. The text has been changed to emphasize that the response by the XKL-1 to Device Status Request is handled by microcode, not by hardware. The CPU does not respond with alacrity to Device Status Request.
- 12/7/94. Added comments to some TD1DEF.MAC entries.

- 12/7/94. Cleaned up definitions to synchronize with new TDBOOT and Microcode. Moved some NVRAM locations: magic numbers, password, default boot path name, and default dump path name. Sixty–four locations at the high end of NVRAM have been reserved for microcode. Allocated some MemA locations for page–failure diagnosis; moved others to be consistent with new allocation; aligned MemA locations to UPT/EPT offsets. Decomitted some MemA locations. Collapsed the hard page–failure codes to four basic codes, with details of "other hard failure(s)" to be decoded from other information.
- 12/19/94. In the appendix, corrected the spelling of the mnemonics for the compare string (CMPS—) instructions.
- 12/28/94. Added E=0 to the description of RDCFG.
- 1/13/95. Changed the date of copyright notice.
- 1/13/95. Updated the description of the memory's response to device status requests. Documented how the ID ROM is read. The board serial number is held in the first three bytes of the ID ROM.
- 1/13/95. PXCT documentation change: Immediate instructions are now documented as requiring the A field of PXCT to be either 4 or 14. This is the same as is documented for "general" instructions.
- 1/30/95. *Local Address Word*, a term used in Chapter 1 but not defined, has been defined. Some wording changes were made to the extended effective address calculation.
- 1/30/95. Added a footnote concerning KL10's handling of S=0 in ADJBP: it gives a No Divide, etc. in this case.
- 1/31/95. Revised the description of the XRH-1's **MX%INV** bit in Device Status at address 0. When set, it now means that the in–memory status is stale.
- 2/1/95. In RDCFG, the contents of AC+1 have been defined for the case when the slot contains an XRH-1. For an XRH-1, documented that the 200 bit in a SCSI ID byte means to take the corresponding channel offline.
- 2/3/95. In the XRH-1, documented Device Control Request to address 3: the program provides a BAW and the XRH-1 returns its main status word at the specified address. Further emphasized the potential bus timeout and/or busy problems of Device Status Request to the XRH-1. Added two more miscellaneous error codes: "emulex gross error" and "emulex rejects an illegal command".
- 2/20/95. Added TD1DEF definitions for memory and XNI-1 registers.
- 2/22/95. Redefined SWPIO, SWPUO, and SWPVO. All now require that bits 27–35 of the contents of E (a BAW) must be zero when the instruction is started. Contents of E may be changed by the execution of this instruction.
- 2/25/95. Redefined **NA%CSN** as 77, address of XNI-1 serial number register.
- 2/25/95. Corrected a typo in the description of MOVSRJ.

- 3/2/95. Added to documentation of the string instructions. In MOVSO and MOVST, emphasized that if the instruction terminates because of a source data condition, the source byte pointer addresses the byte that caused the termination and the destination byte pointer addresses the last byte that was stored successfully. In CMPS—, repeated the notice that the comparision is on unsigned bytes. (This affects only comparisions of 36–bit bytes.) Revised footnotes whose numbers were skewed.
- 3/2/95. Corrected the description of the interval timer to refer to locations 100–103 of the EPT.
- 3/2/95. Corrected the description of the flags in WCTRLF and RCTRLF for the third prototype board.
- 3/10/95. Cleaned up diagrams, etc. to reduce the number of complaints from $\TeX$.
- 3/13/95. Revised the descriptions of the Sweep All instructions. The instructions are interruptable, but they save their state internally, not externally. Caveats have been added warning about executing any Sweep All instruction at interrupt level or in a page–failure trap handler. (See also 4/3/95.)

  Added Machine Check page–failure trap code.

  Added material describing the XRH-1's processing of Report Asynchronous or Error Status MSCBs. Added material regarding the XRH-1's cache. Described the Bus Bad bits in Device Status Read at address 0.
- 4/3/95. Revised definiton of SWPUA and SWPVA. If a SWPUA is interrupted and the interrupt program performs another SWPUA, then the interrupt program's SWPUA will start at the beginning and perform the entire sweep; after the interrupt program dismisses, the interruped SWPUA will terminate immediately. The CPU handles an interrupted SWPVA similarly.

  The rationale for this is that the interrupt program has requested a complete sweep, which might as well be started from the beginning; after that sweep is complete, it may be presumed that the interrupted sweep is logically complete as well.
- 4/5/95. Added bits to WCTRLF for dump, diagnose, and reboot functions in TDBOOT.
- 4/6/95. Clarified that XBLT is legal in section 0 and that PXCT of XBLT ignores PCS.
- 4/7/95. Clarified that the XRH-1 will not alter the slot number in the BAW that describes a transfer; hence, all words specified by a BAW (or by one command of a command list) are in the same slot.
- 4/9/95. Added `AP%IOR`, I/O Reset, to WRAPR. The effect is to clear all APR flags, to clear the Interval Timer, and to clear selected bits in the Console Terminal Status.
- 4/14/95. The name "MSIP" has been changed to XRH-1, corresponding to the name on the board edge. A new returned CBS field in the MSCB has been added: Bus is Being Reset, to aid in restarting MSCBs that were not finished due to a SCSI bus hang.
- 4/28/95. Revised the description of WRITM.
- 4/30/95. Added an explicit description of the main status word of general backplane devices to the explanation of the backplane. Expanded `DS%TYP` to be three bits; shortened `DS%STY` to five bits. Added `DS%TST`, the symbolic name of the device and subtype fields together. Removed `MX%STP`, the subtype field for the XRH-1, in favor of `DS%STY`.

- 5/1/95. Added an appendix containing program–generated documentation for TDBOOT.
- 5/2/95. Added an NVRAM location for auxiliary terminal port parameters. Added a flag in WCTRLF and RCTRLF to enable the auxiliary terminal port.
- 5/17/95. Defined names for XRH-1 cache control functions.
- 5/17/95. Renamed and repositioned the flag in WCTRLF that enables the auxiliary terminal port. Enabling the port lights the corresponding "Port OK" LED.
- 5/18/95. Added a note to WRAPR: it does not sequence through selected options.
- 5/18/95. Added `CF%KPA` to WCTRLF and RCTRLF: Keep Alive counting enable.
- 5/30/95. Reorganized the discussion of page–failure. Described the implementation–specific information stored by the XKL-1 for a hard page–failure.
- 6/1/95. Renamed "TD–1" to be "TOAD-1 System" or "XKL-1 processor", as appropriate.
- 6/12/95. Updated the discussion of XNI-1 to reflect the changed status word.
- 6/12/95. Renamed "keep–alive trap" as "keep–alive interrupt." The keep–alive interrupt is effective regardless of the state of the PI system; it does not change the state of the PI system. We expect to revisit keep–alive.
- 6/13/95. Removed XNI-1 commands for port, serial number, and microcode version number. These will be replaced with fixed locations in XNI-1 memory from which these values can be read. The command reservation scheme will be revised also. RWF will write new descriptions of the Message Control Blocks, since they differ from the description.
- 7/6/95. Replaced section 3.2, Initialization and Console, with sections 3.2, Console, and 3.3, Initialization.
- 7/7/95. Added a warning about bus writes to empty slots. Made minor edits to figures.
- 7/7/95. Prepared the 7/7 printing.

9 July 1995 – 12 October 1995

- 7/19/95. Figures for the KI10 and KA10 section of Chapter 4 have been created.
- 7/21/95. The description of the Word Read Response bus transaction mentions that MISC[7] is the parity error signal. The description of the Status Read Response bus transaction mentions that MISC[7] should be driven to 0 by the responding device.
- 7/26/95. Documented the purpose of the option jumpers; added "jumper" to the index.
- 8/15/95. Documented XRH-1 restrictions in long transfers: the first word of a long transfer command list must be aligned to the first word of a memory line; the address in a "jump" command must likewise be aligned.
- 8/19/95. Corrected an inconsistency in the definition of a global stack pointer. In a global stack pointer, bit 0 is 0, bits 1–5 are unspecified, and bits 6–17 are non–zero. Changed a figure in the appendix.

- 8/21/95. Corrected a note regarding MOVNI. MOVNI AC,0 sets both Carry 0 and Carry 1.
- 8/27/95. After having to re–derive the algorithm twice, the footnote on DIV has been expanded.
- 9/1/95. Defined "Release Cache Data" command in XRH-1. It pertains to recovery of cached operations that could not be completed without error.
- 9/5/95. Updated the table to decode the status of MSCBs returned because of a bus reset.
- 9/6/95. Defined MSCB for negotiating synchronous transfers.
- 9/12/95. Defined Environmental Sense bits in RCTRLF (read–only). The AC Fail signal on the backplane is actually (AC Fail) OR (Thermal Warning).
- 9/13/95. Added a new XRH-1 error message: CBS field invalid.
- 9/13/95. Documented console parameters.
- 9/21/95. Documented restrictions on XRH-1 unaligned transfers.
- 9/28/95. An explanation of how to do transfers of less than integral disk sectors has been added to the XRH-1 documentation.
- 9/28/95. Added further clarification of the interval timer.
- 10/4/95. Changed XRH-1 definitions of soft reset; added quietus reset.
- 10/6/95. Added definition of `.MDERR` memory error register.

17 October 1995 – June 1996 (Revision 01)

- 10/17/95. Updated a figure in the XRH-1 description to show the quietus reset bit.
- 10/24/95. In the description of the GFLTR and DGFLTR instructions, supplied correct values for the inserted exponents.
- 10/31/95. Added further explanation of the behavior of PXCT when the EA calculation is in current context and data reference is in previous context.
- 11/29/95. Added material on the behavior of the XRH-1 as a target.
- 1/2/96. Editorial revisions to prepare Revision 01. Changed the date of the copyright notice.
- 1/17/96. The format of data stored in hard page–failures has been revised. EPT word 501 now contains page–failure data, (offset `UP.PFD`), specifically, the contents of the "D to D" latch at the time of the failure. The state of the PI system (PION) prior to to the failure is stored in EPT word 502, bit 11.

  Temporarily, 1B1 in microcode options ("exotic microcode") will be set to 1 to indicate the new microcode. When we upgrade all systems, we will decomission the bit. (Meanwhile TOPS–20 needs to know where the PI state was stored.)

- 4/3/96. XRH-1 documentation changes: When a target is blocked, all MSCBs are returned marked “Target is Blocked”; the former exception for a Request Sense command is removed. When a “Clear Target is Blocked” command is received, the XRH-1 will force any pending MSCBs back to the CPU before returning the “Clear Target is Blocked” MSCB.
- 4/24/96. In an extended KL10, an LUUO from a non–zero section in exec mode uses the contents of EPT location 420 as the exec address of a 4–word LUUO block, by analogy of the behavior in user mode. Formerly, the manual said that an LUUO trapped as an MUUO. The behavior of the XKL-1 has been changed to correspond to the actual behavior of the KL10 in this case.
- 4/29/96. The section on the XNI-1 has been replaced.
- 5/2/96. Added two more flags to WCTRLF and RCTRLF: `CF%ATO` (automatic), and `CF%DBG` (debug). These to increase the amount of information that TDBOOT can pass to a newly–loaded monitor.
- 5/3/96. Miscellaneous cleanup. Tied up some loose ends.

  Moved the definitions of MemA and NVRAM locations to an appendix: these are **not** part of the architectural specification.

  Zero PC is a “hard” page-failure.

  The initialization error codes have been documented.

  The interrupt FIFO bits 18–25 have been documented more accurately.

  The CST bits used by the microcode are described.

  The description of “permanent” executive PTB entries has been omitted. Nothing has been implemented as yet.
- 5/8/96 The password for the auxiliary console has not been implemented. The battery life estimation has not been implemented. References to these have been deleted.
- 5/10/96 The description of RDPI now omits mention of write bad parity; this should have been changed 11/28/94.
- 5/16/96 Examples of processor differences have been cleaned up. An example in which the XKL-1 was said to produce a different result than the KL10 for FAD has been omitted.
- 5/23/96 Added NVRAM location for auto–boot delay.
- 5/28/96 The Processor Identification code fragment was rewritten to more accurately select between processors.
- 5/30/96 Additional material clarifying BLT.
- 6/1/96 Added footnote in DFMP: the KL10 does not round negative numbers according to the usual rules of floating–point rounding. When the result is negative and the fraction being dropped is precisely 1/2 LSB, the KL10 adds 1 LSB. In twos complement, 1 should be added to the LSB of a negative result only if the fraction is strictly greater than 1/2 LSB.

6 June 1996 – 30 April 1997 (Revision 02)

- 6/6/96. Corrected the description of MemA location `AM%PFD`. It contains a copy of the data found in the DtoD latch when a hard page failure occurs. (This was part of the change made 1/17/96.)
- 6/6/96. Corrected the depiction of a local stack pointer in Appendix A. The right half is now labeled "Local Address of the Latest Element". Formerly it was "In–Section Address ...", the distinction being that local 0 is an accumulator, whereas, above section 1, in–section 0 is memory.
- 6/6/96 Additional material clarifying BLT behavior in the KL10.
- 6/7/96. Additional clarification regarding extended addressing, in chapter 1, in BLT, and in EDIT.
- 6/17/96. New page failure code, `PF.HMC`, for hard failures delivered subsequent to processing by the macro–console.
- 6/18/96. Added symbols to support TDBoot and its new facility for inspecting/correcting the cache and Pager Translation Buffer.
- 6/24/96. To support TDBoot, added symbolic names for the hard page–failure bits in EPT 500. Added `AP%HPM`.
- 6/29/96. Corrected definition of `AP%DPC`.
- 7/17/96. New bit in hard page–failure or MAP word, `PF%HMF`, hard map failure. Added explanation to the description of MAP.
- 7/21/96. Definitions for the AC block addresses in MemA, `AM%AB0` ... `AM%AB7`, have been added.
- 8/9/96. Definitions for fields in the data for WRADB have been added.
- 8/31/96. Locations 421-423 in the UPT are reserved for software. (For compatibility with the KL10 and TOPS-10, the Monitor is allowed to store images of the user's "trap instructions" in these locations. TOAD-1 System does not support trap instructions, but this is a convenient place for the software to store the instructions to emulate for the user.) Revised figure 3.3. (Added a second version of the UPT/EPT configuration for KI10 Paging Mode. We have not yet commited to support KI paging.) [Subsequently, we have decided not to support KI paging and this second version has been omitted from the document.]
- 10/27/96. Slightly revised the description of the XNI-1's response to a Device Control bus cycle.
- 12/24/96. Added an example of comparison of two double–length floating numbers.
- 1/3/97. Incorporated corrections suggested by customer review. Updated copyright year. Changed company name to "XKL LLC".
- 1/3/97. The instructions that perform G–format floating–point operations are not implemented in the XKL-1. Instead, they trap as unassigned codes and are simulated by the operating system.

- 3/31/97. Added a description of new backplane bus cycle types, "DoubleWord_Write_Multiple" and "DoubleWord_Read_Multiple_Request". These have not yet been implemented in any component of the present TOAD-1 System.
- 4/16/97. Changed telephone area code on the title page. (Our local provider says that the new 425 area code will be usable by the beginning of May, 1997 and mandatory before the end of the year. If 425 fails, try area 206.)
- 4/24/97. Corrected typographical errors. Updated certain portions to reflect changes made by the June 1982 updates to the Digital version of this manual.

30 April 1997 — present (Revision 03)

- 6/4/97. Reformatted Table 3.2 to fit properly on one page. Fixed title page to show the new revision number.
- 10/6/97. Corrected the spelling of the instruction mnemonics for `GDFIX` and `GDFIXR` in Appendix A.2. Corrected a sorting problem in Appendix A.3.
- 9/26/98. Added a preliminary description of XMG-3. Corrected the names and descriptions of backplane cycle types that support longword transfers (that were added 3/31/97). The XMG-3implements these.
- 9/26/98. Added a description of TDBoot's "define slot *n* off–line" and "... on–line" commands. Added definitions of NVRAM and AMEM locations needed to support this command.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A TOAD-1 System, DECsystem–10, or DECSYSTEM–20 is a general–purpose, stored–program computing system that includes at least one PDP–10 compatible central processor, a memory with error–checking capability, and a variety of peripheral equipment. Each central processor is the control unit for an entire large–scale subsystem, in which it is connected by buses to random–access storage modules and peripheral equipment, some of which may be shared with other central processors. Within a given system the central processor governs all peripheral equipment, either directly or indirectly; sequences the program; and performs all arithmetic, logical, and data–handling operations.

A given system may also contain other kinds of processors.

- A TOAD-1 System is based on an XKL-1 central processor; all in–out and memory operations are performed over a high–speed backplane bus. The console functions are supported by microcode in the CPU (with a terminal). Communications equipment (other than the console terminal) and unit–record peripherals are supported indirectly via a network.
- A system based on the KL10 central processor contains a small PDP–11 front–end processor; this acts as the system console and it may also handle communications equipment and the unit–record peripheral equipment via a Unibus.
- The DECSYSTEM–2020, the only system based on the KS10 processor, contains a microprocessor for handling console functions (with a terminal). All of its peripheral equipment is handled over two or more Unibuses.
- Earlier central processors (the KI10 and the KA10) have manual consoles and handle unit–record equipment directly via an in–out bus.

A system may also include direct–access processors, which have much more limited program capability and serve to connect large, fast peripheral devices to memory, bypassing the central processor. Every direct–access processor is connected, for control purposes, to some central processor, to which it appears as a peripheral device. The direct–access processor is also connected to its peripheral equipment by a device bus, and to memory either directly by its own memory bus or via a channel bus through the memory control part of the central processor. Although a DECSYSTEM–2020 cannot include direct–access processors, the Unibus adapters themselves have much of the capability of such processors: in particular, an adapter can gain direct access to memory via the same KS10 system bus used by the processor.

A system may also contain peripheral subsystems, such as for data communications, which are themselves based on small computers; from the point of view of the PDP–10, such a subsystem in toto is regarded as a peripheral device. Unless otherwise specified, the words "processor" and "central processor" refer to the large–scale PDP–10 central processor.

Five types of PDP–10 central processors are discussed in this publication: the XKL-1, the KL10, the KS10, the KI10, and the KA10. The XKL-1 processor in the TOAD-1 System implements full 30–bit extended addressing and the largest instruction set (that of the KL10) including string manipulation and double precision in fixed point, floating point, and extended–range floating point.

The KL10, which exists in two versions, with and without extended addressing, is the fastest and most powerful processor in the K–series; the KL10 implements the largest instruction set.

The KS10 executes a subset of the KL10 instructions: the extended–range floating point instructions are omitted. The KS10 lacks extended addressing and is slower than the KL10; on the other hand, it is also considerably less expensive.

All systems handle words of thirty–six bits. Earlier memories store these with a parity bit for detecting single–bit errors. In the MOS memories on the KL10 and KS10, each word is accompanied by a 7–bit code for correction of single errors and detection of double errors. The TOAD-1 System memory, also MOS, implements a single parity bit for detecting single–bit errors. Maximum memory capacity depends upon the physical addressing capability of the processor. However, the physical capacity of the memory is not particularly relevant to a typical user programmer, as all recent processors are structured to operate in a sophisticated virtual memory environment.

The fundamental virtual address is thirty bits, although only the TOAD-1 System is capable of using all of them. The virtual memory space is divided into sections of 256K each, whose locations are specified by the right eighteen address bits (the "in–section" address). Paging hardware further divides each section into 512 pages of 512 locations each. The actual size of the virtual address space for a given processor depends on how many out of the twelve possible section bits it implements. The addressing characteristics of the various processors are these:

	XKL-1	*Extended KL10*	*Single–section KL10*	*KS10*	*KI10*	*KA10*
Physical address (number of address bits)	4+29[*]	22	22	20[‡]	22	18
Physical memory capacity (number of locations)	128M[†]	4M	4M	512K	4M	256K
Section bits implemented	12	5	0	0	0	0
Number of sections	4096	32	1	1	1	1
Virtual address (number of bits)	30	23	18	18	18	18
Virtual address space (number of locations)	1024M	8M	256K	256K	256K	256K

K = 1024 (decimal); M = 1,048,576 (decimal).

[*] 4–bit physical slot number and 29–bit in–module address

[†] Four 32M memory boards.

[‡] The maximum physical memory capacity of the KS10 is 512K.

In an Extended KL10 whose operating system supports extended addressing only in executive address space, user space is the same as that in a single–section KL10.

The XKL-1 processor, by using all twelve section bits, has a virtual memory larger than its physical memory capacity. The extended KL10, by using five section bits, has a virtual memory twice the size of the maximum physical memory. All other processor configurations use only the 18–bit in–section address, so all access is defined as being in section zero. This means that the KS10 has a physical memory that can be twice as large as the virtual space available to a single program; and the single–section KL10 and the KI10 can have a physical memory sixteen times as large. However, a virtual address limitation of 256K may be problematic in some applications, thus the KS10 and other single–section processors may be unsuited for large applications. All processors except the KA10 have features that allow for dynamic paging and working–set management so that the system may obtain the best utilization of physical resources. KA10 memory management is limited to a basic one– or two–part protection and relocation scheme.

The bits of a word are numbered 0–35, left to right (most significant to least significant), as are the bits in the registers that hold the words. All processors handle half words and bytes. The XKL-1, KL10, and KS10 can also handle double words and strings.

In this manual bit numbers are given in decimal notation. However, most other numbers are in octal, i.e., radix 8, notation. Specifically, memory addresses are in octal unless otherwise specified.

> Half words are simply the two halves of a word, wherein the left half is bits 0–17 and the right half is bits 18–35. In operations on half words, the two halves of a given word are handled independently; e.g., when both are incremented, no carry from right to left can occur. (However, this is not true on the KA10, where incrementing both halves is done

> by adding 1000001 to the entire word.)
>
> A byte is any contiguous set of bits within a word. It is identified by a byte pointer.
>
> A double word is two adjacent words treated as a single 72–bit entity, where the word with the lower address is on the left. In some operations, such as the product in double–precision multiplication, this concept is extended to multiple–length operands involving more than two consecutive words. The direction from more to less significance is always from lower to higher addresses. (The KA10 cannot handle double words, except to the limited extent of double–length products and dividends; the KI10 handles double words to the extent of operands in double–precision floating–point operations.)
>
> A string is a sequence of bytes packed into and encompassing an arbitrary number of words. It is defined by its length in number of bytes and an initial value for a pointer that is incremented automatically for handling the bytes. (Neither the KI10 nor the KA10 have string hardware.) (Hardware strings do not necessarily correspond to the implementation of the "string" data type in high–level languages.)

Processor internal registers specifically for holding addresses have a number of bits appropriate to the type of processor and whether the address is physical or virtual. Address bits are numbered according to the right–justified position of an address in a word. Thus the bits of an in–section address are numbered 18–35, and those of a TOAD-1 System 29–bit in–module address are numbered 7–35. Words are used either as instructions in the program, as addresses, or as operands (data for the program).

Most of this introductory chapter, §1.5 through §1.8, is applicable to any PDP–10 compatible processor, although the discussion tends to be oriented towards the TOAD-1 System's XKL-1 processor or systems based on the KL10; these sections are germane to anyone who wants to program these systems in assembly language. Section 1.4 may be of interest only to system programmers. Section 1.1 applies only to the TOAD-1 System; §1.2 applies only to the KL10; and §1.3 applies only to the KS10. Much of the information for the KL10 applies also to systems based on the KI10 and KA10; §1.9 explains the ways in which those earlier processors differ from the architecture defined in the preceding sections.

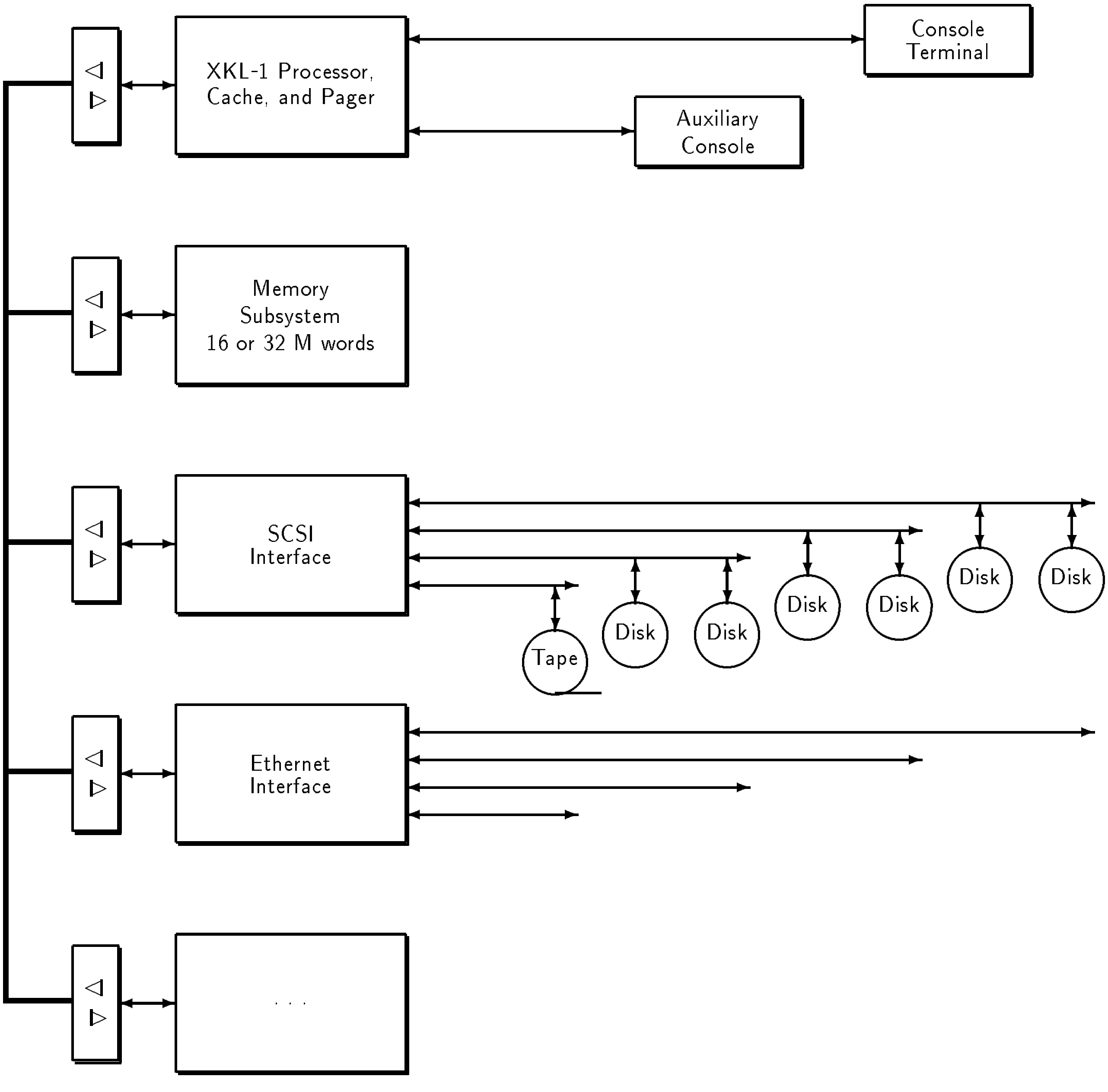
At various points, this manual contains symbolic definitions for individual bits and fields. These definitions are signalled in the text by `typewriter font`; they are suitable for use with Macro, the assembler. The collected definitions are available through a universal file called TD1DEF.UNV.

1.1 TOAD-1 System Organization

Figure 1.1 shows the organization of the TOAD-1 System, which is effectively a collection of processors and memory organized around a backplane bus. At least one XKL-1 central processor must be present in the system. The other processors (e.g., device controllers) generally act at the direction of the XKL-1 processor but perform their actions asynchronously.

The TOAD-1 System backplane bus may have as many as fourteen devices attached. (The initial TOAD-1 System has capacity for just seven devices.) A minimum system consists of the XKL-1 processor (including console terminal ports, cache, and pager), memory, a SCSI subsystem, and a network control subsystem. No direct provision is made for unit–record equipment (e.g., line

Figure 1.1: TOAD-1 System Configuration



printers) or for terminal connections (excepting the console terminal); these can be handled swiftly and efficiently via the network.

The SCSI and network subsystems are designed to read in–memory command lists and transfer data directly to and from memory without interrupting the XKL-1 processor. These devices can request priority interrupts to alert the XKL-1 processor to a change in status (a message queue going from empty to non–empty, a semaphore state change, etc.) or an event (e.g., error conditions) that is beyond the ability of the subsystem to handle.

1.1.1 The XKL-1 Central Processor

Figure 1.2 shows the internal data paths and main processing elements of the XKL-1 processor.

Omitted from the figure is the microcontroller, which, through its programmed instructions (the microcode), controls the operation of the processor by providing step–by–step directions to the various data–path components. The illustration also omits most of the control lines emanating from the microcontroller and extending throughout the machine. Some of the control lines are illustrated: "GP" signifies the general–purpose field of the microcode; thus, some of the microcontroller program is used as data in controlling the data–path elements.

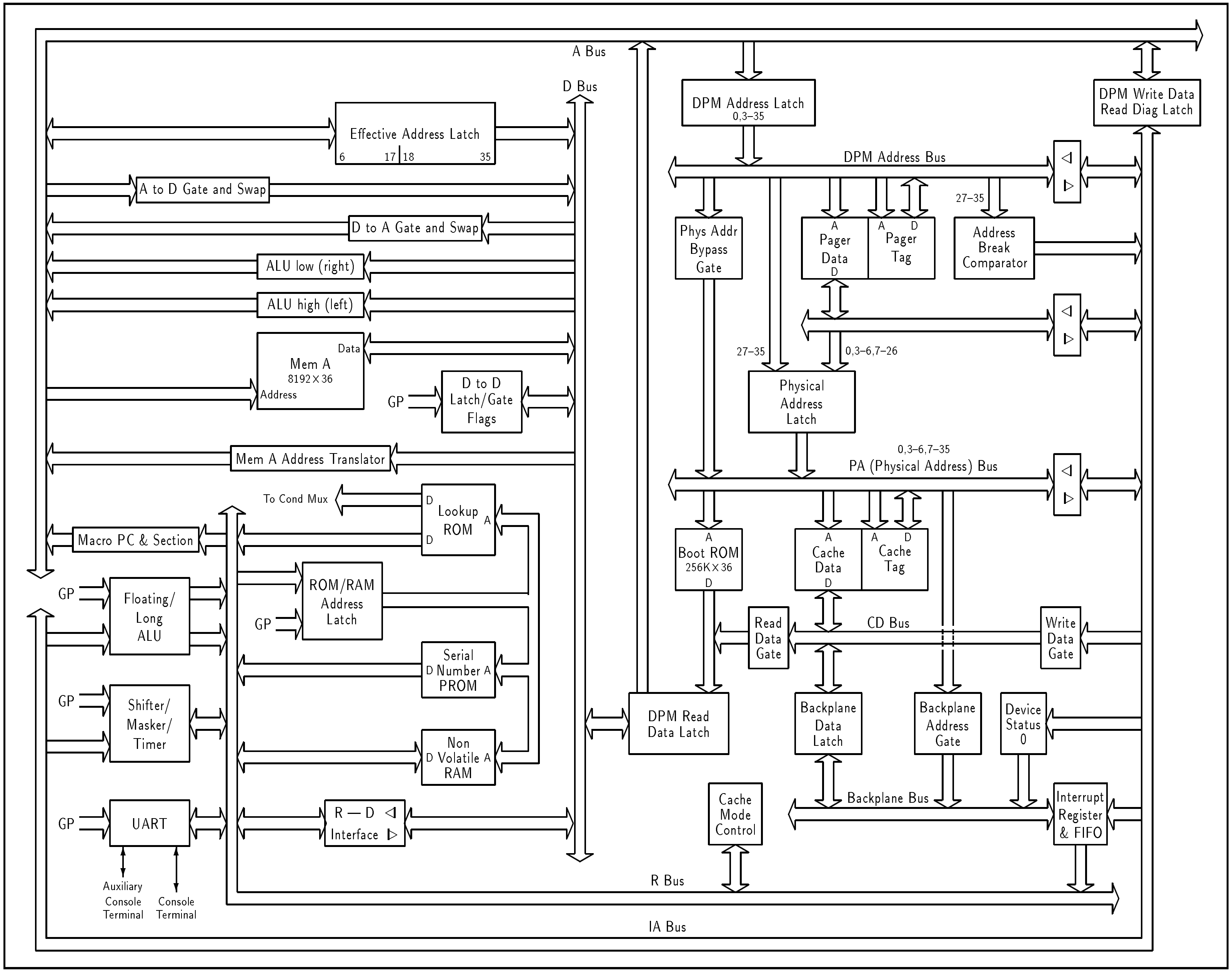
Of the registers shown, only PC (the program counter, labeled "Macro PC & Section"), the Program Flags (in the "D to D Latch/Gate"), and one set of accumulators (within "Mem A") are directly relevant to a typical user. The processor performs a program by executing instructions retrieved from the memory locations addressed by PC. For the normal program sequence, PC is regularly incremented by one so that instructions are taken from consecutive locations. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time in a skip instruction, or by replacing its contents with the value specified by a jump instruction. Throughout the text, the phrase "jump to location n" means to load the value n into PC and to continue performing instructions in the normal counting sequence, beginning at the location then specified by PC. When counting the PC, no carry is allowed into the section part. Hence, although large data structures can arbitrarily cross section boundaries, the program cannot. The program count wraps around in the current PC section, which is specified by PC bits 6–17. For the program to go from one section to another requires an explicit transfer of control by jumping to another section.

Each instruction retrieved from memory contains information identifying the operands and an instruction code specifying the operation to be performed using those operands. The instruction is decoded by the microcontroller, which in turn performs the instruction by manipulating all of the other processor elements and making the necessary requests to the memory. The microcontroller also executes the more fundamental operations of sequencing the program, handling TOPS–20 paging operations beyond the basic address translation made by the pager, processing interrupts, and so forth.

The microcontroller operates from microcode contained in a control store. This microcode bears the same relation to the microcontroller as the program does to the processor. Microprocessing is invisible to the programmer, who need not be concerned with the microcode. The reader should, however, note an important implication of this type of processor implementation: a single XKL-1 could potentially process a different instruction–set by loading a different microcode.

The major working area of the processor is the arithmetic logic unit (ALU). This unit performs 36–bit integer arithmetic, half–word arithmetic, and logic functions. Double–precision integer and

Figure 1.2: XKL-1 Central Processor Data Paths



floating–point arithmetic are handled in the Floating/Long ALU. The shift matrix is employed in shift and rotate instructions and in operations that imply shifting, such as floating–point arithmetic and the byte and string operations. Combinations of these registers play a role in all arithmetic, logical, and data handling operations and in program control operations. Although almost all of the operations necessary for the execution of a program are performed in the ALU, the details of its operation are not important to the programmer because the ALU does not retain information from one instruction to the next. Computations either affect control elements such as PC and the program flags, or produce results that are stored and must be retrieved if they are to be used as operands in other instructions. The program flags report conditions of interest to the programmer, such as arithmetic and stack overflow, which can cause program traps.

Although all computations on both operands and addresses are performed in the arithmetic logic, the computer actually has sixteen accumulators, fifteen of which can double as index registers. The first sixteen memory addresses correspond to the accumulators instead of locations in the storage modules. The factor that determines whether one of the first sixteen locations in memory is used as an accumulator or as an index register is not the information it contains nor how its contents are used, but rather how the location is addressed. The accumulators can be addressed in three ways. First, any instruction can access an accumulator by specifying one of the first sixteen addresses, i.e., addresses 0 through 17. Second, most instructions (including all that combine two operands) can access an accumulator as one of the operands by putting the accumulator number in the accumulator field of the instruction. Third, fifteen of the accumulators can be accessed as index registers by specifying a non–zero accumulator number in the index–register address field of an instruction. (A zero in the index–register address field specifies no indexing, hence, accumulator zero can not be used as an index register.)

These first sixteen locations are not actually in the storage modules—they are in MemA, the fast memory contained in the processor. This allows much quicker access to these locations, whether they are addressed as accumulators, index registers, or ordinary memory locations. They can even be addressed by the program counter so that short instruction sequences can be run in them. Provision is made for referencing these locations from non–zero sections. Moreover, there are actually eight of these fast memory blocks (also referred to as “AC blocks”), but generally only one is available to a program at any given time. The Monitor usually reserves block 0 for itself and assigns the others to user programs.

As mentioned above, the accumulator blocks occupy a portion of the processor’s private random–access memory called MemA. In addition to the accumulators, MemA also holds various parameters that control the pager, the timebase, etc.

An instruction word has one 18–bit address field for addressing any location in the current PC section. Every instruction has a 4–bit index–register address field, which can address fifteen of the accumulator locations for use as index registers in modifying a memory address. Any instruction that requires a second operand has a 4–bit accumulator address field which can address one of the sixteen accumulators. In other words, any accumulator can be addressed as though it were a result held over in the processor from some previous instruction. (The programmer usually has a choice of whether the result of the instruction will go to the location addressed as an accumulator or to that addressed by the 18–bit address field, or to both.)

Addresses, whether from the PC or from the effective address calculation for an instruction, are held in the DPM Address Latch as they are presented to the pager. The DPM Address Latch holds either a 30–bit virtual address or a 34–bit backplane bus address. A virtual address is translated by the pager to a 34–bit backplane bus address that is supplied to the backplane bus via the Backplane

Address Gate and the bus control logic. The bus address is composed of the *D* (device) bit, a 4–bit physical slot number, and a 29–bit in–module address.

The cache speeds up average memory access and increases the efficiency of the storage module. This facility has 131,072 locations that temporarily substitute for a selection of the most–frequently used storage locations. Hence, the cache may be regarded in some respects as a set of general purpose registers. A program loop, once read from storage and then resident in the cache, may be executed hundreds of times without further instruction fetches from storage. Data produced by the program is written in the cache. Thus, if the program sets up a location to be a counter, that location may be read and written thousands of times with only the initial storage access. When the cache does not contain the word the program wants, memory control gets a line of eight adjacent words from storage, including the requested one, and places them in the cache, on the assumption the program will probably want the other seven and can thus get them more quickly. This is a reasonable assumption, since the program generally executes from consecutive locations and many forms of data manipulation are sequential as well. Cache control has a mechanism for determining frequency of use, and it writes the least–recently used line back into storage (or discards it if unchanged) when the cache space is needed for new references. There are 8,192 two–way associative 8–word lines in the cache. Physical address bits 20–32 select a cache line, and bits 33–35 select the word within the line. Only two lines with the same address in bits 20–32 can fit in the cache at a time; but, since user programs have no control over the physical addresses allocated to their programs, there is nothing to do and nothing to avoid in trying to improve a user program's utilization of the cache. There may be complete pages in the cache, but it is more likely to have a selection of lines from a selection of pages depending on frequency of use. Generally the cache contains words for the current user and for the Monitor, as well as for handling interrupts for many users. The reader should be aware that the cache contains representations of memory word lines, not necessarily the actual storage contents. For example, when the program writes a word, the contents of that cache location then differ from the contents of the corresponding storage location. This caution is of interest, however, only to the operating system. A typical program simply makes memory references; the more of these in which the cache substitutes invisibly for storage, the better.

Also included within the processor are a number of internal devices that are similar to external controllers in that they operate asynchronously but are controlled by the program. Some of these have already been mentioned: the program sets up the pager, instructs cache control to update storage, sets up the memory system, and gets diagnostic information from the memory controllers and storage modules. Other such "devices" are the console terminal, the interval timer, the timebase, the error logic, and the priority interrupt system. The priority interrupt system facilitates processor control of the entire system by means of a number of priority–ordered levels through which external signals may interrupt the normal program flow. The processor acknowledges an interrupt request by transferring control (by means of XPCW) to a memory location selected by the backplane location (slot number) of the requesting device. Assignment of levels to devices is entirely under program control. Among the devices to which the program can assign levels are the error logic, the console port, and the interval counter.

1.1.2 TOAD-1 System Memory

The TOAD-1 System main memory is organized as modules of 16– or 32–million (2^{25}) 36–bit words, with single–bit error detection. More than one module may be present in a system. The physical constraint on memory capacity is complicated to state. The initial TOAD-1 System has seven device slots, with a mandatory processor, memory, SCSI subsystem, and network subsystem occupying four slots. Up to three memory boards could be added to a minimum system, bringing the maximum memory capacity to 128 megawords, but that may not be a well–balanced system. The architectural constraint on memory capacity is more liberal, allowing for larger capacities in the future: the backplane bus provides four address bits for slot selection (slots 1–15 are allowable; slot 0 has special meaning) and a 29–bit in–module address.

The memory is organized to read and write single words or to read and write 8–word "lines" of memory corresponding to the cache structure of the TOAD-1 System.

With the cache enabled for a given page, memory access is handled using the cache wherever possible; when storage access is required, transfers are in 8–word lines. For a read request, the processor reads from the cache if the word is there; otherwise, it initiates a storage–to–cache transfer, which may require a prior cache–to–storage transfer to make room for the new data. For a write request, the processor always writes in the cache, and this too may require a cache–to–storage transfer to make room. When a write operation is directed to a storage location not already represented in the cache, a storage–to–cache transfer is performed to initialize the cache line to which the write is directed. Other than the cache–to–storage transfers, the processor writes in storage only when the cache is not being used or when the Monitor specifically updates storage from the contents of the cache.

A cache–to–storage transfer occurs when the Monitor needs to be sure that memory is validated (i.e., updated according to the written–in portions of the cache); for example, just prior to a device output operation. Cache–to–storage transfers are also performed when a cache line is needed and the least–recently used line is "modified" (i.e., has data that needs to be written). A cache–to–storage transfer will send all eight words in the line in one bus transaction of five bus cycles.

A storage–to–cache transfer occurs when a word that is not in the cache is read or written. The storage–to–cache transfer may initiate a cache–to–storage transfer in order to make a cache line available, as described above. When space in the cache is available, the cache control will ask the memory to fill the line; the request will also specify which pair of words to send first. The memory will respond with the designated pair of words. In the case of a read, one of these is the data for which the processor was waiting, so the processor continues. In the case of a write, the processor was not waiting, and a word is available to replace one of the words read from memory; when the first pair of words is read, one will go to the cache, the other is discarded and the newly written data is put in the cache instead. Then, in sequence, in the next three bus cycles, the memory supplies data to fill in the other six words of the line.

Memory is addressed on the backplane bus by means of a physical slot number and an in–module address. The pager (§3.7) translates virtual addresses to bus addresses for memory references. The TOAD-1 System hardware interprets virtual addresses 0–17 as accumulators (in the currently–selected AC block) in MemA; these addresses are not interpreted by the pager.

Although backplane physical slot number 0 does not exist, the XKL-1 processor makes its boot program and a collection of diagnostic software in read–only memory (the boot ROM) addressable via slot number 0; thus, in a multiprocessor system, each processor accesses its unique boot ROM. Processor microcode also makes use of the fast, on–board, random–access memory known as MemA. MemA cannot be accessed by regular instructions, but two privileged instructions provide system and diagnostic access to MemA. Particular locations within MemA that may be of interest to operating system programmers are discussed in §3.6.1. All other hardware–defined addresses are relative to pages, such as the process tables, whose physical location are specified by the Monitor. Physical memory in a system is a constant, unless a storage module is actually added or removed. The virtual address space accessible to a particular program is entirely a function of the way in which the Monitor sets up user operating conditions, except that any space and any restrictions must encompass an integral number of pages.

1.2 KL10–based System Organization

The illustrations that follow show the organization of the two types of computer systems based on the KL10 central processor and the internal organization of that processor. A KL10–based system is effectively a group of processors organized around an E or execution bus. The other processors (controllers, interfaces) generally act at the direction of the central processor but carry out those actions independently of it.

On the E bus of a DECSYSTEM–20, there may be up to four DTE20 interfaces, each of which connects to a PDP–11 front–end processor, and up to eight RH20 Massbus controllers (Figure 1.3). An RH20 handles disks or tapes via a Massbus; although fundamentally under control of the KL10, the RH20 operates from its own command list in memory and uses a separate C or channel bus for data transfers to and from internal memory via the M box, bypassing the E box. All DECSYSTEM–20 memory is internal: the memory controllers with their storage modules are connected directly to the S or storage bus, and access to them is possible only through the M box.[1] Unit record equipment, such as line printers and card readers, and communication subsystems are handled by PDP–11 front–end processors. The data path to memory for these is via the E bus, but it uses automatic features of the priority interrupt, thus interfering minimally with the KL10 program. Among the front–end processors, one is master: it acts as the system console and bootstraps the system by loading the KL10 microcode from disk; it is also the system diagnostic facility (for which it has a direct connection to one of the disks on the RH20).

Figure 1.4 shows a typical DECsystem–10 based on a KL10. In terms of memory and peripherals, such a system is much like a KI10–based DECsystem–10, but it has the faster and more powerful central processor. Here external memory is on a KI10 memory bus interfaced to the S bus by a DMA20, and the peripherals are on a KI10 in–out bus interfaced to the E bus by a DIA20. Massbus devices are handled by an RH10, which maintains a direct path to external memory by way of a data channel. Such a system generally has only one front–end processor, which acts as the console and diagnostic facility and bootstraps the microcode from disk or DECtape. One version of the DECsystem–10 is more of a hybrid 10–20: a machine in the 1090 series has KI10 memory and in–out buses but uses the RH20 Massbus controller, which is located on the E bus and maintains a path to external memory by way of the C bus through the M box.

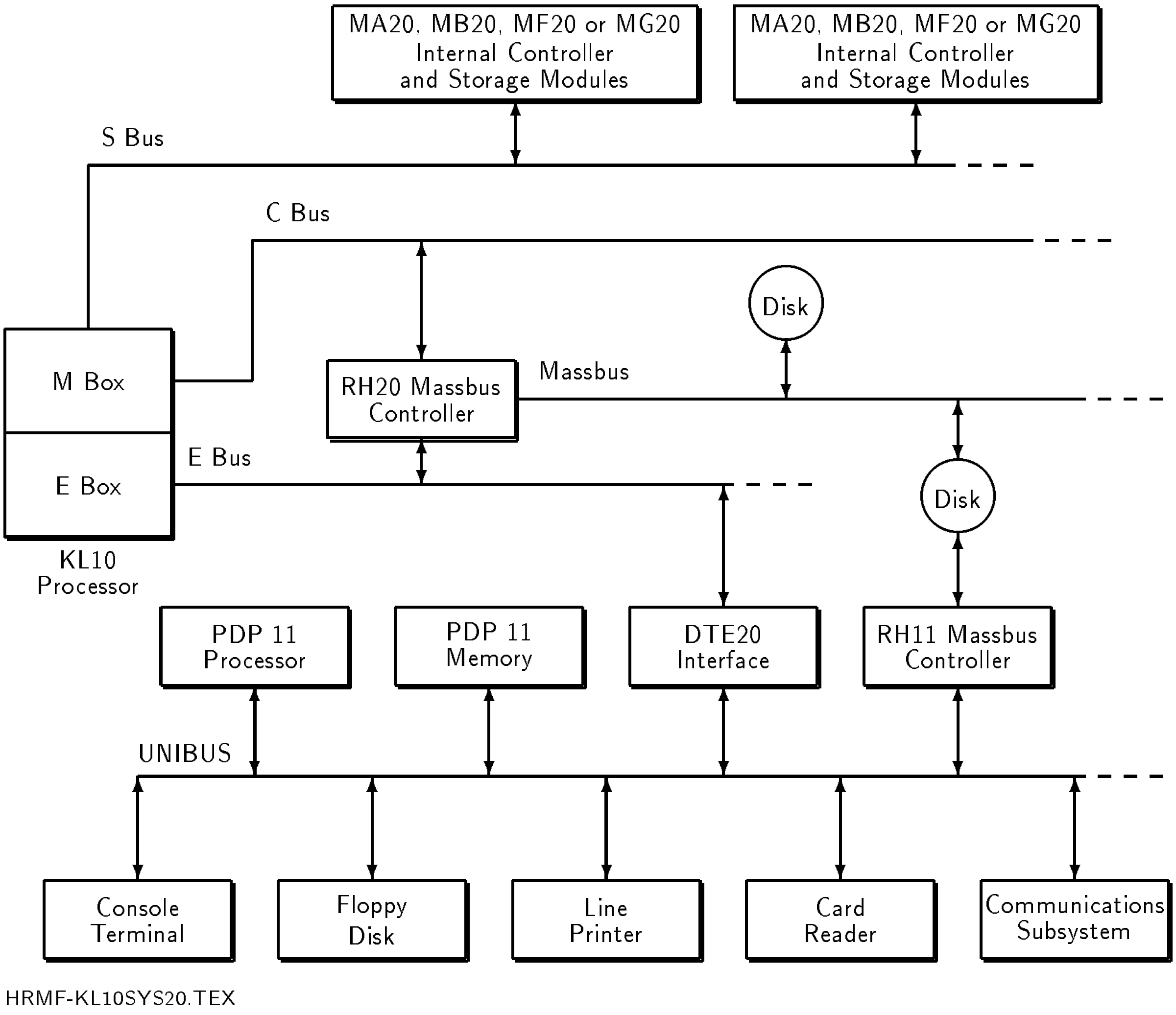
There are also two versions of the operating system for use with the KL10: the TOPS–20 Monitor and the TOPS–10 Monitor. The Extended KL10 with both user and executive space extended is available only in TOPS–20 systems. In a TOPS–10 system, an Extended KL10 can have extended addressing only in executive space, and for this it must run microcode version 271 or greater (in which case, the TOPS–10 Monitor actually uses so–called "TOPS–20 paging"). In other words and Extended KL10, regardless of Monitor, has TOPS–20 paging; in a single–section KL10 the paging always matches the Monitor.

1.2.1 The KL10 Processor

---

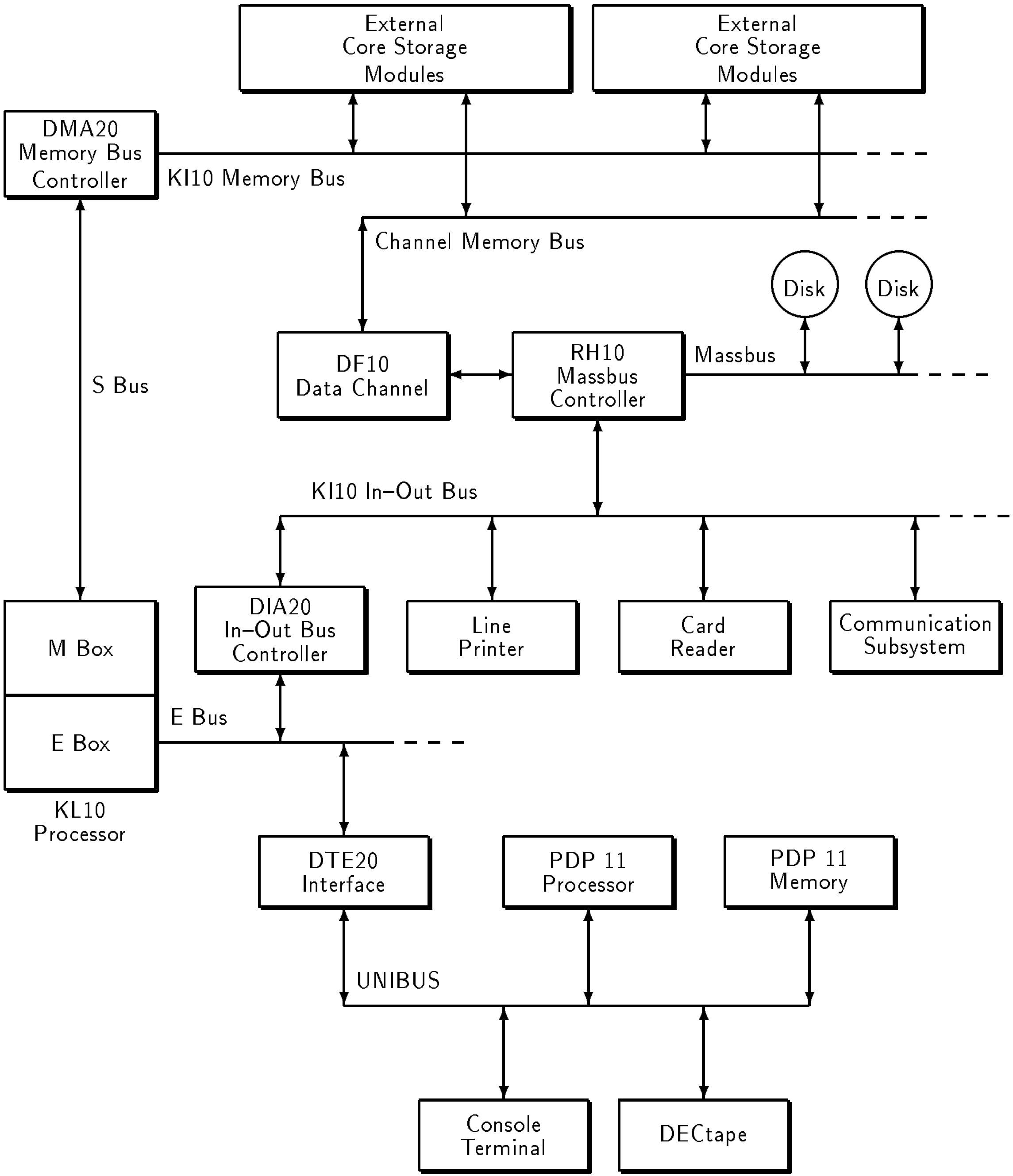
[1] MOS and core memory cannot be mixed on the same bus. If the system includes both, there must be two S buses.

Figure 1.3: KL10–based DECSYSTEM–20



HRMF-KL10SYS20.TEX

Figure 1.4: KL10–based DECsystem–10



HRMF-KL10SYS10.TEX

Figure 1.5: KL10 Processor Simplified

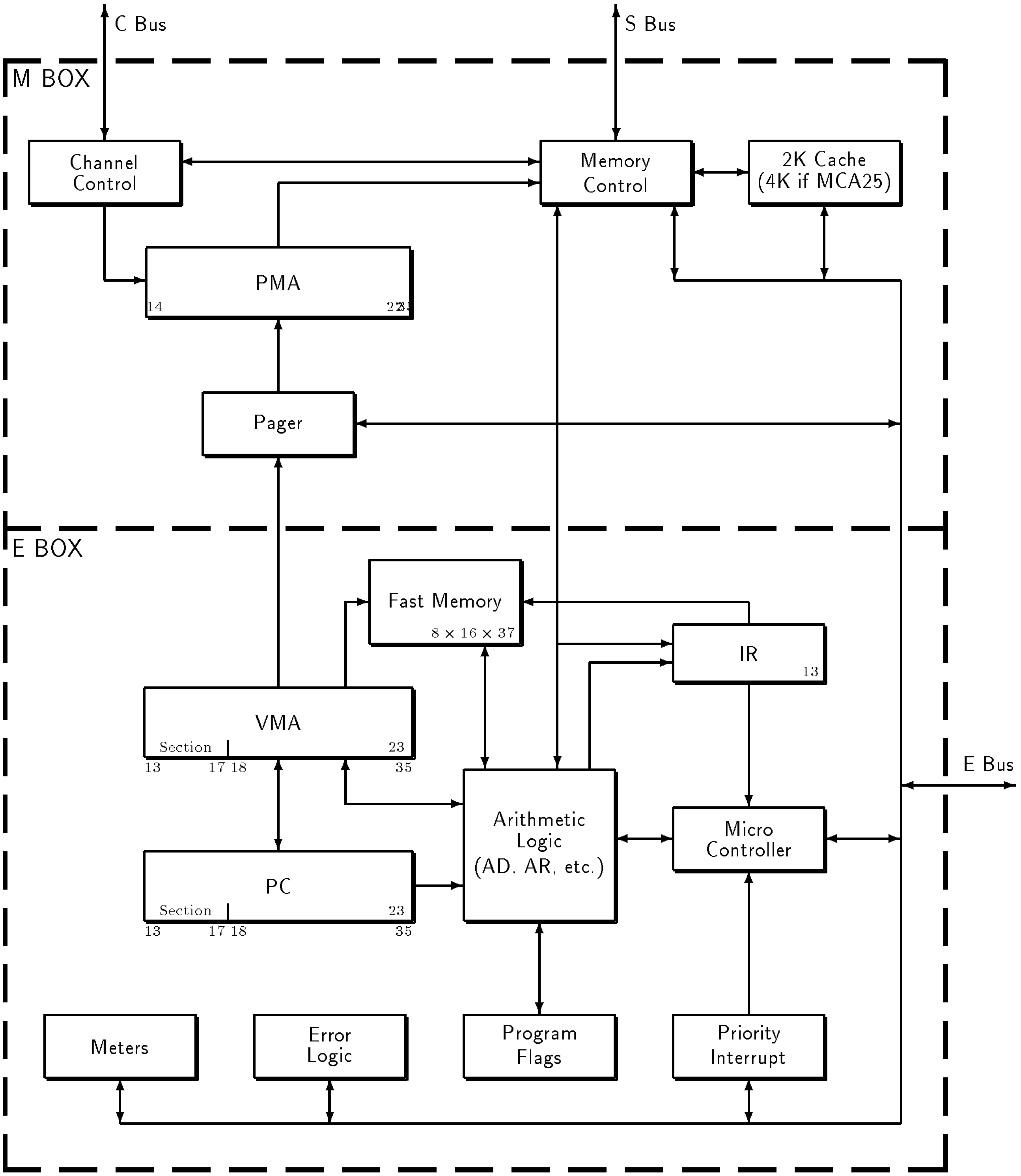


Figure 1.5 shows the internal configuration of the KL10 processor. Of the registers shown, only PC (the program counter), Program Flags, and one set of sixteen Fast Memory accumulators are directly relevant to a typical user. The processor performs a program by executing instructions retrieved from the memory locations addressed by PC. For the normal program sequence, PC is regularly incremented by one so that instructions are taken from consecutive locations. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time in a skip instruction or by replacing its contents with the value specified by a jump instruction. Throughout the text the phrase "jump to location n" means to load the value n into PC and continue performing instructions in the normal counting sequence beginning at the location then specified by PC. Physically PC is not a counter at all—it just holds the program count; the actual counting is done in the virtual memory address register VMA. The entire VMA functions as a counter, but no carry is allowed into the section part in program counting. Hence, large data structures can arbitrarily cross section boundaries, but the program cannot. The program count wraps around in the current PC section, which is specified by PC bits 13–17. For the program to go from one section to another requires an explicit transfer of control by jumping to another section. In a single–section KL10, all section bits are held at zero, so VMA and PC function as 18–bit registers. The virtual address from VMA, whether eighteen bits or twenty-three, is translated by the pager to a 22–bit physical address that is supplied to memory via PMA.

Each instruction retrieved from memory contains information identifying the operands and an instruction code specifying the operation to be performed using those operands. The code goes to the instruction register IR, from which it is decoded by the microcontroller, which in turn performs the instruction by manipulating all of the other E box elements and making the necessary requests to the M box. The microcontroller also executes the more fundamental operations of sequencing the program, handling TOPS–20 paging operations beyond the basic address translation made by the pager (TOPS–10 operations are built into the M box pager), processing interrupts, and so forth. (Not shown in the illustration is a multitude of control lines emanating from the microcontroller and extending throughout the machine.) The microcontroller operates from a microcode contained in a control store. This microcode bears the same relation to the microcontroller as the program does to the processor. Microprocessing is invisible to the programmer, and he need not be concerned with the microcode except to the extent of loading it at system initialization. The reader should, however, note an important implication of this type of processor implementation: a single KL10 processor can actually be any one of a number of different processors merely by loading different microcodes.

The major working area of the processor is the arithmetic logic. This contains three full–word registers: the arithmetic register (AR), the buffer register (BR), and the multiplier–quotient register (MQ). For handling double–length operands, AR and BR have 36–bit right extensions, called ARX and BRX, respectively, Various combinations of these registers play a role in all arithmetic, logical, and data handling operations and in program control operations as well. Also included in the arithmetic logic are an extremely fast, double–length adder, AD–ADX, and smaller registers that handle floating–point exponents and control shift operations and byte manipulation. However, from the point of view of the programmer, the arithmetic logic can be disregarded. Almost all of the operations necessary for the execution of a program are performed in it, but it never retains any information from one instruction to the next. Computations either affect control elements, such as PC and the program flags, or produce results that are stored and must be retrieved if they are to be used as operands in other instructions. The program flags report conditions of interest to the programmer, such as arithmetic and stack overflow; some of these conditions also are reported via program traps.

An instruction word has only one 18–bit address field for addressing any location in the current PC section. Most instructions have two 4–bit fields for addressing the first sixteen memory locations. Any instruction that requires a second operand has an accumulator address field which can address one of these sixteen locations as an accumulator; in other words as though it were a result held over in the processor from some previous instruction. (The programmer usually has a choice of whether the result of the instruction will go to the location addressed as an accumulator, to that addressed by the 18–bit address field, or to both.) Every instruction has a 4–bit index–register address field which can address fifteen of these locations for use as index registers in modifying a memory address (a zero index–register address specifies no indexing). Although all computations on both operands and addresses are performed in the arithmetic logic, the computer actually has sixteen accumulators, fifteen of which can double as index registers. The factor that determines whether one of the first sixteen locations in memory is an accumulator or an index register is not the information it contains nor how its contents are used, but rather how the location is addressed. These first sixteen locations are not actually in the storage modules—they are in a fast memory contained in the processor. This allows much quicker access to these locations, whether they are addressed as accumulators, index registers, or ordinary memory locations. They can even be addressed from the program counter, and provision is made for referencing them from non–zero sections. Moreover, there are actually eight of these fast memory blocks (also referred to as "AC blocks"), but generally only one is available to a program at any given time. Blocks 6 and 7 are reserved specifically for the microcode; the Monitor usually reserves block 0 for itself and assigns the others to user programs.

An optional feature that speeds up memory access and increases the efficiency of storage module use is a cache. This facility has 2048 locations that temporarily substitute for a selection of the most-frequently used storage locations. Hence, the cache may be regarded in some respects as a set of general–purpose registers. A program loop once read from storage and then resident in the cache may be executed hundreds of times without further instruction fetches from storage. Data produced by the program is written in the cache. Thus, if the program sets up a location to be a counter, that location may be read and written thousands of times with no storage access, even initially. When the cache is present but does not contain the word the program wants, memory control gets a group of four adjacent words from storage, including the requested one, and places them in the cache, on the assumption the program will probably want the other three and can thus get them more quickly. This is a reasonable assumption, since the program counts sequentially and data manipulation is frequently sequential as well. Cache control has a mechanism for determining frequency of use, and it writes the least–recently used word groups back into storage (or discards them if unchanged) when the cache space is needed for new references. The only address restriction on the 512 4–word groups is that the cache can have the same groups from, at most, four pages. There may be complete pages in the cache, but it is more likely to have a selection of groups from a selection of pages depending on frequency of use. Generally the cache contains words for the current user and for the Monitor, as well as for handling interrupts for many users. The reader should be aware that the cache contains representations of memory word groups, not necessarily the actual storage contents. For example, when the program writes a word, the contents of that cache location then differ from the contents of the corresponding storage location, and the other words in the group may not even be in the cache. This caution is of interest, however, only to the operating system. A typical program simply makes memory references; the more of these in which the cache substitutes invisibly for storage, the better.

The MCA25 cache and pager upgrade (part of the extended KL10) expands the size of the cache to 4096 words, organized as 1024 four–word groups; this cache can hold the same groups from, at most, eight pages.

Also included within the processor are a number of internal devices that are similar to external controllers in that they operate independently of the program but are controlled by it over the E bus. Some of these have already been mentioned: the program sets up the pager, instructs cache control to update storage, sets up the memory system, and gets diagnostic information from the memory controllers and storage modules. Other such "devices" are the error logic, the meters, and the priority interrupt. By means of the error logic, the program can monitor conditions in the processor. The meters provide a time base, an interval counter, and facilities for keeping track of program use of the system and for analyzing system performance. The interrupt facilitates processor control of the entire system by means of a number of priority–ordered levels over which external signals may interrupt the normal program flow. The processor acknowledges an interrupt request by executing the instruction contained in a particular location for the level or by doing some special operation specified by the device (such as incrementing the contents of a memory location). Assignment of levels to devices is entirely under program control. Two of the devices to which the program can assign levels are the error logic and the interval counter.

1.2.2 KL10 Memory

When dealing with storage modules, the processor need not wait the entire memory cycle time. To read, the processor waits only until the information is available and then continues its operations, regardless of whatever else the memory must do to complete the read cycle. To write, the processor waits only until the data is accepted; the memory then performs an entire cycle to write that data. To save time in an instruction that fetches an operand and then writes new data into the same location, the processor can request a read–modify–write cycle from the memory, in which the memory performs only the read part initially and then completes the cycle when the processor supplies the new data. This procedure is not used however in a lengthy instruction (such as multiply or divide), which would tie up a storage module that may be needed by some other processor. Such instructions instead request separate read and write access. However, the above considerations apply only when the cache is not in use or is not present, thus requiring that the processor always deal with the storage modules and that it request one word at a time.

With the cache in use for a given page, memory access is handled using the cache wherever possible, and when storage access is required, transfers are in 4–word groups. For a read request, the M box reads from the cache if the word is there; otherwise, it initiates a storage–to–cache transfer, which may require a prior cache–to–storage transfer to make room for the new data. For a write request, the M box always writes in the cache, and this too may require a cache–to–storage transfer to make room; otherwise, the M box writes in storage only when the cache is not in use, the Monitor specifically updates memory, or the data is supplied by an internal channel.

For handling storage transfers for a channel or with a cache, the M box interprets physical addresses in this format:

When the E box requests a word that is not in the cache, the M box gets the four words in the group specified by bits 27–33 or, more specifically, gets whichever of them are not already in the cache. For the quickest possible service, the M box first gets the particular word requested; e.g., if the program requests word 2 in a group, the M box retrieves word 2 first, followed by words 3, 0, and 1. Even without a cache, channel transfers are always in groups of four, except perhaps for the first or last group in a block. Except with an MF20 memory, the processor further increases the speed of memory operation by overlapping memory cycles: it can start one module to read a word before receiving a word previously requested from a different one. Such speedup is unnecessary with an MF20 memory because it is four words wide. Of course fast memory and the cache have no basic cycle; with them the processor reads or writes a word directly.

From the simple hardware addressing point of view, the entire physical memory is a set of locations whose addresses range from zero to a maximum dependent upon the capacity of the particular installation. In a system with the greatest possible capacity, the largest address is 17777777 (decimal 4,194,303). The whole memory would usually be made up of a number of storage modules of different capacities. Hence, a given address actually selects a particular module and a specific location within it. For a 64K module with 22–bit addressing, the high–order six address bits select the module, and the remaining sixteen bits address a single location in it; selecting a 32K memory takes seven bits, leaving fifteen for the location. The times given below assume the addressed memory is idle when access is requested. The processor can avoid waiting for its own previously requested memory cycles to end by making consecutive requests to different storage modules. With an MF20 or an MG20 memory, almost all transfers are of four words at a time, so there is seldom any conflict among requests. With other memories, and provided a cache is in use, ordering requests among modules can be guaranteed by interleaving them in sets of four in such a way that requests for the words in a group are cycled through the four modules in the set. Interleaving is effected by assigning four modules, each of n locations, to the same $4n$–location area of the address space, and setting each module to respond only to one request out of the four in a group. Hence, within the given area, all addresses ending in 0 or 4 are locations in one module, those ending in 1 or 5 are locations in a second, and so forth. Some of the earlier modules can be interleaved only in pairs, which is not as effective but is worthwhile. Without a cache, interleaving is not as effective, but it is still advisable since the program is sequential. Without interleaving or a cache, some alternation between modules is produced by keeping instructions in one and operands in another. Interleaving, assigning module numbers, and so forth, are done by the program for internal memories but by manual switch settings for external memories. Complete information is given in Appendix G.3.

The only physical locations uniquely defined by the hardware are those in fast memory, locations 0–17. All other hardware–defined addresses are relative to pages, such as the process tables, whose physical locations are specified by the Monitor. Physical memory in a system is a constant unless a storage module is actually added or removed. The virtual address space accessible to a particular program is entirely a function of the way in which the Monitor sets up user operating conditions, except that any space and any restrictions must encompass an integral number of pages.

1.2.3 Memory Characteristics

Table 1.1 gives the characteristics of the various memories for the two types of KL10 processor. Times are in microseconds, and for external memories they include the delay introduced by 10 feet (3 meters) of cable. Read access for a single word or the first word in a group is the time from the request until the word is in AR. For an entire 4–word group, read access is the time from the request until the last word is in the cache. Write access is the time from the request until the processor receives the memory acknowledgment, for either the first word or the fourth. Except for the MF20, these figures define the system access rates for storage modules with 4–way interleaving, since all

Table 1.1: KL10 Memory Characteristics

Physical Characteristics

	Number of Modules	*Size*
MF10 Core Memory	1	32K, 64K
MG10 Core Memory	2	64K, 128K
MH10 Core Memory	2	128K, 256K
MA20 Core Memory	4, 8	64K, 128K
MB20 Core Memory	4, 8	128K, 256K
MF20 MOS Memory	1	256K
MG20 MOS Memory	1	1024K
KL10 Fast Memory		16 [×8 sets]
KL10 Cache		0 or 2K (MCA20) or 4K (MCA25)

Extended Processor Timing

	First or Single–Word Access		*Four–Word Access*	
	Read	*Write*	*Read*	*Write*
MF10 Core Memory	1.493	1.084	2.227	1.484
MG10 Core Memory	1.553	1.134	2.287	1.534
MH10 Core Memory	1.633	1.134	2.367	1.534
MA20 Core Memory	.883	.40	1.467	1.60
MB20 Core Memory	1.017	.40	1.60	1.60
MF20 MOS Memory	.800	.267	1.40	.667
MG20 MOS Memory	.550	.160	1.10	.550
KL10 Fast Memory	.067	.067		
KL10 Cache	.133	.133		

Single–section Processor Timing

	First or Single–Word Access		*Four–Word Access*	
	Read	*Write*	*Read*	*Write*
MF10 Core Memory	1.627	1.217	2.507	1.697
MG10 Core Memory	1.687	1.267	2.567	1.747
MH10 Core Memory	1.767	1.267	2.647	1.747
MA20 Core Memory	1.06	.48	1.76	1.92
MB20 Core Memory	1.22	.48	1.92	1.92
KL10 Fast Memory	.080	.080		
KL10 Cache	.160	.160		

memory operations are absorbed within them: by the time the processor receives the data or the acknowledgment, it can make a new request, for which the memory will be ready. Sizes given are those in which the units are available. Note that interleaving depends on the number of modules, not the number of units, most of which contain more than one module. Hence, 4–way interleaving can be done with a single MA20 or MB20 memory, whereas it requires two MH10s or MG10s and four MF10s.

With MF20 memories, there is only one module per unit and interleaving is not used. (Each controller can handle three units or "groups".) The times given in the table are the actual times the processor must wait to get data or an acknowledgement, except that hitting a refresh cycle can cause a delay of up to 533 ns (refreshing requires about 3.5% of total memory time). Following a read, the processor can make another request immediately. Following a write, it must wait from 467 to 867 ns before another request can be handled by the same controller. However, since a single MF20 handles four words at once, one request following another within that time is unlikely.

Fast memory times are for referencing a memory location for an operand; a fast memory instruction fetch takes slightly more time than a cache access. When a fast memory location is addressed as an accumulator or index register, the access time is considerably shorter and usually takes no time at all, as it is done in parallel with instruction operations that are required anyway.

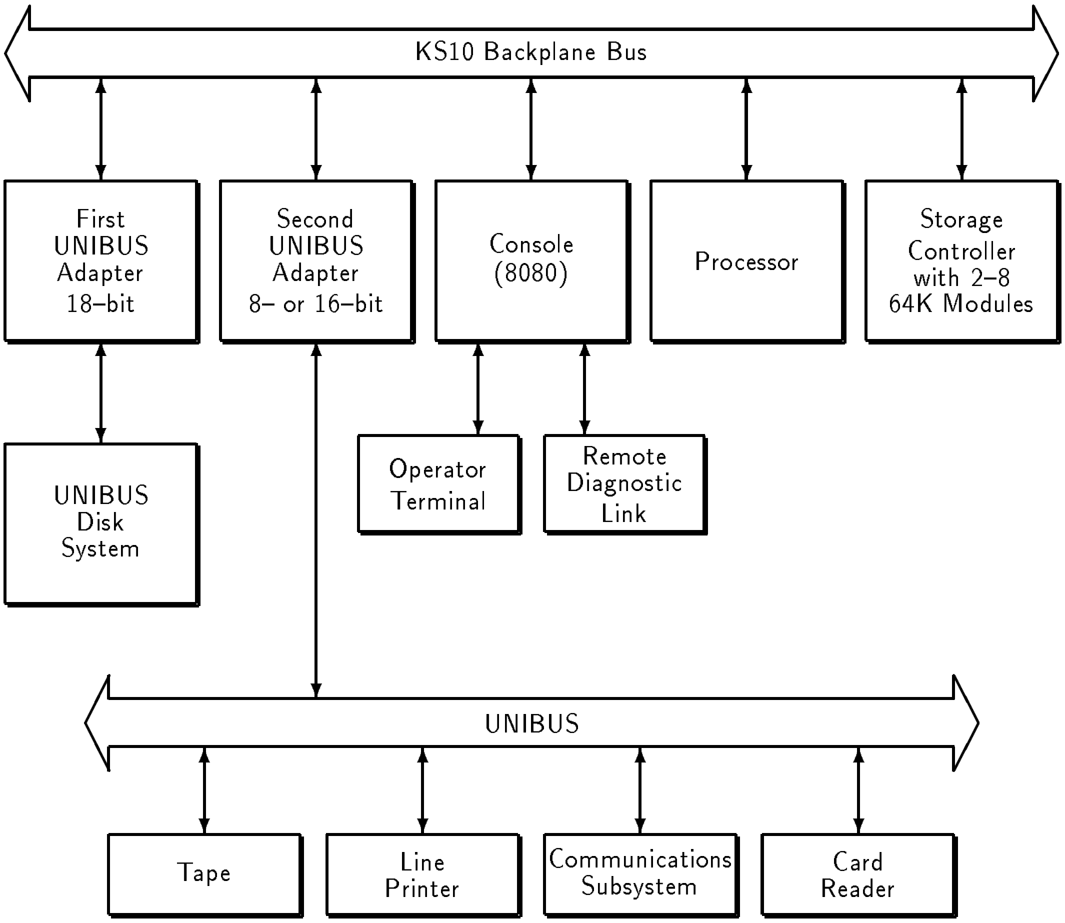
The MF20 and MG20 have a 7–bit error correction code; all other units have only a single parity bit. The MF20 and MG20 also have a spare bit that can be substituted for a known bad bit.

1.3 KS10–based System Organization

Figure 1.6 and Figure 1.7 show the organization of the DECSYSTEM–2020 and the KS10 processor used in it. The overall system (Figure 1.6) comprises a number of major units or subsystems that communicate with one another over a bus built into the backplane. The minimal system has five subsystems: processor, MOS storage, console, and two in–out subsystems, each based on a Unibus. One Unibus adapter handles the disk system, the second handles all other peripheral equipment. Depending on the device, these adapters can make direct access to storage or request that the processor handle the transfer via the program. The console, which is based on a microprocessor, boots the system from disk and handles interaction of the operator or a remote diagnostic link with the other subsystems. The backplane bus and most other full–word data paths are actually thirty–eight bits, having a parity bit for each half word. The system can run under either the TOPS–20 or TOPS–10 Monitor.

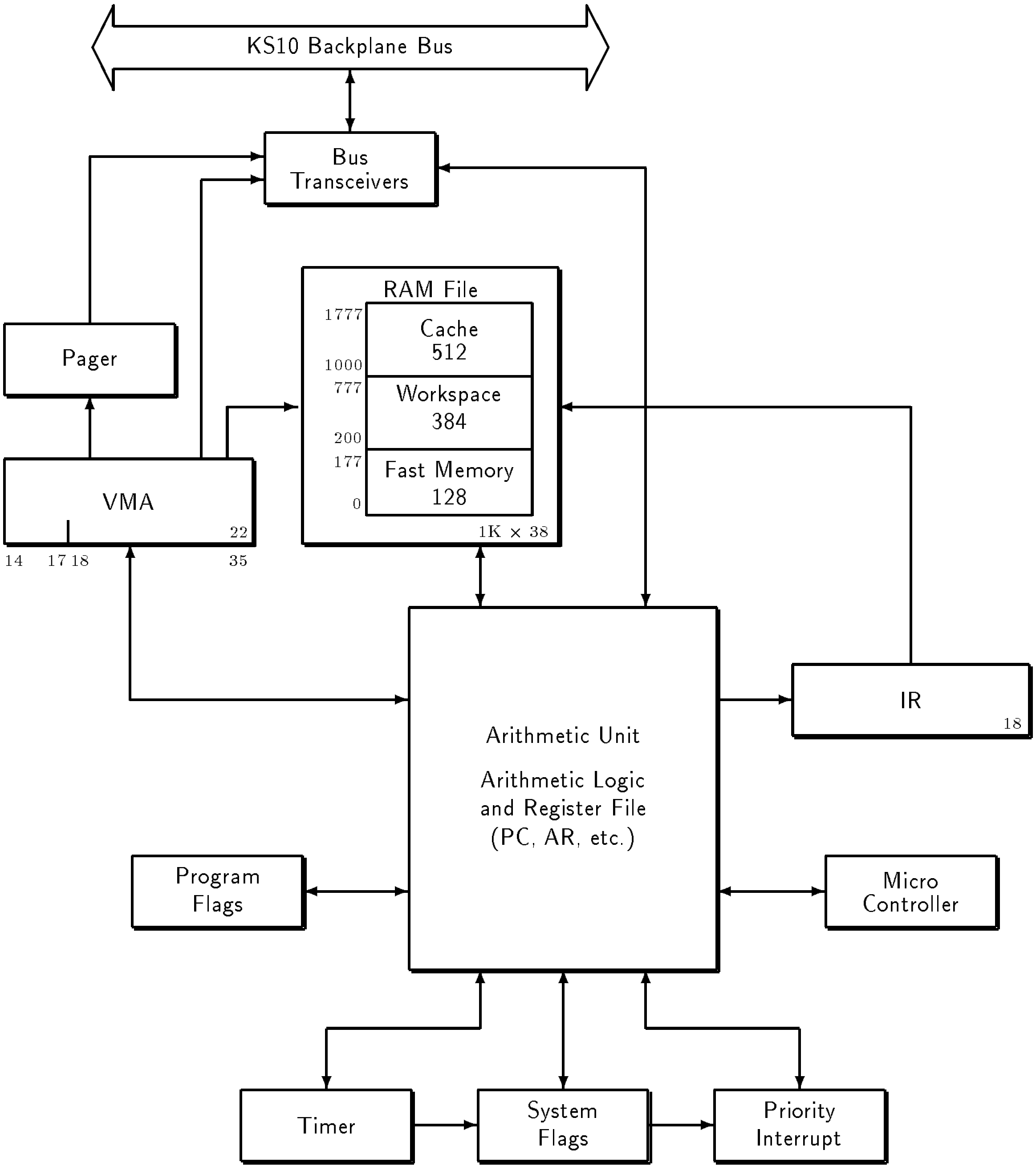
Of the elements shown in the processor illustration (Figure 1.7), only fast memory, the program flags, and the program counter PC are directly relevant to a typical user. The processor performs a program by executing instructions retrieved from the memory locations addressed by PC. For the normal program sequence, PC is regularly incremented by one so that instructions are taken from consecutive locations. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time in a skip instruction or by replacing its contents with the value specified by a jump instruction. Throughout the text the phrase "jump to location n" means to load the value n into PC and continue performing instructions in the normal counting sequence beginning at the location then specified by PC. Physically PC is not a counter at all—it is a register in the register file (described below). This register just holds the program address, and the actual counting is done by the arithmetic logic, which wraps the count around in eighteen bits because the virtual space is limited to section zero. Addresses from PC, or calculated by the arithmetic logic, go

Figure 1.6: DECSYSTEM–2020



HRMF-KS10SYS20.TEX

Figure 1.7: KS10 Processor Simplified



to the virtual memory address register VMA. Each virtual storage address from VMA is translated by the pager to a 20–bit physical address that is supplied to the storage subsystem via the bus. VMA actually has twenty–two bits, for handling both physical storage addresses and addresses for other types of bus transactions, such as those to the console, to in–out equipment, and to memory status.

Each instruction retrieved from memory contains information identifying the operands and an instruction code specifying the operation to be performed using those operands. The code goes to the instruction register IR, from which it is decoded by the microcontroller, which in turn performs the instruction by manipulating all of the other processor elements and making the necessary requests for bus transactions. The microcontroller also executes the more fundamental operations of sequencing the program, handling paging operations beyond the basic address translation made by the pager, processing interrupts, and so forth. (Not shown in the illustration is a multitude of control lines emanating from the microcontroller and extending throughout the machine.) The microcontroller operates from a microcode contained in a control store. This microcode bears the same relation to the microcontroller as the program does to the processor. Microprocessing is invisible to the programmer, and he need not be concerned with the microcode except to the extent of loading it at system initialization. The reader should, however, note an important implication of this type of processor implementation: a single KS10 processor can actually be any one of a number of different processors merely by loading different microcodes.

The major working area of the processor is the arithmetic unit. Central to this unit is a set of ten 4–bit microprocessor slices, which together contain the full–word arithmetic logic and a file of ten registers. The register file includes, besides PC, the arithmetic register (AR); other associated registers used in manipulating data and performing arithmetic and logical operations; and registers that contain system addresses, status information, and constants. The arithmetic logic includes a full–word adder, shifter, and mixers. It also contains complete 10–bit logic for direct manipulation of floating–point exponents, standard 7–bit bytes, and for controlling shifting and operations on bytes of other sizes. Multiple–length operands are handled by separately manipulating their higher– and lower–order words using the registers in the file. Like the microcontroller, the arithmetic unit (except for PC) can be disregarded by the user. Almost all of the operations necessary for the execution of a program are performed in it, but it never retains any information from one instruction to the next. Computations either affect control elements such as PC and the program flags, or produce results that are stored and must be retrieved if they are to be used as operands in other instructions. The program flags report conditions of interest to the programmer, such as arithmetic and stack overflow; some of these conditions may also be reported via program traps. (Several registers in the file do retain information of interest to the system programmer, however.)

An instruction word has only one 18–bit address field for addressing any location in the virtual space. Most instructions have two 4–bit fields for addressing the first sixteen memory locations. Any instruction that requires a second operand has an accumulator address field which can address one of these sixteen locations as an accumulator; in other words, as though it were a result held over in the processor from some previous instruction. (The programmer usually has a choice of whether the result of the instruction will go to the location addressed as an accumulator, to that addressed by the 18–bit address field, or to both). Every instruction has a 4–bit index–register address field which can address fifteen of these locations for use as index registers in modifying a memory address. (A zero index–register address specifies no indexing.) Although all computations on both operands and addresses are performed in the arithmetic unit, the computer actually has sixteen accumulators, fifteen of which can double as index registers. The factor that determines whether one of the first sixteen locations in memory is an accumulator or an index register is not the information it contains nor how its contents are used, but rather how the location is addressed. These first sixteen locations are not actually in the storage modules—they are in a fast memory contained in the processor. This allows much quicker access to these locations, whether they are addressed as accumulators, index registers, or ordinary memory locations. They can even be addressed from the program counter. Moreover there are actually eight of these fast memory blocks (also referred to as "AC blocks"), but generally only one is available to a program at any given time. Block 7 is reserved specifically for the microcode; the Monitor usually reserves block 0 for itself and assigns the others to user programs.

A feature that speeds up memory access and increases the efficiency of storage module use is a virtual cache. This facility has 512 locations that duplicate the contents of storage locations in current use in the virtual address space of the program. Every time a word is read from storage or written in storage, it is also written in the cache location selected by the right–most nine virtual address bits, which represent position within the virtual page. Provided there is no intervening reference to the same position in some other page, a subsequent read reference to the same virtual location can be made to the cache (referred to as a "cache hit") instead of going over the bus to storage. A program loop, once read from storage and then resident in the cache, may be executed hundreds of times without further instruction fetches from storage; and data produced by the program can be retrieved without requiring bus transactions. To a great extent the cache is also invisible. A typical program simply makes memory references; the more of these in which a word is read from the cache instead of storage, the better. However, a program that tends to settle in one virtual page at a time, instead of alternating references among a number of pages, will maintain a much higher cache hit rate, saving considerable time.

Fast memory and the cache are contained respectively in the bottom 128 and top 512 locations in a RAM file in the processor. The remaining 384 locations are a workspace used by the microcode as a scratch pad and for handy storage of various system quantities and constants that expedite the execution of the more complicated instructions. Also included within the processor are several elements, such as the pager already mentioned, that are similar to external controllers in that they operate independently of the program but are controlled by it. The timer provides a time base and an interval counter. By means of the system flags, the program can monitor various conditions throughout the system and can interrupt the console or be interrupted by it. The interrupt facilitates processor control of the entire system by means of a number of priority–ordered levels over which external signals may interrupt the normal program flow. The processor acknowledges an interrupt request by executing the instruction contained in a particular location for the level or the source of the request. Assignment of levels is entirely under program control. Two levels can be assigned to each Unibus adapter, and one can be assigned to the system flags.

1.3.1 KS10 Memory

Any subsystem can request use of the bus to write a word into storage or read a word from it. To save time in byte input operations, a Unibus adapter can also get the bus for a read–modify–write cycle. In this transaction a word goes from memory to the adapter, which inserts the byte and immediately sends the modified word back. A requesting subsystem may have to wait until the bus is free and it has priority, and even then there may occasionally be a further wait of up to 750 ns for memory refresh (which requires about 5% of total memory time). Reading from storage takes 900 ns. Writing to storage takes 600 ns, although the memory remains busy for an additional 300 ns. Whenever the processor writes or reads a word in storage, that word is automatically written in the cache. Thus, if the processor wishes to read the same word at a later time, retrieval requires only 300 ns. The cache hit rate is generally about 80%.

The following table gives the characteristics of KS10 memory with times in nanoseconds.

	Read	*Write*	*Size*	*Error Facility*
MOS Memory	900	600	128K–512K	7–bit correction code
Fast Memory	300	300	16	2 parity bits
Cache	300		512	2 parity bits

There is no cache write time, because writing is automatic and is absorbed in storage access time. Fast memory times are for addressing accumulators as memory locations. Access to an accumulator as an accumulator or as an index register is made in a single microinstruction period of 150 ns; frequently this represents no extra time, because the same microinstruction often performs other functions.

The memory array comprises from two to eight storage modules of 64K each. From the hardware addressing point of view, the entire physical memory is simply a set of locations whose addresses range from zero to a maximum dependent upon the capacity of the particular installation. In a system with the greatest possible capacity, the largest address is 1777777 (decimal 524,287).

At a halt, the microcode places a halt code and PC in *storage* locations 0 and 1. The only other physical locations uniquely defined by the hardware are those in fast memory, locations 0–17. All other hardware–defined addresses, such as the process tables or the halt–status block, are relative to physical locations specified by the Monitor. Physical memory in a system is a constant unless a storage module is actually added or removed. The virtual address space accessible to a particular program is entirely a function of the way in which the Monitor sets up user operating conditions, except that any space and any restrictions must encompass an integral number of pages.

1.4 Timesharing

Inherent in the machine hardware are restrictions that apply universally: only certain instructions can be used to respond to a priority interrupt, and certain memory locations have predefined uses. Above this fundamental level, the timeshare hardware provides for different modes of processor operation and establishes certain instruction and memory restrictions so that the processor can handle a number of user programs (programs run in user mode) without their interfering with one another. The memory restrictions are dependent to a great extent on the type of processor; however, the instruction restrictions are not, and these are relatively obvious: a program that is sharing the system with others cannot usually be allowed to halt the processor or to operate the in–out equipment arbitrarily. (Some processors permit unrestricted access to a limited set of in–out devices for the use of special real–time applications.) A program that runs in executive mode—the Monitor—is responsible for scheduling user programs, servicing interrupts, handling input–output needs, and taking action when control is returned to it from a user program. Any violation of an instruction or memory restriction by a user transfers control back to the Monitor. Dedication of the entire facility to a single purpose, i.e., operation for only one user, is equivalent to operation in executive mode.

The paging hardware maps pages from the virtual address space into pages anywhere in physical memory. A page map for each program specifies not only the correspondence from virtual address to physical address, but also whether or not an individual virtual page is accessible and alterable, and whether or not the cache can be used for references to it. In the KL10 and KI10, both user and executive modes are subdivided according to whether the program is running in a public area or a concealed area; these areas are distinguished by whether or not their pages are labeled public. Within user mode these submodes are public and concealed; within executive mode they are supervisor and kernel. A program in concealed mode can reference all accessible user memory, but the public program cannot reference the concealed area except to transfer control into it at certain legitimate entry points. The concealed area would ordinarily be used for proprietary programs that the user can call but cannot read or alter.

In the XKL-1 and KS10, all pages may be regarded as concealed, because none are labeled public; but in reality the concept of public *vs* concealed simply does not apply. In the XKL-1 and KS10, executive mode is identical to kernel mode in that supervisor restrictions do not exist. In this treatment of timesharing, any mention of public in contrast to private is irrelevant to the XKL-1 and KS10, and functions indicated as being performed by the kernel or supervisor program are all handled by the executive in these processors.

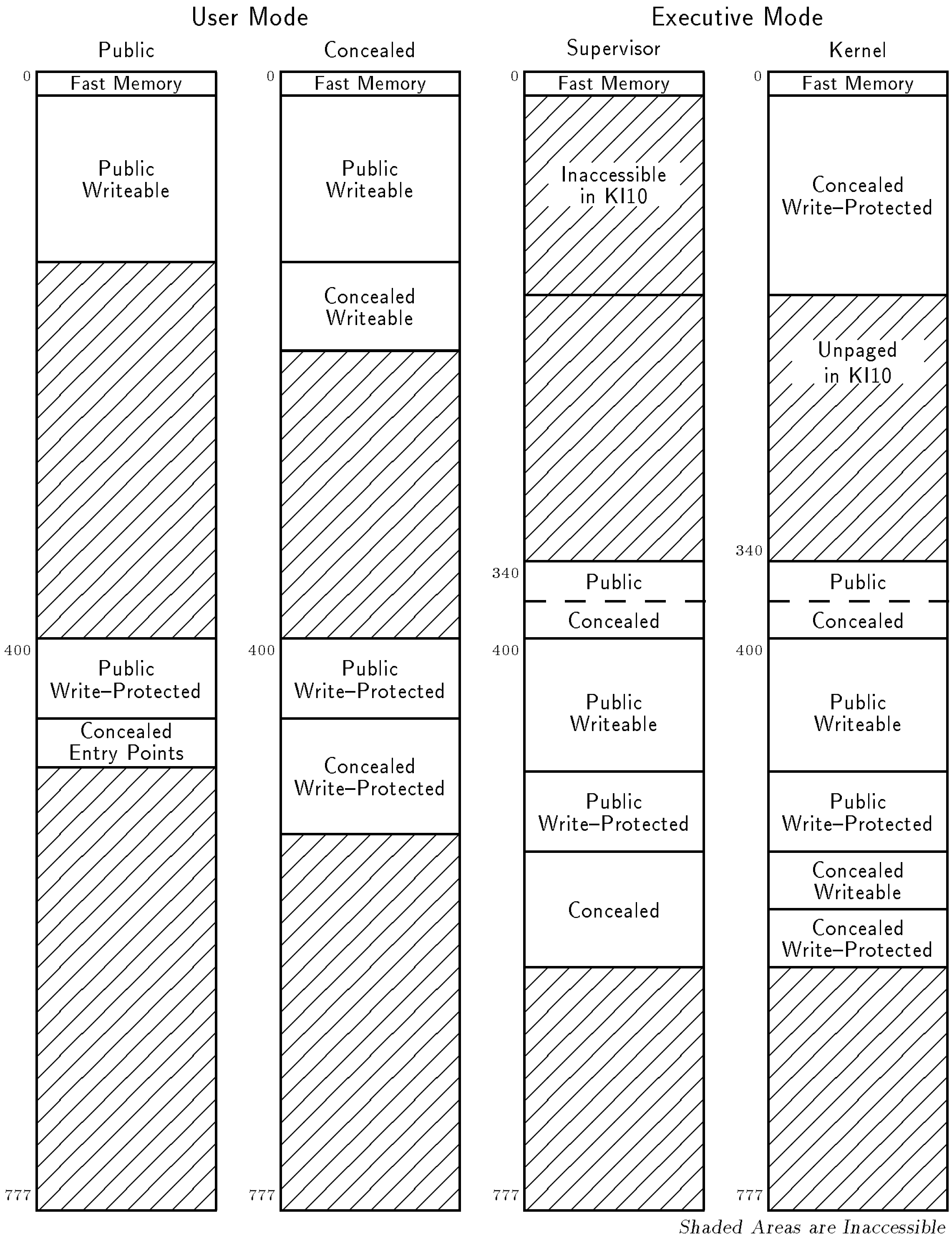
In kernel mode the Monitor handles the in–out for the system, handles priority interrupts, constructs page maps, and performs those functions that affect all users. This mode has no instruction restrictions, the program can even turn off the pager to address memory directly, using physical addresses; the address space is then said to be unpaged. In paged address space, individual pages may be restricted as inaccessible or write–protected, but it is the kernel program that establishes these restrictions. In supervisor mode the Monitor handles the general management of the system and those functions that affect only one user at a time. This mode has essentially the same instruction and memory restrictions as user mode, although the supervisor program can read, but not alter, the concealed areas; in this way the kernel mode Monitor supplies the supervisor program with information the latter cannot affect, even though the locations are not write–protected in kernel mode. The kernel program generally assigns fast memory block 0 for use by the Monitor in either mode (especially in a TOPS–10 system—to be compatible with the KI10 where the hardware requires it). Typically, the Monitor assigns block 1 to all users and uses blocks 2 and 3 for handling interrupts (e. g. block 2 just for the highest priority level and block 3 for the others).

The most extensive hardware features for timesharing exist in the KL10 and KI10. The reason for this is that the newest software is much more sophisticated and thus requires less hardware to do the job— the XKL-1 and KS10 take advantage this fact to cut cost.

Figure 1.8 shows an example of the use of the most extensive timeshare hardware. This diagram shows the layout of a single–section KL10 address space that is configured to make full use of the various modes, be used with a TOPS–10 Monitor, and be compatible with earlier machines. The space is 256K, made up of 512 pages numbered 0–777 octal. Any program can address locations 0–17, because these are in fast memory and are completely unrestricted (although the same addresses may be in different blocks for different programs). The public user program operates in the public area, part of which may be write–protected. The public program cannot access any locations in the concealed areas except to fetch instructions from prescribed entry points. The concealed user program has access to both public and concealed areas, but it cannot alter any write–protected location, whether public or concealed, and fetching an instruction from the public area automatically returns the processor to public mode.

In a TOPS–20 system, an area labeled "write–protected" might better be called "copy–on–write." Write protection is generally for "pure" code (i.e., those portions of the program that are not expected to be changed by the execution of the program) shared by a number of users. If one user attempts to alter a copy–on–write page, the TOPS–20 Monitor will ordinarily make a separate copy

Figure 1.8: Possible TOPS–10 Virtual Address Space Configuration



HRMF-T10VASC.TEX

for that user in his alterable space and keep the original write–protected page for the remaining users to continue sharing.

In our example, write–protected user pages are in the high address half of the space for compatibility with the two–part protection and relocation scheme of the KA10. We define the supervisor program as confined to pages 340 and above, even though there is actually nothing to prevent it from reading that part of the kernel program shown in the lower numbered pages. The reason for specifying it this way is for compatibility with the KI10, where the bottom 112K of executive space is unpaged and accessible only in kernel mode. Part of the executive public area may be write–protected; and, even though the supervisor can read concealed information, it cannot change a concealed location whether write–protected or not. For executive concealed areas, the distinction between writable and write–protected applies only to kernel mode. As in the case of concealed user mode, when the kernel program fetches an instruction from a public area, the processor returns to supervisor mode. With TOPS–10 paging, pages 340–377 constitute the per–process area, which contains information specific to individual users and whose mapping accompanies the user page map. In other words, the physical memory corresponding to these virtual pages can be changed simply by switching from one user to another, rather than the Monitor changing its own page map.

In the executive space of an XKL-1 there is no requirement to use section zero; the TOPS–20 monitor avoids all use of section zero.

In executive space of an extended KL10, the interrupt code must be in section zero. The rest of the KL10's executive program is usually in section one; but the two sections are mapped identically, so a given in–section address in either section refers to the same physical location. In terms of instructions implemented and procedures used, the KS10 acts like an extended processor that is confined to section zero.

A single–section user program would ordinarily be run in section zero for compatibility with an unextended processor. For the multisection case, the program might be in section 1, special tables in section 3, and a large data structure, such as an immense matrix, might occupy sections 10–12.

To manage the system effectively, the Monitor keeps a special table for each process in each processor. These process tables are defined in physical memory; each requires a single page whose whereabouts must be specified by the Monitor, which keeps an executive table for itself and a user table for each user. In a TOPS–10 processor, the first half of the table holds the page map for the process;[2] in a TOPS–20 processor, the process table contains a table of (super–) section pointers to page maps for whatever (super–) sections are in use. The hardware defines the use of many other locations in the process tables, especially in the KL10: these include locations that hold trap and interrupt instructions, control blocks for channels and front–end processors, and various quantities associated with paging and the meters. In the KS10 there are no control blocks since there are no channels or front–end processors; moreover timing information and many of the words associated with paging are kept in the workspace instead of the process tables. In the XKL-1, many of the parameters that control the paging environment are kept in MemA. Parts of a process table not used by or set aside for the hardware are available to the software. In each user process table the Monitor generally keeps a stack for use with the process, job tables, and various user statistics such as memory space and billing information. In the text the phrase "user process table" refers to the table currently specified by the Monitor as the one for the user, even if that user is not currently running.

---

[2] This distinction is no longer strictly true: advanced versions of TOPS–10 use TOPS–20 paging.

1.5 Number System

Fundamentally, the computer memory stores 36 bits (i.e., binary digits) in each word. The interpretation of the contents of a memory word, whether as a fixed–point number, as text, as a floating–point number, as an instruction, or whatever else, rests entirely with the programmer's selection of which instruction(s) interpret the data. This section discusses two broad classes of data: fixed–point and floating–point numbers.

1.5.1 Fixed–Point Numbers

One of the usual interpretations of a data word is as a signed integer with 35 magnitude bits. However, a program can interpret a data word as a 36–bit, unsigned binary number, or the left and right halves of a word can be taken as separate 18–bit numbers. The PDP–10 repertory includes instructions that add or subtract one from both halves of a word, so the right half can be used for address modification when the word is addressed as an index register, while the left half is used to keep a control count.

The fixed–point arithmetic instructions use twos–complement representations to do binary arithmetic. In a word used as a number, bit 0 (the leftmost bit) represents the sign: 0 for positive, 1 for negative. In a positive number the remaining thirty–five bits are the magnitude in ordinary binary notation. In a negative number the sign bit is 1 and the remaining bits are the twos complement of the magnitude.

Arithmetically, if x is an n–bit binary number, its twos complement is $2^n - x$. The twos complement is actually formed by computing the equivalent, $(2^n - 1) - x + 1$. Although the latter seems more complicated, it is quite easy to do in hardware. The expression $(2^n - 1) - x$ represents the ones complement (also called the logical complement, or simply the complement) of x. The ones complement is easy to compute because $(2^n - 1)$ is a word containing n consecutive 1s. When x is subtracted from a word containing all ones, the result is a word in which every bit that was 1 in x is now 0, and every bit that was 0 in x is now 1. Thus each bit of x has been changed to its complement. To form the twos complement of x, 1 is added to the ones complement of x. (The adder is directed to use the ones complement of its input as one operand, zero as the other operand, and a carry is injected at the right end of the adder; the result is the twos–complement of the orginal operand.)

$$+153_{10} \quad = \quad +231_8 \quad = \quad \underset{0 \hspace{20em} 35}{\boxed{000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 010\ 011\ 001}}$$

$$-153_{10} \quad = \quad -231_8 \quad = \quad \underset{0 \hspace{20em} 35}{\boxed{111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 101\ 100\ 111}}$$

A twos–complement addition actually acts as though the words represented 36–bit unsigned numbers; i.e., the sign bits are treated just the same as the magnitude bits. In the absence of a carry into the sign stage, adding two numbers with the same sign produces a plus sign in the result. The presence of a carry gives a positive answer when the addends have different signs. The result has a minus sign when there is a carry into the sign bit and the addends have the same sign, or when the addends have different signs and there is no carry. Thus the program can interpret the numbers processed in fixed–point addition and subtraction as signed numbers with thirty–five magnitude bits or as unsigned 36–bit numbers.

A computation on signed numbers produces a result that is correct as an unsigned 36–bit number even if overflow occurs. The hardware interprets the result as a signed number for the purpose of detecting and indicating overflow. Adding two positive numbers whose sum is greater than or equal to 2^{35} gives a negative result, indicating overflow; but that result, which has a 1 in the sign bit, is the correct answer interpreted as a 36–bit unsigned number in positive form. Similarly, adding two negatives gives a result which is always correct as an unsigned number in negative form.

Zero is represented by a word containing all 0s. Complementing this number produces a word containing all 1s, and adding 1 to that produces all 0s again. Hence, there is only one representation for the number zero and its sign is positive. Since the numbers are symmetrical in magnitude about a single zero representation, all even numbers, both positive and negative, end in 0. All odd numbers end in 1. (A number containing all 1s represents -1.) However, since there are the same number of numbers with each sign and zero has a plus sign, there is one more negative number than there are strictly positive numbers (non–zero numbers with a plus sign). This is the largest negative number and it cannot be produced by negating any positive number. Its octal representation is 400000000000, meaning -2^{35}, i.e., decimal –34,359,738,386. The magnitude of this number is one greater than the largest positive number.

If ones complement were used for negatives a person could read a negative number by attaching significance to the 0s instead of the 1s. In twos complement notation each negative number is one greater than the complement of the positive number of the same magnitude, so a negative number can be read by attaching significance to the rightmost 1 and to the 0s to the left of it. (The negative number of largest magnitude has a 1 in only the sign position.) In a negative integer, 1s may be discarded at the left, just as leading 0s may be dropped in a positive integer. In a negative fraction, 0s may be discarded at the right. So long as only 0s are discarded, the number remains in twos–complement form because it still has a 1 that possesses significance; but if a portion including the rightmost 1 is discarded, the remaining part of the fraction is now a ones–complement number. For example, single–precision multiplication (the `MUL` instruction) produces a double–length product; the programmer must remember that discarding the low–order part of a double–length negative leaves the high–order part in correct twos–complement form only if the low–order part is zero.

The computer does not keep track of a binary point—the programmer must adopt a point convention and shift the magnitude of the result to conform to the convention used. Two common conventions are to regard a number as an integer (binary point at the right) or as a proper fraction (binary point at the left); in these two cases the range of numbers represented by a single word is -2^{35} to $2^{35} - 1$, or -1 to $1 - 2^{-35}$. Since multiplication and division make use of double–length numbers, there are special instructions for performing these operations with integral operands.

The format for double–length fixed–point numbers is just an extension of the single–length format. The magnitude (or its twos–complement) is the 70–bit string in bits 1–35 of the high– and low–order words. Bit 0 of the high–order word is the sign, and bit 0 of the low–order word is made equal to the sign in any result. The range for double–length integers and proper fractions is thus -2^{70} to $2^{70} - 1$ and -1 to $1 - 2^{-70}$. The double–precision instructions actually use quadruple–length numbers for products and dividends. Numbers of any length are just a further extension of the basic format: thirty–five additional bits of the number in each lower–order word, with bit 0 made equal to the sign in results. Remember that truncating a multiple–length negative requires an adjustment for the twos–complement unless the part discarded is zero. The convention for bit 0 of lower–order words is inconsistent with that used for floating–point format (see below). This does not affect the arithmetic instructions themselves, as they ignore bit 0 in all lower–order words. However, the instructions that negate a double–word (e.g., DMOVN) follow the floating–point convention. This means that, if such instructions are used for fixed–point numbers, a problem could arise when comparing one double–precision integer with another.

1.5.2 Floating Point Numbers

The floating–point instructions provide for conversion between fixed and floating forms and handle both single– and double–precision floating–point numbers. The same format is used for a single–precision number and the high–order word of a double–precision number. A floating–point instruction interprets bit 0 as the sign but interprets the rest of the word as an 8–bit exponent and a 27–bit fraction. For a positive number, the sign is 0, as before. However, the contents of bits 9–35 are now interpreted as a binary fraction and the contents of bits 1–8 are interpreted as an integral exponent in excess–128 (decimal, i.e., excess–200_8) code. Exponents from (decimal) -128 to $+127$ are therefore represented by the binary equivalents of 0 to 255 (i.e., 000_8 – 377_8). Floating–point zero is represented by a word containing all 0s. Negative floating–point numbers is represented by the twos–complement of its positive counterpart. A negative number has a 1 for its sign and the twos–complement of the fraction; since every fraction must ordinarily contain a 1 unless the entire number is zero (see below), it has the ones–complement of the exponent code in bits 1–8. Since the exponent is in excess–128 code, an actual exponent x is represented in a positive number by $x + 128$, in a negative number by $127 - x$. The programmer, however, need not be overly concerned with the details of these representations because the hardware compensates automatically. For example, for the instruction that scales the exponent, the hardware interprets the integral scale factor in standard twos–complement form but produces the correct ones complement result for the exponent.

$$
\begin{array}{rcrcl}
+153_{10} & = & +231_8 & = & +0.462_8 \times 2^8 \\
 & & & = & \begin{array}{|c|c|l|}\hline 0 & 10\ 001\ 000 & 100\ 110\ 010\ 000\ 000\ 000\ 000\ 000\ 000 \\ \hline \end{array} \\
 & & & & \scriptstyle 0\ 1\qquad\qquad 8\ 9\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 35 \\
-153_{10} & = & -231_8 & = & -0.462_8 \times 2^8 \\
 & & & = & \begin{array}{|c|c|l|}\hline 1 & 01\ 110\ 111 & 011\ 001\ 110\ 000\ 000\ 000\ 000\ 000\ 000 \\ \hline \end{array} \\
 & & & & \scriptstyle 0\ 1\qquad\qquad 8\ 9\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 35
\end{array}
$$

The floating–point instructions assume that all non–zero operands are normalized. The floating–point instructions normalize a non–zero result. A floating–point number is considered normalized if the magnitude of the fraction is greater than or equal to $\frac{1}{2}$ and less than 1. The hardware may give incorrect or imprecise results if the program supplies an operand that is not normalized or that has a zero fraction with a non–zero exponent.

Single–precision floating–point numbers have a fractional range in magnitude of $\frac{1}{2}$ to $1 - 2^{-27}$, about eight significant decimal digits. Increasing the length of a number to two words does not significantly change the range but rather increases the precision; in any format the magnitude range of the fraction is $\frac{1}{2}$ to 1 decreased by the value of the least–significant bit. In these formats the exponent range is -128 to $+127$, giving a decimal range of approximately 1.5×10^{-39} to 1.7×10^{38}. The G–format floating–point numbers (described below) extend the range of the exponent.

The precaution about truncation given for fixed–point multiplication applies to single–precision floating–point operations because they are done in extra length; but the programmer may request rounding, which automatically restores the high–order part (the result) to twos–complement form if it is negative. In double–precision floating–point instructions, all operands and results are double length, and all instructions calculate an extra length–answer, which is rounded to double length with the appropriate adjustment for a twos–complement negative. In double–precision format the high–order word is the same as a single–precision number, and bits 1–35 of the low–order word are simply an extension of the fraction, which is now sixty–two bits, or over eighteen decimal digits. Bit 0 of the low–order word is made 0 in a result but it is ignored in all operands; e.g., the number $2^{18} + 2^{-18}$ has this two–word representation in double–precision format,

```
|0|10 010 011|100 000 000 000 000 000 000 000 000|
 0 1        8 9                                 35

|0|00 000 000 010 000 000 000 000 000 000 000 000|
 0 1                                            35
```

and its negative is

```
|1|01 101 100|011 111 111 111 111 111 111 111 111|
 0 1        8 9                                 35

|0|11 111 111 110 000 000 000 000 000 000 000 000|
 0 1                                            35
```

1.5.3 G–format Floating–Point Numbers

A collection of instructions to handle extended–range (or "*giant*") floating–point numbers has been included in the KL10, and XKL-1.[3] These instructions include the usual arithmetic operations as well as conversions between G–format floating–point numbers and integers, double word integers, single–precision floating–point, and double–precision floating point. The G–format operands are similar to double–precision floating–point numbers; however, in G–format numbers, the exponent field has been expanded by three bits at the expense of losing bits in the fraction. For this small loss in precision, one decimal digit of significance, the range has been greatly extended.

In G–format, bit 0 of the first word is interpreted as the sign; the next eleven bits are the exponent; twenty–four bits of binary fraction follow in the first word with thirty–five additional fraction bits in the second word, for a total of fifty–nine fraction bits. For positive numbers, the sign is 0; the contents of bits 1–11 are interpreted as an integral exponent in excess–1024 (decimal, i.e., excess–2000_8) code. Exponents from decimal -1024 to $+1023$ are represented by the binary equivalents of 0 to 2047 (0000_8 – 3777_8). Floating–point zero is represented by a double word containing all 0 bits. Negative numbers have the sign bit set to 1, the ones complement of the exponent in bits 1–11, and the twos–complement of the fraction in bits 12–35 of the first word and bits 1–35 of the second word. Bit 0 of the second word is zero in results and ignored in operands.

For example, the number $2^{18} + 2^{-18}$ has this two–word representation in G–format,

---

[3] In XKL-1, these instructions trap to a macro–code simulator.

0	10 000 010 011	100 000 000 000 000 000 000 000
0	1 11	12 35

0	00 000 000 000 010 000 000 000 000 000 000 000
0	1 35

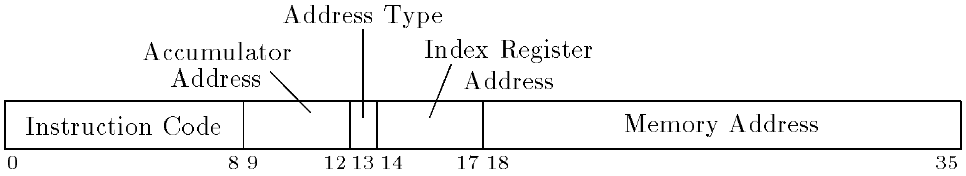
and its negative is

1	01 111 101 100	011 111 111 111 111 111 111 111
0	1 11	12 35

0	11 111 111 111 110 000 000 000 000 000 000 000
0	1 35

These numbers give a decimal range of approximately 2.8×10^{-309} to 9×10^{307}.

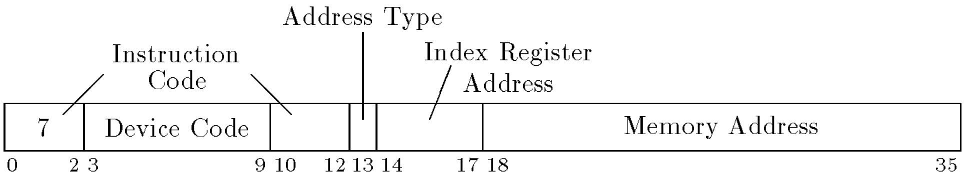
1.6 Instruction Format



Basic Instruction Format

In the basic instruction format, the nine high–order bits (0–8) specify the operation, and bits 9–12 address an accumulator. The rest of the instruction word supplies information for calculating the effective address, which is the actual address used to fetch the operand or alter program flow. Bit 13 specifies the type of addressing, bits 14–17 specify an index register for use in address modification, and the remaining eighteen bits (18–35) address a memory location. In variations on this basic format, bits 9–12 may be used for addressing flags, or all thirteen high order bits (0–12) may be used for an expanded instruction code. The instruction codes that are not assigned as specific instructions are performed by the processor as so–called "unimplemented operations." Among the unimplemented operations are some that are specified as "unimplemented user operations" or UUOs (a mnemonic that means nothing to the assembler). Some of these are for the local use of a program (LUUOs) and some are for communication with the Monitor (MUUOs). In general, unassigned codes act like MUUOs.

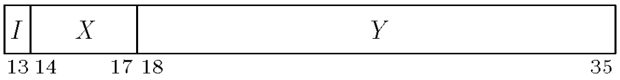
In the KL10 and earlier processors, three 1s in bits 0–2 indicate an input–output instruction; these instructions have a different format, as indicated below. In the IO instruction format used in the KL10 and earlier processors, bits 3–9 address the in–out device to be used in executing the instruction and bits 10–12 specify the operation. The rest of the word is the same as in other instructions.



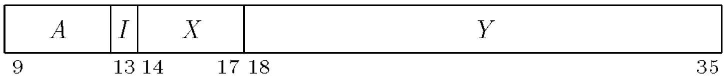
Pre–KS10 In–Out Instruction Format

In all processors from the KS10 on, in–out instructions use the basic instruction format, but for consistency they always do have 1s in the leftmost three bits. (Note there are also non–IO instruction codes beginning with 7.) Post–KL10 IO instruction codes are opportunely chosen so equivalent instructions generally have the same configuration in all processors.

Note that bits 13–35 have the same format in both types of instructions; in fact these bits are the same in every instruction, whether it addresses a memory location or not. In the format illustrations throughout the manual, this part of an instruction word is shown as



where bit 13 is represented by *I* for "indirect bit;" i.e., the address type is either direct or indirect, where the latter is indicated by a 1. For every instruction, the processor carries out an effective address calculation that results in a quantity referred to as *E*. This is the effective address of the instruction if indeed it is an address, whether for an operand or a jump. *E* may, however, represent effective conditions, an effective shift, or something else, but the result of the calculation is always referred to as *E*. In illustrations for the basic instructions, bits 9–35 are almost always represented by

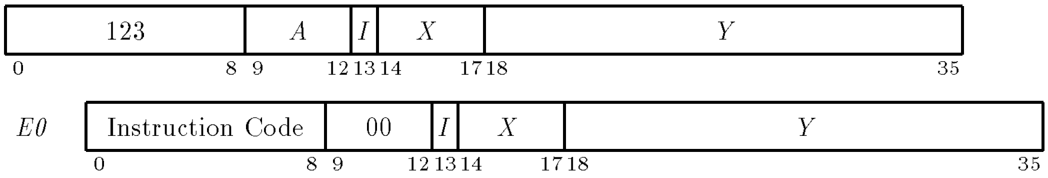


where *A* is the accumulator address.

Note

> Although the various parts of an instruction word are always labeled, in some instructions the result of the effective address calculation is not actually used. Unless otherwise specified, in such cases the I, X, and Y parts of the word are reserved by XKL for possible future use, and they *must be zero* for compatibility with such use. Similarly when bits 9–12 are not used, they are also reserved and must be zero.
>
> A similar stricture holds for all the formats defined throughout the manual for address words, pointers, and miscellaneous special words associated with system features. In words supplied by the program, unassigned bits are available for arbitrary use by the user only if specifically so indicated. Bits labeled "reserved" or simply left blank are reserved to Digital for future use by the hardware or use by the system software. In any word read by the program, unlabeled bits are read as 0s unless there is a specific indication otherwise.

The XKL-1, KL10, and KS10 have a feature that allows expansion of the instruction repertory by an extension of the basic format to two words. In a two–word instruction, it is only the first word that actually appears in the program sequence (i.e., that is referenced by PC), and the accumulator used by the instruction is that specified by the A field of the first word. However, the instruction the processor actually executes is the second word; it is found at location *E0*, which is the result of the effective address calculation for the first word. Moreover, the way the processor interprets the instruction code of the second word is entirely different from the way it would if that same word appeared in the program sequence as a one–word instruction. Thus, use of a single instruction code in the first word effectively creates a whole new instruction set as large as the one the processor already has. At present there is only one such extended instruction set, and only a small number of the available extended codes are used. In extended instructions, the first instruction word is the EXTEND instruction, which has code 123. The format illustrations for these instructions are like this.



Remember, however: although the two words are shown together, they never appear one after the other in the program sequence. If they did, the processor might well perform the second word as a standard instruction after executing it as an extended instruction. As with all instructions, before executing the second word the processor calculates an effective address for it; this is referred to as *E1*, and its use depends on the instruction. Bits 9–12 of the second instruction word must be zero for compatibility with possible future use. Unassigned extended instruction codes are executed as MUUOs.

1.7 Effective–Address Calculation

Note

> The calculation of E, the Effective–Address, is the first step in the execution of every instruction. No other action taken by any instruction, no matter what it is, can possibly precede that calculation. There is absolutely nothing whatsoever that any instruction can do to any accumulator or memory location that can in any way affect its own effective–address calculation.
>
> An effective–address is calculated for every instruction regardless of whether or not the instruction actually references memory.

Effective–address calculation generally is performed in the virtual–address space of the program. This is true even for fast memory, which every program regards as in its virtual space even though fast–memory addresses are treated as unmapped addresses and are not sent to the pager for mapping.

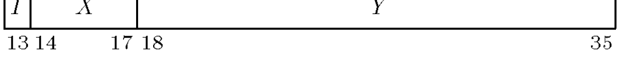
The exceptional cases where effective–address calculations are not done in the virtual–address space of the program occur either when an executive–mode program is performing a PXCT instruction that specifies that the target instruction's effective–address calculation is to be performed in the previous context, or when an executive–mode program is executing with the pager turned off, e.g., at system start up. In the latter case, all addresses used are physical addresses for memory and the program must not give addresses that lie outside the range determined by available memory. When the Monitor is setting up page maps, it must select appropriate physical translations.

1.7.1 Section Zero Effective–Address Calculation

For our discussion of the effective–address calculation, we shall begin with the simpler case—a virtual space limited to section zero (all quantities are eighteen bits). This is the calculation performed by the KA10, KI10, unextended KL10, and KS10 processors. This description applies also to the extended KL10 and the XKL-1, when operating in section zero. This calculation is depicted in Figure 1.9.

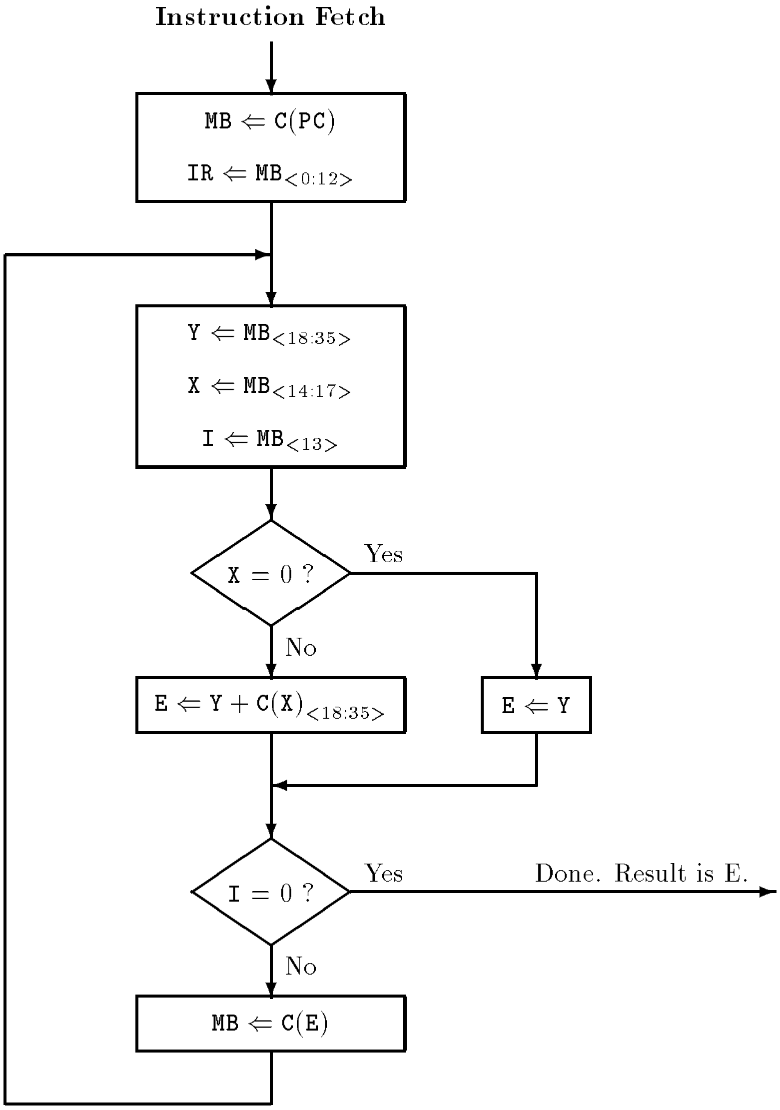
As explained at the beginning of this chapter, the address space of an unextended processor is limited to one section, which by definition is section zero. Such processors employ only in–section addresses, because no section number is necessary when there is only one section.

Bits 13–35 have the same format in *every* instruction whether it addresses a memory location or not. Bit 13 is the indirect bit; bits 14–17 are the index register address; and bits 18–35 are called the address Y.



The effective–address E of the instruction depends on the values of I, X, and Y. If I and X are both zero, Y is E, i.e., bits 18–35 contain the effective address. If X is non–zero, the contents of the right half of index register X are added to Y to produce an 18–bit modified address. If I is 0, addressing

Figure 1.9: Single–Section Effective–Address Computation



is direct and the modified address is the effective address used in the execution of the instruction; if I is 1, addressing is indirect and the processor retrieves another address word (referred to as an "indirect word") from the location specified by the modified address already determined. This new word is processed in exactly the same manner: X and Y determine the effective address if I is 0, otherwise, they are used for yet another level of address retrieval. This process continues until some referenced location is found with a 0 in the indirect bit; the 18–bit number calculated from the X and Y parts of this location is the effective address E.

We have taken Y to be a memory address, but the program can just as well have an address in the index register, and have the Y part of any instruction or indirect word that references it be an offset or displacement. An instruction or indirect word is still an "address word", even though it may not contain an address; the quantity in an index register is still called an "index", even when it is an address instead of an offset.

Note that, throughout the procedure, no computed quantity is ever larger than eighteen bits. In the arithmetic operations, overflows are discarded by disabling the carry from bit 18 to bit 17. Hence adding a large offset can be the same as subtracting a small one.

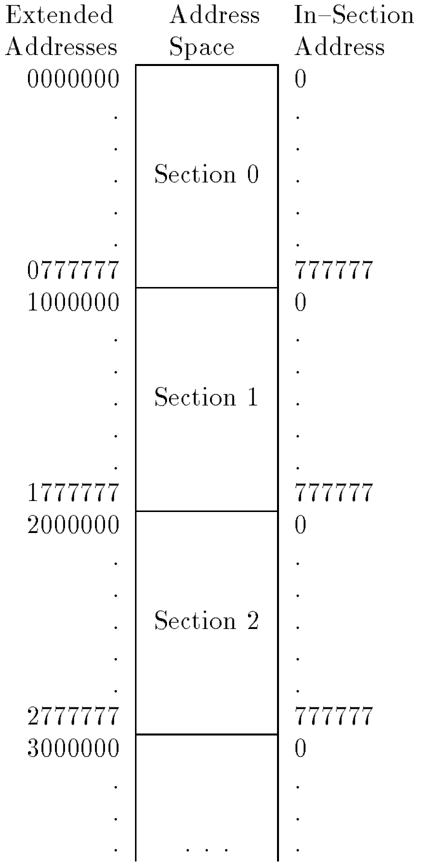
The calculation outlined above is carried out for *every* instruction, even if it need not address a memory location. If the indirect bit in the instruction word is 0 and no memory reference is necessary, then Y is not a memory address. It may be a mask in some kind of test instruction, conditions to be sent to an in–out device, an offset for bytes in a string, or part of it may be the number of places to shift in a shift or rotate instruction or the scale factor in a floating scale instruction. Even when modified by an index register, bits 18–35 do not contain a memory address when I is 0 and no memory reference is required. But when I is 1, the number determined from bits 14–35 is an indirect address no matter what type of information the instruction requires, and the word retrieved in any step of the calculation contains an indirect address so long as I remains 1. When a location is found in which I is 0, bits 18–35 (perhaps modified by an index register) contain the desired effective mask, effective conditions, effective offset, effective shift number, or effective scale factor. Many of the instructions that usually reference memory for an operand have an "immediate" mode in which the result of the effective address calculation is itself used as a half–word operand instead of a word taken from the memory location it addresses. The KS10 IO instructions do not use the result of the effective address calculation; instead, they recompute an IO address by a similar procedure (§2.17).

The important thing for the programmer to remember is that the same calculation is carried out for every instruction regardless of the type of information that must be specified for its execution, or even if the result is ignored. In the discussion of any instruction, E refers to the actual quantity derived from I, X, and Y and used in the execution of the instruction, be it the entire half–word, as in the case of an address, immediate operand, mask, offset, or conditions, or only part of it, as in a shift number or scale factor.

1.7.2 Extended Effective–Address Calculation

In an extended processor the much larger address space is divided into sections of 256K each, and an individual location is identified by an address containing both a section number and an in–section part, as depicted in Figure 1.10. There are still many circumstances, however, in which in–section addresses are used alone in an extended processor. The most obvious case is the address given directly by an instruction: this is limited to eighteen bits and is confined to the section from which

Figure 1.10: Extended Address Space



the instruction is retrieved, being usually the section in which the program is currently running as determined by PC.

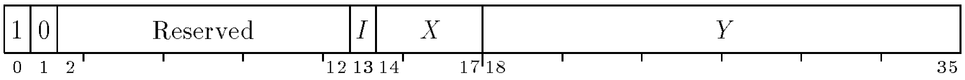
Even in an extended processor, an effective–address calculation performed in section zero is done exactly as outlined above, with all addresses and displacements taken as 18–bit quantities contained in bits 18–35 of an instruction word, an index register, or an indirect word. In other words, when a program is running in section zero, E can never reference a non–zero section for either an operand or a jump (although it can reference an operand that supplies an extended address). Moreover, in terms of addressing, section zero of an extended processor is entirely compatible with the single section of an unextended processor. However, in non–zero sections, the effective–address calculation can use extended addressing. To understand how extended addressing works, the reader must understand the following terms.

- Every address can be represented by 31 bits: one bit to distinguish between a *local address* and a *global address* and 30 bits to represent the address itself. The 30 bits of address are thought of as a 12–bit section number and an 18–bit in–section address.

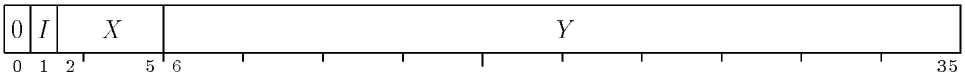
- An *instruction word* is a word addressed by PC (the program counter) and read and interpreted by the processor as an instruction. An instruction word contains an operation code and fields that specify the effective address of an operand.
- An *address word* is any word that is used to supply an address during the effective–address computation. The effective–address computation references a sequence of one or more address words; the first of which is the instruction word. Indirect address words (discussed below) and byte–pointer words are also examples of address words.
- For a *local address*, an explicit computation provides the 18–bit in–section address, but the 12–bit section number is supplied implicitly. The implicit section number is supplied by the section from which the last of the address words was fetched. (When indirect addressing is not used, the instruction word is the last address word, so the implicit section number is the PC section.)
- A *global address* is provided by a computation that supplies all 30 bits explicitly.

  Note that section number 7777 is reserved; a memory reference to section 7777 always traps to the Monitor.

  The extended KL10 implements only the right–most twenty–three bits (sections 0–37), although larger section numbers could be used for software purposes.
- A *local index* is an 18–bit unsigned displacement or address in bits 18–35 of an index register.
- A *global index* is a 30–bit unsigned displacement or address in bits 6–35 of an index register.
- A *local indirect word* is one containing a local address or displacement in this format:

  Because of its similarity to the format of an instruction word, an address word of this sort is also called an “instruction–format indirect word”.
- A *local address word* is a word that contains a local address. A local address word is either a local indirect word or an instruction word.
- A *global indirect word*, also called a *global address word* is one containing a global address or displacement in this format:

  An address word of this type is also called an “extended–format indirect word”

We can now state that an extended effective–address calculation is carried out by essentially the same procedure as described above, with index and indirect steps depending on the values of I and X supplied by a sequence of address words. Now, however, there are differences in the meanings of individual terms and in the way individual operations are performed. First, the indirect bit can be either bit 13 or bit 1, depending on whether it is supplied by a local or global address word (instruction or extended format). Second, there are several varieties of indexing: local and global, with two versions of the latter depending on whether the quantity being indexed is local or global.

- *Local indexing* occurs when the address word is local and either the left half of the index register is negative (i.e., bit 0 is one) or the section number part of it (bits 6–17) is zero. (Index register bits 1–5 are available to software.) In this case the operation is carried out just as in the unextended procedure, and the indexing produces a local address in the section from which the address word is taken (the PC section in the case of an instruction word). Thus, the program can use local indexing in a non–zero section. Furthermore, a program can use the left half of the local index register for a control count that counts up through negative numbers to end an iterative process at zero (as is done in, e.g., the AOBJN instruction).
- *Global indexing* means the interpretation of an index register as a 30–bit address; consequently a 30–bit global address will result. Global indexing can occur in either of two circumstances (with quite different effects). First, global indexing occurs when the address word is global. The 30–bit value in bits 6–35 of the index register is added to the 30–bit Y value from bits 6–35 of the global address word. This is simply a global extension of local indexing: the address word may contain an address and the index register an unsigned offset, or vice versa; adding a large offset can be the same as subtracting a small one. (In this case, bits 0–5 of the index register are available to software.)

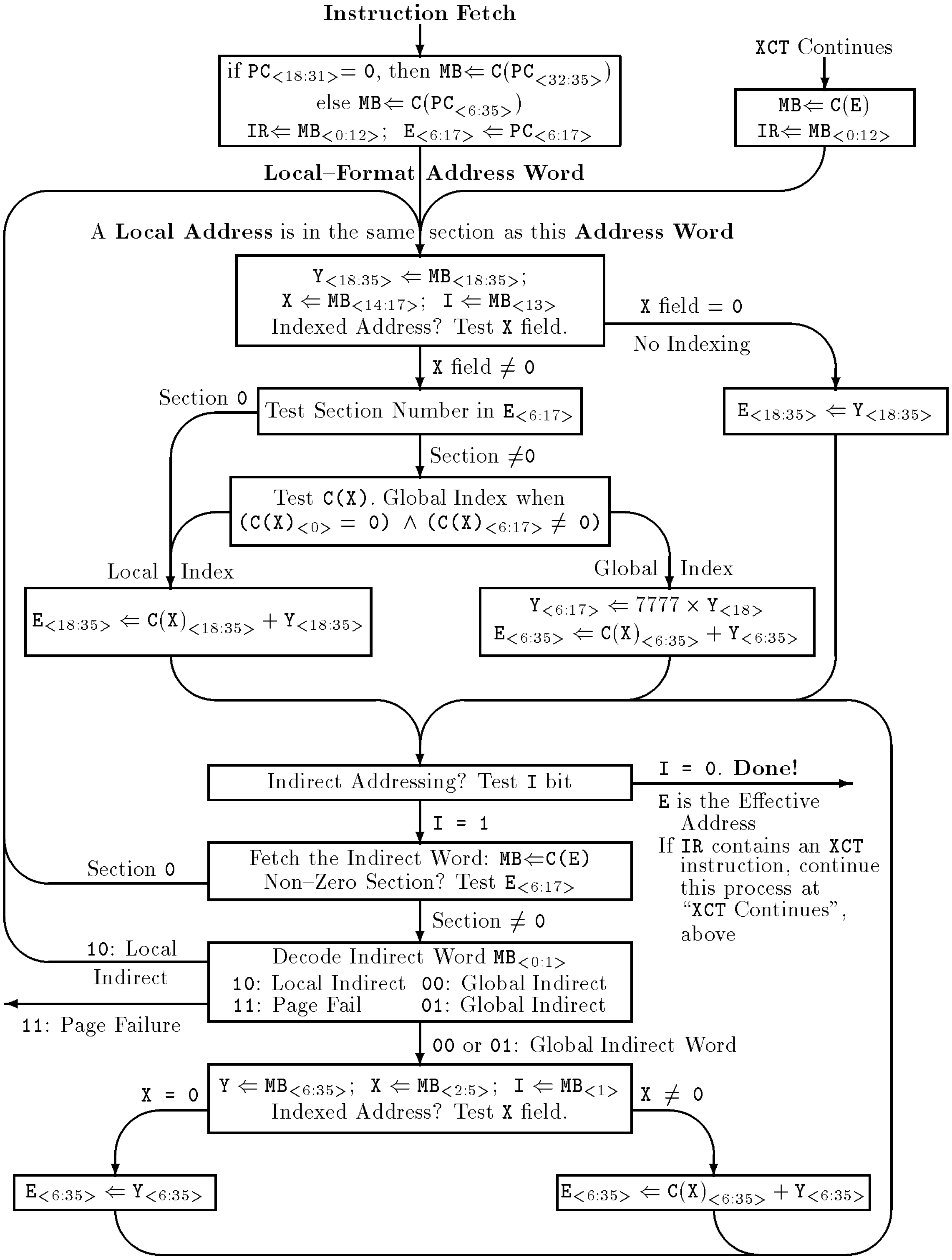
  Second, global indexing occurs when the address word is local and the left half of the index register is positive (i.e., bit 0 is zero) and bits 6–17 contain a non–zero section number. This case is quite unlike local indexing: the index (again, bits 6–35) is assumed to be a global address, and the 18–bit Y is interpreted as a signed displacement (in the range from -2^{17} to $2^{17}-1$), which is added to it algebraically. The value Y is sign–extended by copying its bit 18 to bits 6–17 of the addend. (In this case, bits 1–5 of the index register are available to software.)

As shown in Figure 1.11, the effective–address calculation begins in the section from which the first address word is taken. This is the “local” section for the given address word—the PC section in the case of an instruction word specified by PC. The calculation remains in the local section until the appearance of a global quantity (index or indirect word) changes the section number. So long as only local events occur, all addresses are interpreted as being in the same section (local indexing wraps around 256K). Note that either a local or global address can be used to fetch either a local or global indirect word, but indexing can change only a local quantity to a global one—it cannot modify a global address into a local one. No matter how long the procedure remains local, global indexing or retrieval of a global indirect word can switch to a new section. However if the procedure enters section zero it can never get out. This is because the calculation then interprets all further quantities as local, no matter what their format; i.e., no matter what the program may have meant by the information placed in the words containing them.

At the end, if E is an address, then either it is a global address or it is a local address in the last section from which an address word was fetched. The distinction between a local address and the numerically equivalent global address is sometimes important. For example, in an instruction that uses a two–word operand in E and $E+1$, if E is local then $E+1$ will be in the same section as E; however, if E is global and bits 18–35 of E are 777777, then $E+1$ will be location 0 of the section following E. Also, when bits 18–35 contain a value in the range 0–17 and the address is local, the address specifies one of the accumulators; however, if the address is global (and the section is greater than 1), the address specifies a memory location.

In an instruction in which E is not an address, the section number is ignored and E is whatever number of bits is appropriate. In particular, an immediate–mode operand is always eighteen bits, except in two instructions that specifically handle an extended address as an immediate operand.

Figure 1.11: Extended Effective–Address Computation



The accumulators are regarded as being in the local section of the instruction that addresses them. Hence, unless otherwise specified, a local pointer taken from an accumulator addresses a location in the same section as the instruction.

Finally, there is the matter of fast–memory reference. An address references a fast–memory location if its in–section part is in the range 0–17 and either the address is supplied by PC, the section number is 0, the section number is 1, or the address is local. Note that if PC counts beyond the last in–section address, the wraparound causes instructions to be taken from the ACs. There are two means by which AC references can be made from any section: by using a local address or by using what is specifically regarded as a *global AC address*: a section number of 1 combined with a fast–memory in–section address.

1.8 Programming Conventions

Two elements of system software intimately associated with the presentation in this manual are the assembler and the operating system. The manual explains the DECsystem–10 and DECSYSTEM–20 in terms of machine language programming. Such programming makes use of those basic characteristics of the MACRO assembler described here. The assembler naturally has many other features, such as use of predefined and user–defined pseudo–instructions. The overview of the system presented in the first two sections and the more detailed presentation of system operations in later chapters are in a sense a presentation of the sophisticated features of the operating system: its most impressive features related to the processor are essentially its capabilities for taking advantage of these sophisticated hardware characteristics. There are two operating systems: the TOPS–10 Monitor and the TOPS–20 Monitor. The basic thrust of both is the timesharing of the system among a number of independent users, all of whom can make extensive use of all system facilities, including front–end processing and the advanced file system.

MACRO recognizes a number of mnemonics and other initial symbols that facilitate constructing complete instruction words and organizing them into a program. In particular there are mnemonics for the instruction codes (Appendix A.3), which are nine or thirteen bits (six in pre–KS10 in–out instructions). The assembler translates every statement into a 36–bit word, placing 0s in all bits whose values are unspecified. For example, the mnemonic

```
        MOVNS
```

assembles as 213000 000000, and

```
        MOVNS   2570
```

assembles as 213000 002570. This latter word, when executed as an instruction, produces the twos complement negative of the word in memory location 2570.

Note

> Throughout this manual all numbers representing instruction words, register contents, codes, and addresses are always octal, and any numbers appearing in program examples are octal unless otherwise indicated. On the other hand, the ordinary use of numbers in the text to count steps in an operation or to specify word or byte lengths, bit positions, exponents, etc. employs standard decimal notation.
>
> In the rare instances where hexadecimal notation is appropriate, such numbers are introduced with “0x”.

The initial symbol `@` preceding a memory address places a 1 in bit 13 to produce indirect addressing. The example given above uses direct addressing, but

```
MOVNS @2570
```

assembles as 213020 002570 and produces indirect addressing. Placing the number of an index register (1–17) in parentheses following the memory address causes modification of the address by the contents of the specified register. Hence:

```
MOVNS @2570(12)
```

which assembles as 213032 002570, produces indexing using index register 12, and the processor then uses the modified address to continue the effective–address calculation.

An accumulator address (0–17) precedes the memory address part (if any) and is terminated by a comma. Thus,

```
MOVNS 4,@2570(12)
```

assembles as 213232 002570, which negates the word in location E and stores the result in both E and in accumulator 4. The same procedure may be used to place 1s in bits 9–12 when these are used for something other than addressing an accumulator, but mnemonics are available for this purpose.

The device code in a pre–KS10 In–Out instruction is given in the same manner as an accumulator address (terminated by a comma and preceding the address part), but the number given must be a multiple of 4 and within the octal range 000–774. Mnemonics are customarily used, and they are defined for all standard device codes. To control the priority interrupt system whose code is 004, one may give

```
CONO 4,1302
```

which assembles as 700600 001302, or equivalently

```
        CONO PI,1302
```

The programming examples in this manual use the following notational conventions:

- A colon following a symbol indicates that it is a symbolic location name.

  ```
  A:      ADD 6,5704
  ```

  indicates that the location that contains ADD 6,5704 may be addressed symbolically as A.

- The period represents the current address, e.g.,

  ```
          ADD 5,.+2
  ```

  is equivalent to

  ```
  A:      ADD 5,A+2
  ```

- Square brackets specify the contents of a location, leaving the address of the location implicit but unspecified. For example,

  ```
          ADD 12,[7256004]
  ```

  and

  ```
          ADD 12,A
             .
             .
             .
  A:      7256004
  ```

  are equivalent. The bracketed quantity, which is called a "literal", can be given as the left and right halves separated by a double comma, not only eliminating the need to insert leading zeros for the right half, but allowing use of a minus sign for a negative half word as well. In other words

  ```
          [-246,,135]
  ```

  is equivalent to

```
[777532000135]
```

- A literal can encompass any number of lines of code, employing any of the programming conventions defined above, and be assembled in consecutive locations. In fact a reasonable way to assemble the extended instructions is to give the individual extended instruction code and any necessary follow–up words as a literal in an EXTEND instruction. The assembly of these two lines,

```
STRING: EXTEND  AC,[MOVSO OFF
                   FILL]
```

  produces, in location STRING, an EXTEND instruction whose Y part ($E0$) points to the location containing the second instruction word MOVSO OFF. The Y part ($E1$) of the MOVSO contains the signed offset OFF, and location $E0+1$ contains the fill character FILL.

- Anything written at the right of a semicolon is commentary, not interpreted by the computer, that explains the program.

1.9 KI10 and KA10 Characteristics

The KI10 and KA10 are similar, even identical, to the KL10 in many respects, but their implementation is quite different: they have no microcontroller or microcode. They use the PDP–10 instruction set but not in its full variety as available in the KL10: neither earlier processor can handle strings or double–precision fixed–point numbers; the KA10 has no capability for handling double words or performing double–precision floating–point arithmetic, although it does have instructions (retained on all KL10 and KI10 TOPS–10 systems) for assisting the software in doing double-precision floating–point arithmetic in a special software format.

Figure 1.12 shows the organization of a DECsystem–10 based on either of the earlier processors. The processor handles its peripheral equipment directly over an in–out bus. There is no cache, there is a real–time clock but no meters, and all memory is external. The extra four bits shown on address registers are applicable only to the KI10. Both processors use an 18–bit internal address providing a virtual memory of one section that is compatible with section zero of the KL10. However, whereas the KA10 has a maximum physical memory equal in size to its virtual memory, which is organized by protection and relocation hardware, the KI10 has a physical addressing capability equal to that of the KL10 (22–bit address, 4096K) and has paging hardware. The KI10 virtual address space is the same as that of a KL10 with the TOPS–10 Monitor, except that, in executive mode, the first 112K of memory is unpaged (and thus not available to the supervisor program), and the Monitor can define a so–called “small user” whose accessible space must lie within the virtual ranges 0–37777 and 400000–437777. The KI10 has four fast–memory blocks, of which hardware requires that the Monitor use block 0; the KA10 has only one block.

Both processors have manual operator consoles with facilities that are directly relevant to the programmer, although they are used mostly for manually stepping through a program to debug it. From the sense switches and the 36–bit data switch register DS, the program can read information supplied by the operator; and through the memory indicators MI, the program can display data for the operator. By means of the address switch register AS, the operator can examine the contents of, or deposit information into, any memory location; stop or interrupt the program whenever a particular location is referenced; and supply a starting address for the program. In these processors, IR contains the entire left half of the current instruction word; i.e., eighteen bits rather than thirteen. The memory address register MA supplies the address for every memory access. In the arithmetic logic of the KA10, there are only single–length registers; but in the KI10, AR and AD have 28–bit left extensions for double–precision floating–point. The KA10 has no trapping mechanism: arithmetic and stack overflow signal the program by way of interrupts. Individual processor differences relevant to user programming are listed in Appendix C.

1.9.1 Memory

The following table gives the characteristics of the various memories available with the KI10 and KA10. Modify completion is the time to finish a read–modify–write cycle after the processor supplies the new data. Times are in microseconds and include the delay introduced by ten feet (three meters) of cable. Fast–memory times are for referencing as a memory location (18–bit address); when a fast–memory location is addressed as an accumulator or index register, the access time is considerably shorter.

	Read Access	*Write Access*	*Cycle*	*Modify Completion*	*Size*
MA10 Core Memory	.61	.20	1.00	.57	16K
MB10 Core Memory	.60*	.20*	1.65*	.97	16K
MD10 Core Memory	.83	.33	1.8	1.23	32–128K
ME10 Core Memory	.61	.20	1.00	.65	16K
MF10 Core Memory	.61	.20	1.00	.63	32K, 64K
MG10 Core Memory	.67	.23	1.00	.63	32–128K
MH10 Core Memory	.74	.23	1.18	.68	64–256K
KA10 Fast Memory	.21	.21			16
KI10 Fast Memory	.28	.0			16 [×4 sets]

---

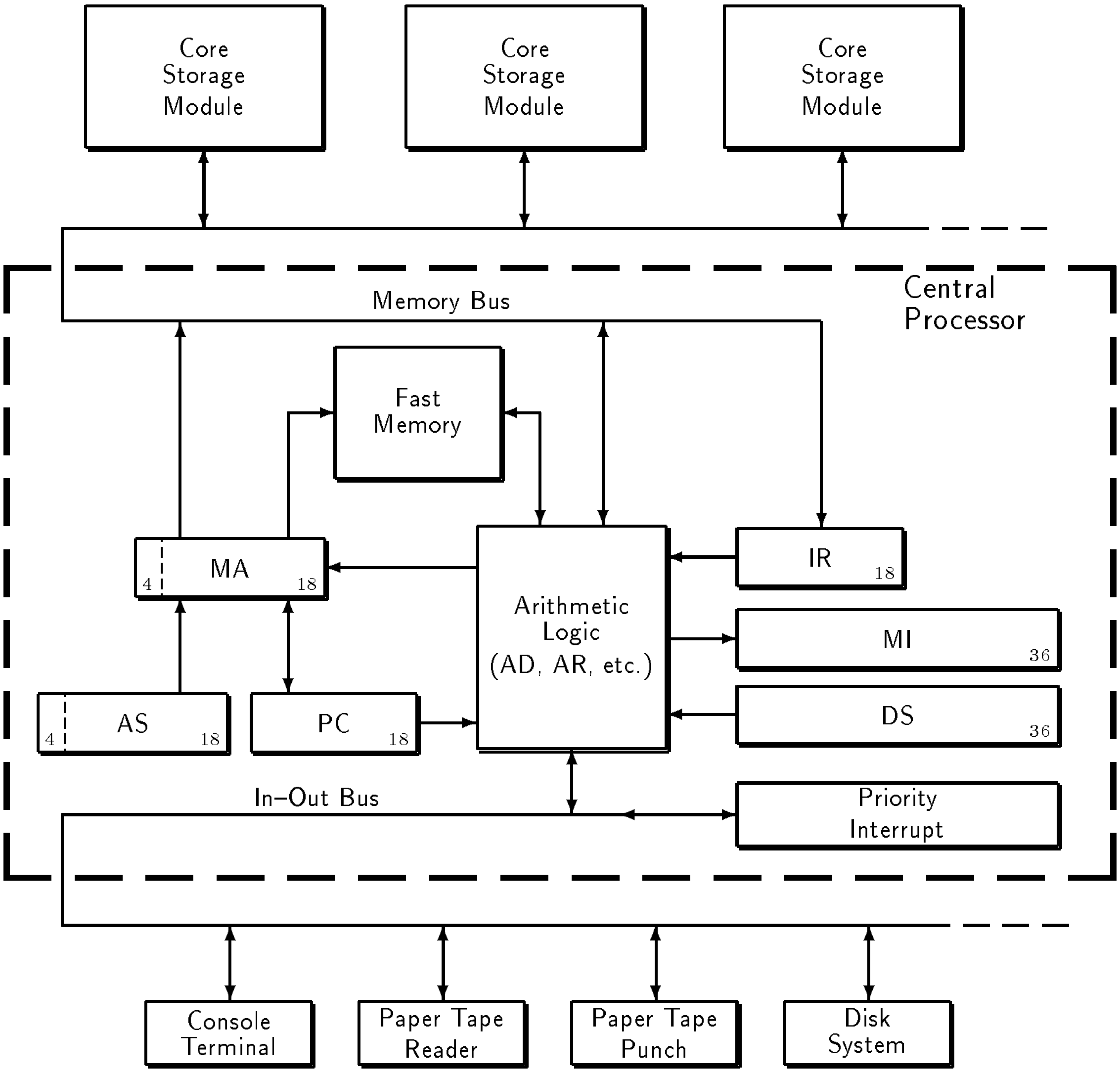
* Add .1 in a multiprocessor system.

KI10 access to accumulators and index registers effectively takes no time—it is done in parallel with other required instruction operations. Retrieval of instructions or memory operands from fast memory is generally not worthwhile because of the extensive overlapping that speeds up core access. However, except in instructions that use two accumulators, storage of a memory operand in fast memory not only takes no time but actually slightly decreases the non–memory time.

In a system with the greatest possible capacity, the largest KI10 address is octal 17777777, decimal 4,194,303; the largest KA10 address is octal 777777, decimal 262,143. All storage modules can be interleaved in pairs, and some of them in sets of four (see Appendix G.3). The KA10 cannot overlap memory access.

KI10 Memory Allocation. The KI10 hardware defines the use of certain memory locations, but most are relative to pages whose physical location is specified by the Monitor. The auto restart uses location 70. The only other physical locations uniquely defined by the hardware are those

Figure 1.12: DECsystem–10 Based on KI10 or KA10



HRMF-KI10SYS10.TEX

in fast memory, whose addresses are the same for all programs: location 0 holds a pointer word during a bootstrap read–in, 0–17 can be addressed as accumulators, and 1–17 can be addressed as index registers. The only addresses uniquely specified in the user virtual space are for user local UUOs—locations 40 and 41. All other addresses defined by the hardware, for use in page mapping, responding to priority interrupts, or other hardware–oriented situations, are to locations in the process tables.

KA10 Memory Allocation. The use of certain memory locations is defined by the KA10 hardware.

0	Holds a pointer word during a bootstrap read–in.
0–17	Can be addressed as accumulators.
1–17	Can be addressed as index registers.
40–41	Trap for unimplemented user operations (UUOs).
42–57	Priority interrupt locations.
60–61	Trap for remaining unimplemented operations: these include the unassigned instruction codes that are reserved for future use, and also the byte manipulation and floating point instructions when the hardware for them is not installed.
140–161	Allocated to second processor if connected (same use as 40–61 for first processor). All information given in this manual about memory locations 40–61 for a KA10 applies instead to locations 140–161 for programming a second KA10 connected to the same memory.

In a user program, the trap for a local UUO is relocated to locations 40 and 41 of the user area; a Monitor UUO uses unrelocated locations. All other addresses listed are for physical (unrelocated) locations.

Chapter 2

User Operations

This chapter describes all PDP–10 instructions that are generally available to the user. It also defines the types of In–Out instructions but does not discuss their effects when they address specific internal system elements or peripheral devices. In the description of each instruction, the mnemonic and name are at the top and the format is in a box below them. The mnemonic assembles to the word in the box, where bits in those parts of the word represented by letters assemble as 0s. The letters indicate portions that must be added to the mnemonic to produce a complete instruction word. For extended instructions, the mnemonic given actually assembles to the word shown in the second format box; the first box shows the configuration of the EXTEND itself. The programmer must write the EXTEND and arrange that its effective–address contains the listed mnemonic; most often this is accomplished by writing the mnemonic in a literal.

For many of the non–IO instructions, a description applies not to a unique instruction with a single code in bits 0–8, but rather to an instruction set defined as a basic instruction that can be executed in a number of modes. These modes define properties subsidiary to the basic operation; e.g., in data transmission the mode specifies which of the locations addressed by the instruction is the source and which the destination of the data; in test instructions it specifies the condition that must be satisfied for a jump or skip to take place. The mnemonic given at the top is for the basic mode; mnemonics for the other forms of the instruction are produced by appending letters directly to the basic mnemonic. Letters representing modes are suffixes which produce new mnemonics that are recognized as distinct symbols by the assembler. Following the description is a table giving the mnemonics and octal codes (bits 0–8) for the various modes.

Most of the non–IO instructions can address an accumulator, and in the box showing the format this address is represented by A; in the description, “AC” refers to the accumulator addressed by A. “AC left” and “AC right” refer to the two halves of AC; sometimes these are written as $\mathrm{AC_L}$ and $\mathrm{AC_R}$, respectively. If an instruction uses two or more accumulators, these have addresses A, A+1, A+2, etc., which are interpreted modulo 20_8; e.g., A+1 is 0 when A is 17. A pair of accumulators holding a double word is referred to as AC,AC+1. In the text, the various accumulators are referred to as AC, AC+1, and so forth. In some cases an instruction uses an accumulator only if A is non–zero: in such cases a zero address in bits 9–12 specifies no accumulator.

In a description, E refers to the effective–address, half–word operand, mask, offset, conditions, shift number, or scale factor calculated from the I, X, and Y parts of the instruction word. In an instruction that ordinarily references memory, a reference to E as the source of information means that the instruction retrieves the word contained in location E; as a destination it means the instruction stores a word in location E. In the immediate mode of these instructions, the effective half–word operand is usually treated as a full word that contains E in one half and 0 in the other, and is represented either as $0, E$ or $E, 0$ depending upon whether E is in the right– or left–half. In extended instructions, *E0* and *E1* refer to the results of the effective–address calculations for the first and second instruction words. E_{R} refers to the right eighteen bits of the effective–address (i.e., the in–section part), but, in a machine lacking extended addressing, E_{R} is equivalent to E.

A reference to "location $E, E+1$" means the contents of the two locations are used together as a double word, such as a double–length number. If the program is running in section zero or the instruction gives a local address, the addresses wrap around so that, when E is 777777, $E+1$ is 0; if the program is running in a non–zero section and the instruction gives a global address in which E_{R} contains 777777, $E+1$ advances to address 0 in the next section. This extends in analogous fashion to instructions with three– and four–word operands. (In contrast to addressing consecutive accumulators with the A field of an instruction, when E is 17, $E+1$ is 20.)

Please Read This

> The calculation of E is the first step in the execution of every instruction. No other action taken by any instruction, no matter what it is, can possibly precede that calculation. There is absolutely nothing whatsoever that any instruction can do to any accumulator or memory location that can in any way affect its own effective–address calculation.

The instructions are described in terms of their overt effects as seen by the user in a normal program situation and on the assumption that nothing is amiss—the program is not attempting to reference a memory that does not exist or to write in a protected area of memory. In general, all descriptions apply equally well to operation in executive mode. For completeness, the effects of restrictions on certain instructions are noted, as are the effects of executing instructions in special circumstances. However, the reader must look elsewhere for the details of programming in such special situations. In particular, §2.9.6 discusses trapping, §2.19 explains the restrictions on user programming, and chapters 3 and 4 describe the special effects and restrictions associated with system operations in the various processors.

Implicit in the execution of an instruction are side effects not overtly visible to the user. Side effects, which vary with different processors, include changes to the system's internal state that result from the normal activities of reading and writing memory. For example, in some processors the cache memory, the pager translation buffer, and the page tables are part of the system's state (the processor and the operating system), not visible to the user, which change with the user program's references to memory. These side effects are generally in the province of programmers who write the operating systems; chapters 3 and 4 describe the special effects of instructions in the various processors.

To minimize processor execution time, the programmer should minimize the number of memory references and iterative operations. When there is a choice of actions to be taken on the basis of some test, the conditions tested should be set up so that the action which results most often takes the least time. There are also various subtleties that affect timing (such as the nature of the arithmetic algorithms), but these are generally not worth considering except in very special circumstances (to determine the effect often takes longer than the time saved).

Execution times are not given with the instruction descriptions, because the time may vary greatly depending upon circumstances. The time depends upon which processor performs the instruction, on the configuration of the operands, and on the number of iterative steps. The processor is designed to save time wherever possible by inspecting the operands in order to skip unnecessary steps.

The text sometimes refers to an instruction as being "executed." To "execute" an instruction means that the processor performs the instruction out of the normal sequence; i.e., the sequence defined by the program counter (this sequence may not be consecutive, as when a skip or jump or some special circumstance changes PC). The processor fetches an executed instruction from a location whose address is supplied not by PC but rather by an extend or execute instruction (whose operand is itself interpreted as an instruction) or by some feature of the hardware such as a priority interrupt, trap, etc. It is assumed that control will shortly be returned to PC at the location it originally specified before the interruption, unless the executed instruction or the hardware feature itself changes PC.

Instruction codes that are not implemented and instructions that are not legal in user mode are said to "trap" as "unassigned codes" or as "Monitor UUOs" (MUUOs). Such an instruction causes a transfer of control to executive mode, as described in §2.16.

Some simple examples are included with the instruction descriptions, but more complex examples using a variety of instructions are given in §2.15

2.1 Full–Word Data Transmission

These are the instructions whose basic purpose is to move one or more full words of data from one place to another, usually from an accumulator to a memory location or vice versa. In a few cases instructions may perform minor arithmetic operations, such as forming the negative or the magnitude of the word being processed.

2.1.1 Exchange Instruction

The presentation of the instruction set begins with a single instruction that simply interchanges the contents of an accumulator and a memory location.

EXCH **Exchange**

250	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Move the contents of location E to AC, and move AC to location E.

2.1.2 Move Instruction Class

This class of instructions consists of a group for general manipulation of single words and a special immediate mode instruction for handling an extended address. Each of the instructions in the standard move group handles one word, which may be changed in the process (e.g., its two halves may be swapped). There are four instructions, each with four modes that determine the source and destination of the word moved.

Mode	*Suffix*	*Source*	*Destination*
Basic		E	AC
Immediate	I	The word $0, E$	AC
Memory	M	AC	E
Self	S	E	E, but also AC if A is non–zero

MOVE Move

200	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move one word from the source to the destination specified by M. The source is unaffected, the original contents of the destination are lost.

`MOVE`	Move	200
`MOVEI`	Move Immediate	201
`MOVEM`	Move to Memory	202
`MOVES`	Move to Self	203

Notes: MOVEI loads the word 0,E into AC. If A is 0, MOVES is a no–op in the sense that it has no overt effect on the contents of memory or the accumulators; however, MOVES both reads and writes in memory, with all attendant side effects. If A is non–zero, MOVES has the same overt effect as MOVE (it loads AC from memory), but it also writes in memory.

MOVS Move Swapped

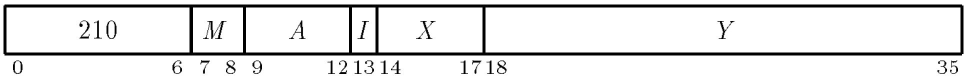
204	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interchange the left and right halves of the word from the source specified by M and move it to the specified destination. The source is unaffected; the original contents of the destination are lost.

`MOVS`	Move Swapped	204
`MOVSI`	Move Swapped Immediate	205
`MOVSM`	Move Swapped to Memory	206
`MOVSS`	Move Swapped to Self	207

Note: Swapping halves in immediate mode loads the word $E, 0$ into AC.

MOVN Move Negative

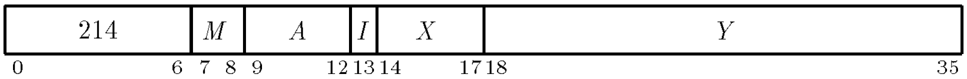


Negate the word from the source specified by M and move it to the specified destination. If the source word is fixed–point -2^{35} (400000 000000) set the Trap 1, Overflow, and Carry 1 flags. (Negating the equivalent floating–point number -1×2^{127} also sets the flags, but this is not a normalized number.) If the source word is zero, set Carry 0 and Carry 1. The source is unaffected; the original contents of the destination are lost.

`MOVN`	Move Negative	210
`MOVNI`	Move Negative Immediate	211
`MOVNM`	Move Negative to Memory	212
`MOVNS`	Move Negative to Self	213

Note: MOVNI loads AC with the negative of the word $0, E$ and cannot overflow.

MOVM Move Magnitude



Take the magnitude of the word contained in the source specified by M and move it to the specified destination. If the source word is fixed–point -2^{35} (400000 000000) set the Trap 1, Overflow, and Carry 1 flags. (Taking the magnitude of the equivalent floating–point number -1×2^{127} also sets the flags, but this is not a normalized number.) The source is unaffected; the original contents of the destination are lost.

`MOVM`	Move Magnitude	214
`MOVMI`	Move Magnitude Immediate	215
`MOVMM`	Move Magnitude to Memory	216
`MOVMS`	Move Magnitude to Self	217

Notes: The word $0, E$ is equivalent to its magnitude, so MOVMI is equivalent to MOVEI.

It is often convenient to keep a control count in the left half of an accumulator and a local address or displacement to be used for indexing in the right half. Suppose one wishes to load 200 into the left half and 1400 into the right half of an accumulator that is addressed symbolically as XR. If the number 200 001400 is stored in location M, one can do this by giving the instruction

```
        MOVE    XR,M
```

Of course, somewhere the source program must define the value of the symbol XR as an octal number between 1 and 17. If the same word, or the negative of the same word, or the same word with its halves swapped must be loaded on several occasions, each transfer still requires only a single move instruction that references M.

2.1.3 Extended Move Immediate

The following instruction makes the result of an effective–address calculation available for use as a global address. If the address specifies a fast–memory location, the instruction loads the global address of that fast–memory location, so that it can be accessed from any section.

XMOVEI Extended Move Immediate

415	A	I	X	Y
0 8	9 12	13	14 17	18 35

If the program is running in a non–zero section, do one or the other of the following.

> If E is not a local AC address, clear AC bits 0–5 and place the global effective–address E in AC bits 6–35.
>
> If E is a local AC address, put 1 in $\mathrm{AC_L}$ and E_{R} in $\mathrm{AC_R}$. (This result is the global form of a fast–memory address.)

If the program is running in section zero, this instruction is called SETMI, a Boolean instruction that performs an analogous function for section zero (§2.4).

Notes. This instruction changes a local AC address to a global AC address, which therefore still refers to fast–memory no matter what section that address is used in. Giving XMOVEI with an address 20 or greater without indexing or indirection places the current PC section number in AC left; this result can be used to determine in what section the program is running.

2.1.4 Double Move Instructions[1]

These four instructions are principally for manipulating the double–length operands used in double–precision arithmetic, fixed or floating. But they may be used to move or negate any double word, i.e., the contents of a pair of adjacent accumulators or memory locations. Two of the instructions are simple extensions of MOVE and MOVEM to double words, and for them the configuration of the operands is irrelevant. The other two instructions are extensions of MOVN and MOVNM, with the operand interpreted as a double–precision floating–point number. With a slight variation in the format, they can also be used for fixed–point numbers: a negative result has a 0 in bit 0 of the low–order word instead of a copy of the sign bit. For arithmetic operations, this difference is inconsequential, because all arithmetic instructions ignore bit 0 of all low–order words. However, this difference in format could cause a comparison of two double–precision fixed–point numbers to fail.

All of these instructions address a pair of adjacent accumulators and a pair of adjacent memory locations. The accumulators have addresses A and $A + 1$ (mod 20_8). The memory locations have

---

[1] In the KA10 these instructions trap as unassigned codes (§2.16).

addresses E and $E + 1$.[2]

DMOVE Double Move

120	A	I	X	Y
0 8	9 12	13	14 17	18 35

Move a double word from location $E, E + 1$ to AC,AC+1. The memory locations are unaffected; the original contents of the two affected accumulators are lost.

DMOVEM Double Move to Memory

124	A	I	X	Y
0 8	9 12	13	14 17	18 35

Move a double word from AC,AC+1 to location $E, E + 1$. The ACs are unaffected; the original contents of the memory locations are lost.

Notes: Do not use the instruction DMOVEM AC,AC+1; its result is indeterminate. In the KI10, do not have E and X address the same (fast) memory location, because a page–failure on the second word would result in a different effective–address calculation when the instruction is restarted.

DMOVN Double Move Negative

121	A	I	X	Y
0 8	9 12	13	14 17	18 35

Interpret the double word from location $E, E + 1$ as double–precision floating–point, and move its negative to AC,AC+1. If the memory double–word is fixed–point -2^{70}, set the Trap 1, Overflow, and Carry 1 flags. (Negating the equivalent floating–point number, -1×2^{127}, also sets the flags, but this is not a normalized number.) If the memory double–word is zero, set Carry 0 and Carry 1. The memory locations are unaffected; the original contents of the ACs are lost.

The negation is done using floating point conventions; hence, a negative fixed–point result has the incorrect value in bit 0 of the low–order word.

In the KI10 there is no overflow test because the KI10 lacks double–precision fixed–point instructions. For floating–point the overflow test is unnecessary, because negating a correctly formatted floating–point number cannot cause overflow.

DMOVNM Double Move Negative to Memory

125	A	I	X	Y
0 8	9 12	13	14 17	18 35

Interpret the double word from AC,AC+1 as double–precision floating–point and move its negative to location $E, E+1$. If the AC double–word is fixed point -2^{70}, set the Trap 1, Overflow, and Carry

---

[2] Refer to the description of $E, E + 1$ on page 52.

1 flags. (Negating the equivalent floating–point number, -1×2^{127}, also sets the flags, but this is not a normalized number.) If the AC double–word is zero, set Carry 0 and Carry 1. The ACs are unaffected; the original contents of the memory locations are lost.

The negation is done using floating–point conventions; hence a negative fixed–point result has the incorrect value in bit 0 of the low order word.

In the KI10 there is no overflow test because the KI10 lacks double–precision fixed–point instructions. For floating–point the overflow test is unnecessary, because negating a correctly formatted floating–point number cannot cause overflow.

Notes. Do not use the instruction DMOVNM AC,AC+1; its result is indeterminate. In the KI10, do not have E and X address the same (fast) memory location, because a page–failure on the second word would then result in a different effective–address calculation when the instruction is restarted.

2.1.5 Block Transfers

There are two instructions for moving blocks of data from one part of memory to another. One is restricted to acting within the section specified by the effective–address. The other can move data arbitrarily anywhere in memory.

BLT Block Transfer

251	A	I	X	Y
0 8	9 12	13	14 17	18 35

Beginning at the location addressed by AC_L (AC left) in the section specified by E, move words to another area in the same section beginning at the location addressed by AC_R (AC right). Continue until a word is moved to location E.[3]

The total number of words in the block is thus $E_R - AC_R + 1$. If $AC_R \geq E$, the BLT moves only one word to location AC_R. If the source block is larger than $2^{18} - AC_L$, it is wrapped around to the beginning of the section.[4]

In the XKL-1, KL10, and KS10, provided AC is not in the destination block, at the end of the instruction AC_L and AC_R contain addresses 1 greater than those of the final source and destination locations referenced, respectively.[5] In the KI10 and KA10, at the end of the instruction, AC is indeterminate unless the interrupt system and the pager are both off, in which case AC is unaffected. In any event, for program compatibility among processors, use of the resulting quantity in AC is strongly discouraged.

---

[3] The source and destination addresses are either local addresses or global addresses, corresponding to whether E is local or global. The distinction between E local or global matters only in the situation where the source and/or destination address is in the accumulators: the accumulators can be addressed only by local addresses in the range 0–17 (or by global addresses 1000000–1000017).

[4] *Caution*: In the extended KL10, wraparound is not implemented correctly: the instruction inadvertently reads source words from the next higher section. However, if the instruction is interrupted after it has counted into the next section, when it resumes, it will revert to reading data from the original section.

[5] In the KL10, if the BLT is abbreviated because the initial value of $AC_R > E$, at the end of the instruction AC_L and AC_R contain values that incorrectly indicate that the BLT moved additional words.

Caution

> Should an interrupt or page failure occur during its execution, the BLT stores the source and destination addresses for the next word in AC, so when the processor restarts upon the return to the interrupted program, it actually resumes at the correct point within the BLT. Therefore, A and X must not address the same register because this would produce a different effective–address calculation upon resumption; and the instruction must not attempt to load an accumulator addressed either by A or X unless it is the final location being loaded.

Examples

A convenient way to clear a block of consecutive locations in memory is to clear the first location and then use a BLT to transfer the zero successively from one location to the next. Suppose the block starts at A and contains B words.

```
MOVE  AC,[A,,A+1]
SETZM A
BLT   AC,A+B-1
```

This technique can be used to spread any one–word pattern through consecutive locations.[6] An n–word pattern can be spread through memory by initializing the right half of the accumulator to be n larger than the left half.

The following instructions load the accumulators from memory locations 2000–2017 in the PC section.

```
MOVSI   17,2000         ;Put two addresses, 2000,,0 in AC 17
BLT     17,17           ;load ACs from 2000-2017
```

As mentioned in the above caution, this example would not work reliably if, for example, AC 10 or AC 16 were used to supply the source and destination addresses. The example is written safely: AC 17 is the last location loaded by the BLT.

To store the accumulators in memory requires that one accumulator first be made available to the BLT:

```
MOVEM   17,2017       ;Move AC 17 to 2017 in memory
MOVEI   17,2000       ;Put two addresses,  0,,2000 in AC 17
BLT     17,2016       ;store ACs 0-16 into addresses 2000-2016
```

To give a more complex example, the following code fragment stores accumulators 0–16 on the stack

---

[6] This function is used so frequently that the KL10 microcode detects it as a special case and reads only the first source word.

(see §2.10) described by accumulator 17, presuming that the stack has room for the new entries. This code works properly in section zero. It also works in non–zero sections, provided that accumulator 17 contains a local–format stack pointer.

```
ADJSP   17,17          ;allocate stack space for 0-16
MOVEM   16,0(17)       ;store 16, AC for the BLT
MOVEI   16,-16(17)     ;load 0,,in-section address of stack
BLT     16,-1(17)      ;copy ACs 0-15 to stack
```

The following restores accumulators that have been saved on the stack by the fragment shown above:

```
MOVSI   16,-16(17)     ;in-section address of the stack,,0
BLT     16,16          ;restore accumulators from stack
ADJSP   17,-17         ;return stack space no longer needed
```

In the examples above, BLT has been used to store the accumulators in the local section (i.e., the PC section). To load or store the accumulators in a non–local section, the following subtle adaptation can be used.[7] This code fragment depends on a characteristic of the XCT instruction: it will perform the effective–address calculation of the target instruction in the section that contains the target instruction (see §2.9.1). Thus, a local effective–address is computed in a section other than the PC section.

```
ADJSP   17,17              ;allocate stack space for 0-16
DMOVEM  15,-1(17)          ;store 15 and 16, BLT AC and Eff Addr
MOVEI   16,-16(17)         ;load 0,,in-section address on stack
MOVEI   15,-2(17)          ;0,,in-section final address for ACs on stack
PUSH    17,[BLT 16,(15)]   ;instruction to XCT, in stack's section
XCT     (17)               ;XCT the BLT.  BLT uses local addressing
ADJSP   17,-1              ;deallocate stack space for BLT instruction
```

The restore is accomplished with somewhat less fuss:

```
MOVSI   16,-16(17)         ;in-section first source address,,0
PUSH    17,[BLT 16,16]     ;instruction to XCT, in stack's section
XCT     (17)               ;XCT the BLT.  BLT uses local addressing
ADJSP   17,-20             ;deallocate space for BLT and ACs
```

For a reverse BLT procedure (highest addresses first), refer to the POP instruction (§2.10) or to the XBLT instruction below.

---

[7] This code can be run in any section, regardless of whether the stack pointer is global or local. Note: However, the region of the stack in which the accumulators are being stored must not cross a section boundary.

XBLT Block Transfer

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	020	00	*I*	*X*	*Y*	*E1* is not used.[8]
	0 8	9 12	13	14 17	18 35	

Move a block of words from one area of memory to another. The block size and the locations of the source and destination areas are defined by the contents of a block of three accumulators.

AC	Number of Words in Block	
AC+1	00	Location of Source Block
AC+2	00	Location of Destination Block
	0 5	6 35

Perform a forward or backward block transfer[9] as follows.

> If AC contains a positive number N, move a block of N words from a source area beginning at the location specified by AC+1 to a destination area beginning at the location specified by AC+2 and extending through increasing addresses. At the end, AC is clear, and AC+1 and AC+2 respectively contain addresses 1 greater than those of the final source and destination locations referenced.
>
> If AC contains a negative number $-N$, move a block of N words from a source area beginning at a location 1 less than that specified by AC+1 to a destination area beginning at a location 1 less than that specified by AC+2 and extending through decreasing addresses. At the end, AC is clear, and AC+1 and AC+2 respectively contain the addresses of the final source and destination locations referenced.

Notes: The contents of AC+1 and AC+2 are interpreted as 30–bit global addresses. This instruction is legal in section zero, and it can reference addresses in non–zero sections when executed in section zero.

Caution

> This instruction uses three accumulators, and under no circumstances should any of these three be part of either the source or destination block. Because of the possibility of an interrupt or page failure, the contents of these accumulators, even as a source, cannot be guaranteed. In any event, a BLT can store (or load) the accumulators to (or from) any section.

---

[8] I, X, and Y are reserved and should be zero.

[9] As of KL10 microcode 2.1[442], there is a problem when XBLT is executed by PXCT: the optimization of reading only the first source word when the destination address is precisely 1 larger than the source address in a forward transfer is mistakenly applied when the source and destination addresses are in different contexts.

2.2 Fixed–Point Arithmetic

For fixed–point arithmetic the PDP–10 has instructions for performing addition, subtraction, multiplication, and division of numbers in single– and double–precision fixed–point format (§1.5.1), although double–precision is not available in the KI10 or KA10. The processor can also do arithmetic shifting—which is essentially multiplication by a power of 2—but those instructions are discussed with logical shifting and rotating (§2.5). For single–precision, the add and subtract instructions involve only single–length numbers, whereas multiply supplies a double–length product and divide uses a double–length dividend. There are also integer multiply and divide instructions that involve only single–length numbers and are especially suited for handling smaller integers, particularly those of eighteen bits or less such as addresses, bytes, and character codes. For double–precision, the add and subtract instructions involve only double–length numbers, whereas multiply supplies a quadruple–length product and divide uses a quadruple–length dividend. In all cases, the position of the binary point is arbitrary; the programmer may adopt any point convention. Even the integer multiply and divide instructions can be used for small fractions, provided the programmer keeps track of the binary point. For convenience in the following discussion, all operands are assumed to be integers (binary point at the right).

The processor has four flags, Overflow, Carry 0, Carry 1, and No Divide, that indicate when the magnitude of a number is or would be larger than can be accommodated. Carry 0 and Carry 1 detect carries out of bits 0 and 1 in certain instructions that employ fixed–point arithmetic operations: the add and subtract instructions treated here, the move instructions that produce the negative or magnitude of the word moved (§2.1), and the arithmetic test instructions that increment or decrement the test word (§2.6). In these instructions an incorrect result is indicated—and the Overflow flag set—if the carries are different; i.e., if there is a carry into the sign but not out of it or vice versa. Overflow is determined directly from the carries, not from the carry flags, because their states may reflect events in previous instructions. The Overflow flag is also set by No Divide being set, which means the processor has failed to perform a division because the magnitude of the dividend is greater than or equal to that of the divisor or, in integer divide, simply that the divisor is zero. In other overflow cases, only Overflow itself is set: these include too large a product in multiplication, too large a number to convert to fixed point (§2.3), and loss of significant bits in left arithmetic shifting. Any condition that sets Overflow also sets the Trap 1 flag (§2.9).

These flags can be read and controlled by certain program control instructions (§2.9, §2.16), but overflow is usually handled by trapping through the setting of Trap 1 (§2.9). The KA10 lacks the trapping feature, so its program must make direct use of the Overflow flag, which is available as a processor condition (via an in–out instruction) that can request a priority interrupt if enabled (§4.3.6). In any event, user overflow is handled by the Monitor according to instructions from the user, as described in Chapter 3 of the appropriate Monitor Calls manual. The conditions detected can only set the arithmetic flags, and the hardware does not clear them; the program must clear them before an instruction if they are to give meaningful information about the instruction afterward. However, the program can check the flags following a series of instructions to determine whether the entire series was free of the types of error detected. Besides indicating error types, the carry flags facilitate performing multiple–precision arithmetic.

2.2.1 Single–Precision Instructions

As noted above, the numbers manipulated by these instructions are single–length except for double–length products and dividends. Such double–length fixed–point numbers are in AC,AC+1, where the magnitude is the 70–bit string in bits 1–35 of the two words, the sign is in bit 0 of the high–order word, and bit 0 of the low–order word contains a copy of the sign. All six instructions have four modes that determine the source of the non–AC operand and the destination of the result.

Mode	*Suffix*	*Source of non–AC operand*	*Destination of result*
Basic		E	AC
Immediate	I	The word 0,E	AC
Memory	M	E	E
Both	B	E	AC and E

ADD **Add**

270	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Add the operand specified by M to AC and place the result in the specified destination. If the sum is $\geq 2^{35}$, set Trap 1, Overflow, and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the sum less 2^{35}. If the sum is $< -2^{35}$, set Trap 1, Overflow, and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the sum plus 2^{35}. Set both carry flags if both addends are negative, or if their signs differ and their magnitudes are equal or if the positive one is the greater in magnitude.

`ADD`	Add	270
`ADDI`	Add Immediate	271
`ADDM`	Add to Memory	272
`ADDB`	Add to Both	273

SUB **Subtract**

274	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Subtract the operand specified by M from AC and place the result in the specified destination. If the difference is $\geq 2^{35}$, set Trap 1, Overflow, and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{35}. If the difference is $< -2^{35}$, set Trap 1, Overflow, and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the difference plus 2^{35}. Set both carry flags if the signs of the operands are the same and AC is the greater or the two are equal, or if the signs of the operands differ and AC is negative.

`SUB`	Subtract	274
`SUBI`	Subtract Immediate	275
`SUBM`	Subtract to Memory	276
`SUBB`	Subtract to Both	277

MUL Multiply

224	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Multiply AC by the operand specified by *M* and place the high–order word of the double–length result in the specified destination. If *M* specifies AC as a destination, place the low–order word in AC+1. If both operands are -2^{35}, set Trap 1 and Overflow; the double–length result stored is -2^{70}.

`MUL`	Multiply	224
`MULI`	Multiply Immediate	225
`MULM`	Multiply to Memory	226
`MULB`	Multiply to Both	227

Caution

In the KA10, an AC operand of -2^{35} is treated as though it were $+2^{35}$, producing the incorrect sign in the product.

IMUL Integer Multiply

220	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Multiply AC by the operand specified by *M* and place the sign and the 35 low–order magnitude bits of the product in the specified destination. Set Trap 1 and Overflow if the product is $\geq 2^{35}$ or $< -2^{35}$ (i.e., if the high–order word of the double length product is not null); the high–order word is lost.

`IMUL`	Integer Multiply	220
`IMULI`	Integer Multiply Immediate	221
`IMULM`	Integer Multiply to Memory	222
`IMULB`	Integer Multiply to Both	223

DIV Divide

234	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

If division is not possible, either because the operand specified by M is zero or because the quotient would not be representable (i.e., if the quotient is larger than $2^{35} - 1$ or smaller than -2^{35}), set Trap 1, Overflow, and No Divide and go immediately to the next instruction without affecting the original AC or memory operand in any way.[10]

If division is possible, divide the double–length number contained in AC,AC+1 by the specified operand, calculating a quotient of 35 magnitude bits including leading zeros. Place the unrounded quotient in the specified destination. If M specifies AC as a destination, place the remainder, with the same sign as the dividend, in AC+1.

`DIV`	Divide	234
`DIVI`	Divide Immediate	235
`DIVM`	Divide to Memory	236
`DIVB`	Divide to Both	237

Note: The magnitude restriction is required because the quotient developed would exceed 36 bits.

IDIV Integer Divide

230	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

If the operand specified by M is zero, or AC contains -2^{35} and the operand specified by M is -1 (except in the KS10), set Trap 1, Overflow, and No Divide and go immediately to the next instruction without affecting the original AC or memory operand in any way. Otherwise, divide AC by the specified operand, calculating a quotient of 35 magnitude bits including leading zeros. Place the unrounded quotient in the specified destination. If M specifies AC as the destination, place the remainder, with the same sign as the dividend, in AC+1.

`IDIV`	Integer Divide	230
`IDIVI`	Integer Divide Immediate	231
`IDIVM`	Integer Divide to Memory	232
`IDIVB`	Integer Divide to Both	233

---

[10] Division is always possible when the magnitude of the operand in AC is smaller than the magnitude of the operand specified by M. Division is never possible when the magnitude of the operand in AC is greater than the magnitude of the operand specified by M. When the magnitudes are equal, the signs of the operands (and sometimes the contents of AC+1) determine whether or not the division is possible. If the divisor is positive and the dividend is negative, division is allowed; if both operands are positive, division is impossible. When the divisor is negative, the contents of AC+1 determine whether division is possible: if the dividend is positive, the division is possible only when the contents of AC+1 are less than the magnitude of the divisor; if the dividend is negative, division is possible only if there are bits of significance in AC+1.

Caution

In the KS10, dividing -2^{35} by -1 gives -2^{35} with no error indication. In the KA10, KI10, and a KL10 with microcode version before 271 (which includes all single–section KL10s), the overflow action is also triggered by attempting to divide -2^{35} by $+1$.

2.2.2 Double–Precision Instructions[11]

There are just four instructions for the four basic operations, and they have no modes. All use AC and memory operands and place the result in the accumulators. Memory operands are double length in location $E, E+1$.[12] Most AC operands are double–length in AC,AC+1, but products and dividends are quadruple–length in AC,AC+1,AC+2,AC+3, and the double–length remainder in division is placed in AC+2,AC+3. Double–length numbers have the same format as the products and dividends of single–precision instructions discussed above. In quadruple–length numbers, AC contains the highest order word; the magnitude is the 140–bit string in bits 1–35 of the four words, the sign is in bit 0 of the highest order word, and copies of the sign are kept in bit 0 of the other three words.

DADD Double Add

114	A	I	X	Y
0 8	9 12	13	14 17	18 35

Add the operand in location $E, E+1$ to AC,AC+1 and place the result in AC,AC+1. If the sum is $\geq 2^{70}$, set Trap 1, Overflow, and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the sum less 2^{70}. If the sum is $< -2^{70}$, set Trap 1, Overflow, and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the sum plus 2^{70}. Set both carry flags if both addends are negative, or if their signs differ and their magnitudes are equal or the positive one is the greater in magnitude.

DSUB Double Subtract

115	A	I	X	Y
0 8	9 12	13	14 17	18 35

Subtract the operand in location $E, E+1$ from AC,AC+1 and place the result in AC,AC+1. If the difference is $\geq 2^{70}$, set Trap 1, Overflow, and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{70}. If the difference is $< -2^{70}$, set Trap 1, Overflow, and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the difference plus 2^{70}. Set both carry flags if the signs of the operands are the same and AC,AC+1 is the greater or the two are equal, or if the signs of the operands differ and AC,AC+1 is negative.

---

[11] In the KI10 and KA10, these instructions trap as unassigned codes (§2.16).

[12] Refer to the description of $E, E+1$ on page 52.

DMUL **Double Multiply**

116	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Multiply AC,AC+1 by the operand in location $E, E+1$ and place the quadruple–word result in AC–AC+3. If both operands are -2^{70}, set Trap 1 and Overflow; the quadruple–length result stored is -2^{140}.

DDIV **Double Divide**

117	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the magnitude of the high–order double word of the quadruple–length number in AC–AC+3 is greater than or equal to the magnitude of the operand in location $E, E+1$, set Trap 1, Overflow, and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way. Otherwise, divide the quadruple length number contained in the accumulators by the operand in location $E, E+1$, calculating a quotient of 70 magnitude bits including leading zeros. Place the unrounded quotient in AC,AC+1and the double–length remainder, with the same sign as the dividend, in AC+2,AC+3.

2.3 Floating–Point Arithmetic[13]

For floating–point arithmetic the PDP–10 has instructions for scaling the exponent (which is multiplication of the entire number by a power of 2); performing addition, subtraction, multiplication, and division of numbers in single– and double–precision floating–point format; and converting single–precision numbers from fixed–format to floating and vice versa. Except for conversion operations, instructions treated here interpret all operands as floating–point numbers in the format given in §1.5.2 and generate results in that format. The reader is strongly advised to reread §1.5.2 if he does not remember the format in detail.

For the four standard arithmetic operations in single–precision, the program has a choice of modes, determining mostly the destination of the result, and can select whether or not the result will be rounded. Rounding produces the greatest consistent precision using only single–length operands. Instructions without rounding save time in one–word operations where rounding is of no significance. Actually, the result is formed in a double–length register in addition, subtraction, and multiplication, wherein any bits of significance in the low–order part supply information for normalization, and then for rounding if requested. Consider addition as an example. Before adding, the processor right shifts the fractional part of the operand with the smaller exponent until its bits correctly match the bits of the other operand in order of magnitude. Thus, the smaller operand could disappear entirely, having no effect on the result ("result" will always be taken to mean the information (one word or two) stored by the instruction, regardless of the number of significant bits it contains or even whether it is the correct answer). In any event, the significance of the result depends on the relative values of

---

[13] In a KA10 without floating point hardware, all of the instructions presented in this section trap as unassigned codes (§2.16). However, no KA10 without floating point hardware was shipped to a customer, so this footnote has no particular utility.

the operands. For example, a subtraction involving two like–signed numbers whose exponents are equal and whose fractions differ only in the LSB (least significant bit) gives a result containing only one bit of significance. In division the processor always calculates a one–word quotient that requires no normalization if the original operands are normalized. An extra quotient bit is calculated for rounding when requested.

The instruction that converts fixed–point to floating–point assumes the operand is an integer and always normalizes and rounds the result. In the opposite direction, only the integral part of the result is saved, and rounding is an option of the program.[14]

The instructions for the four standard operations using double–precision have no modes. In division the processor calculates a two–word rounded quotient that is already normalized if the original operands are normalized. In addition, subtraction, and multiplication, the result is formed in a triple–length register, wherein bits of significance in the lowest–order part supply information for normalization and then for rounding.

The processor has four flags, Overflow, Floating Overflow, Floating Underflow, and No Divide, that indicate when the exponent is too large or too small to be accommodated or a division cannot be performed because of the relative values of dividend and divisor. Except where the result would be in fixed–point form, any of these circumstances sets Overflow and Floating Overflow. If only these two are set, the exponent of the answer is too large; if Floating Underflow is also set, the exponent is too small. No Divide being set means the processor failed to perform a division, an event that can be produced only by a zero divisor if all non–zero operands are normalized. Any condition that sets Overflow also sets the Trap 1 flag. These flags can be read and controlled by certain program control instructions (§2.9, §2.16), but overflow is usually handled by trapping through the setting of Trap 1. The KA10 lacks the trapping feature, so its program must make direct use of Overflow and Floating Overflow, which are available as processor conditions (via an in–out instruction) that can request a priority interrupt if enabled (§4.3.6). The conditions detected can only set the arithmetic flags and the hardware does not clear them, so the program must clear them before a floating–point instruction if they are to give meaningful information about the instruction afterward. However, the program can check the flags following a series of instructions to determine whether the entire series was free of the types of error detected.

The floating–point hardware functions at its best if given operands that are either normalized or zero, and it normalizes a non–zero result.[15] Unnormalized operands will generally cause loss of precision in results. However, such malformed numbers must be created deliberately by the programmer—the processor never produces them. An operand with a zero fraction and a non–zero exponent can give wild answers in additive operations because of extreme loss of significance; e.g., adding $\frac{1}{2} \times 2^2$ and 0×2^{69} gives a zero result, as the first operand (having a smaller exponent) looks smaller to the processor and is shifted to oblivion. A number with a 1 in bit 0 and 0s in bits 9–35 is not simply an incorrect representation of zero, but rather an unnormalized "fraction" with value -1. These

---

[14] Rounding to an integer value is a different procedure than the rounding of floating–point values described above.

[15] The processor normalizes the result by shifting the fraction and adjusting the exponent to compensate for the change in value. Each shift and accompanying exponent adjustment thus multiply the number both by 2 and by $\frac{1}{2}$ simultaneously, leaving its value unchanged.

With normalized operands, the processor uses at most two bits of information from the lowest–order part to normalize the result. In multiplication this is obvious, since squaring the minimum fractional magnitude $\frac{1}{2}$ gives a result of $\frac{1}{4}$. In an addition or subtraction of numbers that differ greatly in order of magnitude, the result is determined almost completely by the operand of greater order. Addition or subtraction involving two numbers with equal exponents requires no shifting beforehand, so there is no information in the lowest–order part. Hence, an addition or subtraction never requires shifting both before the operation and in the normalization; when there is no prior shifting, the normalization brings in 0s.

unnormalized numbers can produce an incorrect answer in any operation. To normalize a number, add (e.g., FAD, DFAD, or GFAD) zero to it.

2.3.1 Scaling

The following two instructions change the exponent of a number without changing the significance of the fraction. In other words they multiply the number by a power of 2 and are thus analogous to arithmetic shifting of fixed–point numbers, except that no information is lost, although the exponent can overflow or underflow. The amount added to the exponent is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 or 2^{11} in magnitude respectively for single precision or extended–range operations. In other words, the effective scale factor E is the number composed of bit 18 (which is the sign) and bits 28–35 or 25–35 of the calculation result. Hence, the programmer may specify the factor directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating it. A positive E increases the exponent, a negative E decreases it; E is thus the power of 2 by which the number is multiplied. The scale factor lies in the range -256 to $+255$ or -1024 to $+1023$.

FSC **Floating Scale**

132	A	I	X	Y
0 8	9 12	13	14 17	18 35

If the fractional part of AC is zero, clear AC. Otherwise add the scale factor given by E to the exponent part of AC (thus multiplying AC by 2^E), normalize the resulting word bringing 0s into bit positions vacated at the right, and place the result back in AC.

The amount added to the exponent is specified by the result of the effective–address calculation taken as a signed number (in twos–complement notation) modulo 2^8 in magnitude. In other words the effective scale–factor E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence, the programmer may specify the factor directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the factor. A positive E increases the exponent; a negative E decreases it. Thus, E is the power of 2 by which the number is multiplied. The scale factor lies in the range -256 to $+255$.

Note

> A negative E is represented in standard twos–complement notation, but the hardware compensates for this when scaling the exponent.

If the exponent after normalization is > 127, set Trap 1, Overflow, and Floating Overflow; the result stored has a exponent 256 less than the correct value. If the exponent after normalization is < -128, set Trap 1, Overflow, Floating Overflow, and Floating Underflow; the result stored has an exponent 256 greater than the correct value.[16]

FSC can be used to float a fixed number with twenty–seven or fewer significant bits. To float an

---

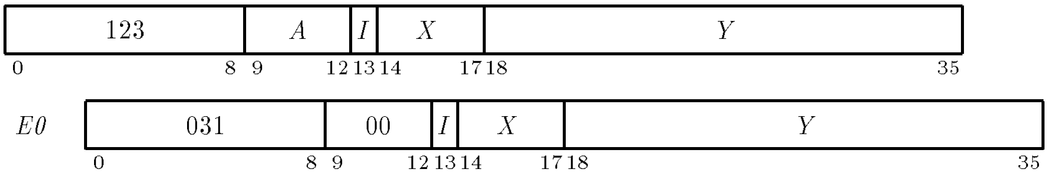
[16] *Caution*: In the KI10 and KA10 only, extreme overflows are not detected properly in this instruction. An exponent > 255 sets Floating Underflow, and an exponent < -256 fails to set it.

integer contained within AC bits 9–35,

```
        FSC     AC,233
```

inserts the correct exponent to move the binary point from the right end to the left of bit 9 and then normalizes ($233_8 = 155_{10} = 128 + 27$). This application of FSC is useful only in the KA10, which lacks the number conversion instructions described in §2.3.2.

GFSC Giant Floating Scale[17]



If the fractional part (bits 12–35) of AC are zero, clear AC,AC+1. Otherwise, scale the G–Format number in AC,AC+1 by adding the immediate operand *E1* to the exponent found in bits 1–11 of AC (thus multiplying the number by 2^{E1}); normalize the double word operand bringing 0s into bit positions vacated at the right; store the result in AC,AC+1.

E1 is interpreted as a twos–complement number composed of bit 18 (the sign) and bits 25–35. The programmer may specify the scale factor directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the scale factor. A positive *E1* increases the exponent; a negative *E1* decreases it. Thus, *E1* is the power of 2 by which the number is multiplied. The scale factor lies in the range -2048 to $+2047$.

Set Overflow, Floating Overflow, and Trap 1 if the resulting exponent exceeds 3777_8 (+1023 decimal); the result stored has an exponent 2048 less than the correct one. Floating Underflow, Overflow, Floating Overflow, and Trap 1 will be set if the resulting exponent is smaller than zero (-1024 decimal); the result stored has an exponent 2048 greater than the correct one.[18]

2.3.2 Number Conversion[19]

Although FSC can be used to float a fixed–point number, there are three single–precision instructions specifically for converting between integers and floating point numbers. In all cases the operand is taken from location *E* and the converted result is placed in AC.

---

[17] In the KI10 and KA10 this instruction traps as an unassigned code (§2.16).

[18] As of KL10 microcode 2.1[442], extreme overflow is signaled as underflow and vice–versa. Moreover, if the double word operand is unnormalized, overflow may be reported when none actually occurs.

[19] In the KA10 these instructions trap as unassigned codes (§2.16). FIX, FIXR, and FLTR are available in all other processors, but the remaining eight are only available in a XKL-1 (or a KL10 with microcode version 271 or greater). However, the four instructions that convert from giant format to fixed point are not implemented in microcode even on the KL10. They are instead simulated by the Monitor.

FIX Fix

122	A	I	X	Y
0 8	9 12	13	14 17	18 35

If the exponent of the floating point number in location E is > 35, set Overflow and Trap 1 and go immediately to the next instruction without affecting AC or the contents of E in any way. Since the largest fixed–point magnitude (without considering sign) is $2^{35} - 1$, a floating–point number with exponent greater than decimal 35 (and assumed normalized) cannot be converted to fixed–point.

Otherwise, replace the exponent EXP in the word from location E with bits equal to the sign of the fraction, and shift the (now fixed) extended fraction $N = EXP - 27$ places to the correct position for its order of magnitude, placing the binary point at the right of bit 35. For positive N, shift left, bringing 0s into bit 35 and dropping null bits out of bit 1. For negative N, shift right, bringing null bits (0s for positive, 1s for negative) into bit 1, and then truncate to an integer. Place the result in AC. Truncation produces the integer of largest magnitude less than or equal to the magnitude of the original number. For example, a number $> +1$ but $< +2$ becomes $+1$; a number < -1 but > -2 becomes -1.

Note: The overflow test checks for a value $\geq 2^{35}$, assuming the operand is normalized.

The truncation is that used in Fortran ("fixation"). For it, the processor drops the fractional part in a positive number, but adds 1 to the integral part (as required by twos–complement format) if any bits of significance are shifted out in a negative number.

GFIX Giant Floating to Integer[20]

123	A	I	X	Y
0 8	9 12	13	14 17	18 35

$E0$	024	00	I	X	Y
	0 8	9 12	13	14 17	18 35

Convert the giant–format floating–point number in $E1,E1+1$[21] to a single–precision integer in AC.

If the exponent field of the floating–point number in $E1,E1+1$ is greater than 2043_8 (meaning an effective exponent greater than 35), set Overflow and Trap 1 and do not affect the accumulator.

Otherwise, copy $E1,E1+1$ to an internal double word register, extending the sign bit to bits 1–11. Then shift (as in ASHC, §2.5) by $EXP - 2030_8$, where EXP is the positive exponent from bits 1–11 of $E1$. Store the high–order word of the result in AC.

Note: The overflow test checks for a value $\geq 2^{35}$, assuming the operand is normalized.

This instruction will always truncate towards zero; i.e., 1.9 is fixed to 1 and -1.9 is fixed to -1. This truncation is that specified in the Fortran language for conversion of real to integer. For positive

---

[20] In the KI10 and KS10 this instruction traps as an unassigned code (§2.16). Because of lack of microcode space, in the KL10 this instruction is handled as an unassigned code but operating system software simulates the effect of this instruction.

[21] Refer to the description of $E, E + 1$ on page 52.

numbers, bits shifted off the right end are ignored. For negative numbers, if any "1" bits are shifted off the right end, then 1 is added to bit 35 to make the result closer to zero.

GDFIX Giant Floating to Double Precision Integer[20]

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

	023	00	*I*	*X*	*Y*
E0	0 8	9 12	13	14 17	18 35

Convert the giant–format floating–point number in *E1*,*E1*+1[21] to a double precision integer and place the result in AC,AC+1.

Set Overflow and Trap 1 if the effective exponent is greater than 70 (2106_8 in the exponent field); if an overflow occurs, do not affect the accumulators. Otherwise, copy *E1*,*E1*+1 to AC,AC+1, extending the sign bit to bits 1–11. Then shift (as in ASHC) by $EXP - 2073_8$, where EXP is the positive exponent from bits 1–11 of *E1*. If the result is negative and any "1" bits were shifted off the right end of AC+1, then add 1 to bit 35 of AC+1 to bring the result closer to zero.

Notes: The overflow test checks for a value $\geq 2^{70}$ assuming the operand is normalized.

FIXR Fix and Round

126	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the exponent of the floating point number in location E is > 35, set Overflow and Trap 1 and go immediately to the next instruction without affecting AC or the contents of E in any way.

Otherwise, replace the exponent *EXP* in the word from location E with bits equal to the sign of the fraction, and shift the (now fixed) extended fraction $N = EXP - 27$ places to the correct position for its order of magnitude, placing the binary point at the right of bit 35. For positive N, shift left, bringing 0s into bit 35 and dropping null bits out of bit 1. For negative N, shift right, bringing null bits (0s for positive, 1s for negative) into bit 1, and then round the integral part. Place the result in AC.

Rounding is in the positive direction: the magnitude of the integral part is increased by 1 if the fractional part is $\geq \frac{1}{2}$ in a positive number but $> \frac{1}{2}$ in a negative number. For example, +1.4 (decimal) is rounded to +1, whereas +1.5 and +1.6 become +2; but with negative numbers, −1.4 and −1.5 become −1, whereas −1.6 becomes −2.

Notes: The rounding procedure in FIXR is the Algol standard for real–to–integer conversion. For it, the processor adds 1 to the integral part if the fractional part is $\geq \frac{1}{2}$ in a positive number or (as required by twos–complement format) is $\leq \frac{1}{2}$ in a negative number. This rounding procedure differs from that used in FADR and the other single–precision floating–point arithmetic instructions that round their results.

GFIXR Giant Floating Fix and Round[20]

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	026	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Convert a giant–format floating–point number to a single–precision integer by rounding.

If the exponent field of the giant–format floating–point number in *E1*,*E1*+1[21] is greater than 2043_8 (meaning an effective exponent greater than 35), then this instruction will set Overflow and Trap 1 and not affect the accumulator.

Otherwise, copy *E1*,*E1*+1 to an internal double word register. Extend the sign bit into bits 1–11 of the high–order word of that register. Then shift arithmetically (as in ASHC) the double word register by $EXP - 2030_8$ bits (where EXP is the positive exponent from bits 1–11 of *E1*). The rounding process will consider the data bit to the right of bit 35 in the high–order word. If that bit is a 1, then 1 will be added to bit 35 of the result. If rounding produces the number 2^{35}, set Overflow and Trap 1; the result stored is actually -2^{35}. Rounding is always in the positive direction; see the notes following FIXR.

Notes: The initial overflow test checks for a value $\geq 2^{35}$ assuming the operand is normalized. Rounding can overflow only if the original operand has exponent 35 and fraction $\geq 1 - 2^{-36}$ (in other words, the fraction is positive and begins with a string of thirty–six 1s).

GDFIXR Giant Floating Fix to Double and Round[20]

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	025	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Convert a giant–format floating–point number to a double precision integer by rounding.

If the exponent field of the giant–format floating–point number in *E1*,*E1*+1[21] is greater than 2106_8 (meaning an effective exponent greater than 70), then this instruction will set Overflow and Trap 1 and not affect the accumulator.

Otherwise, *E1*,*E1*+1 is converted to fixed–point by the following procedure: copy *E1*,*E1*+1 to an internal double word register. Extend the sign bit into bits 1–11 of the high–order word of that register. Then shift arithmetically (as in ASHC) the double word register by $EXP - 2073_8$ bits (where EXP is the positive exponent from bits 1–11 of *E1*). If $EXP - 2073_8$ is non–negative, then no rounding will take place.

If $EXP - 2073_8$ is negative, then the double word register was shifted to the right. The rounding process will consider the last data bit that was shifted off the low–order word. If that bit is a 1, then 1 will be added to bit 35 of the low–order word. The double word register is stored in AC,AC+1.

Rounding is always in the positive direction; see the notes following FIXR.

Notes: The overflow test checks for a value $\geq 2^{70}$ assuming the operand is normalized.

FLTR Float and Round

127	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Shift the magnitude part of the fixed–point integer from location *E* right eight places, insert the exponent decimal 35 (in proper form) into bits 1–8 to move the shifted binary point to the left of bit 9 ($35 = 27 + 8$), and normalize the fraction, bringing first the bits originally shifted out and then 0s into bit positions vacated at the right. If fewer than eight bits (left shifts) are needed to normalize, use the next bit to round the single–length fraction. Place the result in AC.

The rounding function is the same as that used by the floating–point arithmetic with rounding instructions (e.g., FADR, see below); the rounding function differs from that used in FIXR.

Fixed–point numbers can always be converted to floating–point. However, precision can be lost because floating–point format provides fewer significant bits. An integer greater than $2^{27} - 1$ cannot be represented exactly in single–precision floating–point unless all its significant bits are clustered within a group of twenty–seven bits.

GFLTR Giant Float and Round[22]

127	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

	030	00	*I*	*X*	*Y*
E0	0 8	9 12	13	14 17	18 35

Convert the integer in *E1* to a giant–format floating–point number in AC,AC+1. Clear accumulator AC+1 to zero. Copy the data from *E1* to AC; shift it right, arithmetically (as in ASHC), 11 places. The sign and exponent 2043_8 (or, if the number is negative, its ones complement, 5734_8) are inserted into bits 0–11. That result is normalized until bit 12 of the high–order word becomes significant. This instruction does not actually do any rounding, because every single–precision integer has an exact representation in giant–format.

DGFLTR Double Giant Float and Round[22]

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

	027	00	*I*	*X*	*Y*
E0	0 8	9 12	13	14 17	18 35

Convert the double precision integer in *E1*,*E1*+1[21] to a giant–format floating–point number and

---

[22] In the KI10 and KS10 this instruction traps as an unassigned code (§2.16).

put the result in AC,AC+1. Copy the data from *E1*,*E1*+1 to AC,AC+1, where it is shifted right, arithmetically (as in ASHC), 11 places, retaining the bits that are shifted out. The sign and exponent 2106_8 (or its ones complement) are inserted into bits 0–11. That result is normalized by left–shifting until bit 12 of the high–order word becomes significant. The left–shift may restore some or all of the bits that were shifted right initially. If any of the bits shifted right remain outside the double word result, then if the leftmost of those bits is 1, the result is modified by adding 1 to bit 35 of the low–order word of the fraction.

GSNGL Giant Floating to Single Floating[22]

	123	*A*	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35
E0	21	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Convert a giant–format quantity (in *E1*,*E1*+1^{21}) to a single–precision floating–point number in AC. Let EXP signify the exponent in positive form from bits 1–11 of *E1*. If $EXP - 2000_8 \geq 128$, set Floating Overflow, Overflow, and Trap 1; do not affect the accumulator. If $EXP - 2000_8 < -128$, set Floating Underflow, Floating Overflow, Overflow, and Trap 1; do not affect the accumulator.[23]

Otherwise, copy the giant–format quantity in *E1*,*E1*+1 to an internal double word register, set bits 1–11 of the high order word to copies of the sign bit, shift the double word register three bits to the left to move the most significant fraction bit from bit 12 to bit 9, and place the quantity $EXP - 1600_8$ (or its ones complement) in bits 1–8. If rounding produces an exponent > 127, set Overflow, Floating Overflow, and Trap 1; the result stored has an exponent 256 greater than the correct one. Store the high–order word in AC.

Notes: Rounding can overflow only if the original operand has exponent 127 and fractional magnitude $\geq 1 - 2^{-28}$.

GDBLE Single Floating to Giant Floating[22]

	123	*A*	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35
E0	22	00	*I*	*X*	*Y*
	0 8	9 12	13	14 17	18 35

Convert the single–precision floating–point quantity in *E1* to a giant–format quantity in AC,AC+1. This conversion is exact.

Copy *E1* to AC; clear AC+1. Let EXP represent the exponent from bits 1–8 of it E1. Shift AC,AC+1 arithmetically three bits to the right, to move the most significant bit of the fraction from bit 9 to bit 12. Place $EXP + 1600_8$ (or its ones complement) in bits 1–11. If the result is negative, clear bit 0 in AC+1.

---

[23] In KL10 microcode version 2.1[442], conversion of a giant–format number whose exponent is in the range 1570_8 to 1577_8 sets underflow and, incorrectly, stores a result.

2.3.3 Single–Precision with Rounding

There are four instructions that use only one–word operands and store a single–length, rounded result. Rounding is away from 0: if the part of the normalized answer being dropped (the low–order part of the fraction) is greater than or equal in magnitude to one half the LSB of the part being retained, the magnitude of the latter part is increased by one LSB.[24] (This rounding is not the same as the rounding used in FIXR.)

The rounding instructions have four modes that determine the source of the non–AC operand and the destination of the result. These modes are like those of fixed–point arithmetic, including an immediate mode that allows the instruction to carry an operand with it.

Mode	*Suffix*	*Source of non–AC operand*	*Destination of result*
Basic		E	AC
Immediate	I	The word $E, 0$	AC
Memory	M	E	E
Both	B	E	AC and E

Note, however, that floating–point immediate uses $E, 0$ as an operand, not $0, E$. In other words, the half word E is interpreted as a sign, an 8–bit exponent, and a 9–bit fraction.

In each of these instructions, the exponent that results from normalization and rounding is tested for overflow or underflow. If the exponent is > 127, set Trap 1, Overflow, and Floating Overflow; the result stored has an exponent 256 less than the correct value. If the exponent is < -128, set Trap 1, Overflow, Floating Overflow, and Floating Underflow; the result stored has an exponent 256 greater than the correct value.

FADR Floating Add and Round

144	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. Compute their sum. If the double–length fraction in the sum is zero, clear the specified destination. Otherwise, normalize the double–length sum bringing 0s into bit positions vacated at the right; round the high–order part; test for exponent overflow or underflow as described above; and place the result in the specified destination.

`FADR`	Floating Add and Round	144
`FADRI`	Floating Add and Round Immediate	145
`FADRM`	Floating Add and Round to Memory	146
`FADRB`	Floating Add and Round to Both	147

---

[24] In the hardware, the rounding operation is actually somewhat more complex than stated here. If the result is negative, the hardware combines rounding with placing the high–order word in twos–complement form by decreasing its magnitude if the low–order part is $< \frac{1}{2}$ LSB. Moreover, an extra single–step renormalization occurs if the rounded word is no longer normalized.

FSBR Floating Subtract and Round

154	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. Subtract the memory operand from the AC operand. If the double–length fraction in the difference is zero, clear the specified destination. Otherwise, normalize the double–length difference bringing 0s into bit positions vacated at the right; round the high–order part; test for exponent overflow or underflow as described above; and place the result in the specified destination.

`FSBR`	Floating Subtract and Round	154
`FSBRI`	Floating Subtract and Round Immediate	155
`FSBRM`	Floating Subtract and Round to Memory	156
`FSBRB`	Floating Subtract and Round to Both	157

FMPR Floating Multiply and Round

164	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. Form the product of the two operands. If the double–length fraction in the product is zero, clear the specified destination. Otherwise, normalize the double–length product bringing 0s into bit positions vacated at the right; round the high–order part; test for exponent overflow or underflow as described above; and place the result in the specified destination.

`FMPR`	Floating Multiply and Round	164
`FMPRI`	Floating Multiply and Round Immediate	165
`FMPRM`	Floating Multiply and Round to Memory	166
`FMPRB`	Floating Multiply and Round to Both	167

FDVR Floating Divide and Round

174	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. If the magnitude of the fraction in AC is greater than or equal to twice that of the fraction in the operand specified by M, set Trap 1, Overflow, Floating Overflow, and No Divide and go immediately to the next instruction without affecting the original AC or memory operand in any way.

Otherwise, compute the quotient of the AC operand divided by the operand specified by M, calculating a quotient fraction of 28 bits (this includes an extra bit for rounding). If the fraction is zero, clear the specified destination. Otherwise, round the fraction, using the extra bit calculated. If the original operands were normalized, the single–length quotient will already be normalized; if it is not, normalize it, bringing 0s into bit positions vacated at the right. Test for exponent overflow or underflow as described above. Place the result in the specified destination.

`FDVR`	Floating Divide and Round	174
`FDVRI`	Floating Divide and Round Immediate	175
`FDVRM`	Floating Divide and Round to Memory	176
`FDVRB`	Floating Divide and Round to Both	177

Note: Division fails if the divisor is zero. However, the no–divide condition can also occur if the divisor is unnormalized.

2.3.4 Single–Precision without Rounding

Instructions that do not round are faster for processing floating–point numbers with fractions containing fewer than 27 significant bits. They perform the four standard arithmetic operations with normalization but without rounding. All use AC and the contents of location E as operands and have three modes. They lack an immediate mode but are otherwise analogous to the single–precision instructions with rounding.

Mode	*Suffix*	*Effect*
Basic		High–order word of result stored in AC
Memory	M	High–order word of result stored in E
Both	B	High–order word of result stored in AC and E

In each of these instructions, the exponent that results from normalization is tested for overflow or underflow. If the exponent is > 127, set Trap 1, Overflow, and Floating Overflow; the result stored has an exponent 256 less than the correct value. If the exponent is < −128, set Trap 1, Overflow, Floating Overflow, and Floating Underflow; the result stored has an exponent 256 greater than the correct value.

FAD Floating Add

140	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. Form the floating–point sum of the operands. If the double–length fraction in the sum is zero, clear the destination specified by M. Otherwise, normalize the double–length sum bringing 0s into bit positions vacated at the right; test for exponent overflow or underflow as described above; and place the high–order word of the result in the specified destination.[25]

---

[25] *Caution*: In single–precision floating–point addition, the term with the smaller exponent is right–shifted in a double–length register; specifically, a register with 54 magnitude bits. If the difference in the exponents is < 54, there

`FAD`	Floating Add	140
`FADM`	Floating Add to Memory	142
`FADB`	Floating Add to Both	143

FSB Floating Subtract

150	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. Compute the floating–point difference by subtracting the operand specified by M from the AC operand. If the double–length fraction in the difference is zero, clear the destination specified by M Otherwise, normalize the double length difference, bringing 0s into bit positions vacated at the right; test for exponent overflow or underflow as described above; and place the high–order word of the result in the specified destination.[26]

`FSB`	Floating Subtract	150
`FSBM`	Floating Subtract to Memory	152
`FSBB`	Floating Subtract to Both	153

FMP Floating Multiply

160	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. Form the floating–point product of these two operands. If the double–length fraction in the product is zero, clear the destination specified by M. Otherwise, normalize the double–length product bringing 0s into bit positions vacated at the right; test for exponent overflow or underflow as described above; and place the high–order word of the result in the specified destination.

`FMP`	Floating Multiply	160
`FMPM`	Floating Multiply to Memory	162
`FMPB`	Floating Multiply to Both	163

---

is at least one significant bit after the shift (assuming normalized operands). If the difference is > 72 (in the KA10, or > 64 in the KI10), the hardware throws the term away by substituting zero. But when the exponent difference lies in the range 54 to 72 (64), the procedure disposes of all significant bits without actually substituting zero. This means that if the shifted term is positive it appears in the addition as all 0s, but if negative it appears as all 1s. The latter case gives an answer that is less by one LSB.

In the XKL-1and the KL10, no shift is large enough to turn a negative operand to zero. No matter how small the negative operand, it will change the LSB of the result.

[26] The caution given in footnote 25 for addition applies also to subtraction, which is done by adding with the subtrahend negated. Here the lesser answer (as against a true zero substitution) occurs when the term with the smaller exponent is negative after the subtrahend negation; i.e., when the term of smaller magnitude is a positive subtrahend or a negative minuend.

FDV Floating Divide

170	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Interpret the operands specified by M and AC as single–precision floating–point numbers. If the magnitude of the fraction in AC is greater than or equal to twice the magnitude of the fraction in location E, set Trap 1, Overflow, Floating Overflow, and No Divide and go immediately to the next instruction without affecting the original AC or memory operand in any way.

Otherwise, compute the floating–point quotient of AC divided by the contents of location E. Calculate a quotient fraction of 27 bits. If the fraction is zero, clear the destination specified by M. A quotient with a non–zero fraction will already be normalized if the original operands were normalized; if it is not, normalize it, bringing 0s into bit positions vacated at the right. Test for exponent overflow or underflow as described above, and place the single–length quotient in the specified destination.

NOTE

> In the KL10, KS10, and XKL-1, a negative quotient is represented by a twos–complement only when the remainder is zero. Otherwise it is a ones–complement. In the KI10 and KA10, a twos complement is used for a negative quotient regardless of the value of the remainder.

`FDV`	Floating Divide	170
`FDVM`	Floating Divide to Memory	172
`FDVB`	Floating Divide to Both	173

Note: Division fails if the divisor is zero. However, the no–divide condition can also occur if the divisor is unnormalized.

2.3.5 Double–Precision[27]

There are four instructions for the four basic operations; they have no modes. All use AC and memory operands and place the result in the accumulators. Memory operands are double length in location $E, E+1$;[28] AC operands and results are double length in AC,AC+1. All operands are interpreted as double–precision floating–point numbers. All results are normalized regardless of the status of the original operands; except that in KI10 multiplication and division, the result is guaranteed to be normalized only when the original operands are normalized. Except in KI10 division, the result is rounded. The rounding function is the same as that used in single–precision: if the part of the answer being dropped (the low–order part of the fraction) is greater than or equal in magnitude to one half the LSB of the double–length part being retained, the magnitude of the latter part is increased by one LSB (with appropriate adjustment for a twos–complement negative).

---

[27] In the KA10 these instructions trap as unassigned codes (§2.16).

[28] Refer to the description of $E, E+1$ on page 52.

In each of these instructions, the exponent that results from normalization and rounding (if done) is tested for overflow or underflow. If the exponent is > 127, set Trap 1, Overflow, and Floating Overflow; the result stored has an exponent 256 less than the correct value. If the exponent is < -128, set Trap 1, Overflow, Floating Overflow, and Floating Underflow; the result stored has an exponent 256 greater than the correct value.

DFAD Double Floating Add

110	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Add the double–precision floating–point operand in location $E, E+1$ to AC,AC+1. If the fraction in the sum is zero, clear AC,AC+1. Otherwise, normalize the triple–length sum, bringing 0s in at the right; round the high–order double–length part; test for exponent overflow or underflow as described above; and place the result in AC,AC+1.

Note

The KI10 zero test inspects only the high–order 70 bits in the fraction.

DFSB Double Floating Subtract

111	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Subtract the double–precision floating–point operand in location $E, E+1$ from AC,AC+1. If the fraction in the difference is zero, clear AC,AC+1. Otherwise, normalize the triple–length difference, bringing 0s into bit positions vacated at the right; round the high–order double–length part; test for exponent overflow or underflow as described above; and place the result in AC,AC+1.

NOTE

The KI10 zero test inspects only the high–order 70 bits in the fraction.

DFMP Double Floating Multiply

112	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

XKL-1 processor, *KL10*, and *KS10*: Multiply the double–precision floating–point operand in AC,AC+1 by the operand in location $E, E+1$. If the product is zero, clear AC,AC+1. Otherwise, normalize the product, round the high–order double–length part, test for exponent overflow and underflow as described above; and place the result in AC,AC+1.[29]

---

[29] The KL10 considers only one bit to the right of the least significant bit when rounding. If that bit is set, the

KI10: Multiply the double–precision floating–point operand in AC,AC+1 by the operand in location $E, E{+}1$. If the high–order 70 bits of the fraction in the product are zero, clear AC,AC+1. Otherwise, if there are any bits of significance among the high–order 35 bits, do at most one normalization shift if required; if the high–order 35 bits are zero, shift the fraction left 35 places (adjusting the exponent), and then do at most one normalization shift if required. Round the high–order double–length part; test for exponent overflow and underflow as described above; and place the result in AC,AC+1. The 35–bit shift is done only if the original operands are unnormalized. The single normalization shift produces a normalized result for normalized operands.

DFDV Double Floating Divide

113	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the magnitude of the fraction in the double–precision floating–point operand in AC,AC+1 is greater than or equal to twice that of the fraction in the operand in location $E, E + 1$, set Trap 1, Overflow, Floating Overflow, and No Divide and go immediately to the next instruction without affecting the original AC or memory operand in any way.

Otherwise, divide the AC operand by the memory operand, calculating a quotient fraction of 63 bits including one for rounding (62 in the KI10). If the fraction is zero, clear AC,AC+1. Otherwise, in the XKL-1 processor, KL10, and KS10, normalize the quotient and round it using the extra bit calculated. Test for exponent overflow or underflow as described above, and place the quotient in AC,AC+1. The remainder is lost. Division fails if the divisor is zero. However, the no–divide condition can also occur if the divisor is unnormalized.

Note: In the KI10 the quotient is normalized if the original operands are normalized.

2.3.6 Giant–Format Extended–Range Double Precision[30]

There are four instructions for the four basic operations; they have no modes. All use AC and memory operands and place the result in the accumulators. Memory operands are double–length in location $E, E + 1$;[31] AC operands and results are double–length in AC,AC+1. All operands are interpreted as giant–format floating–point numbers. All results are normalized regardless of the status of the original operands. All results are rounded. The rounding function is the same as that used in single precision: if the part of the answer being dropped (the low–order part of the fraction) is greater than or equal in magnitude to one half the LSB of the double–length part being retained, the magnitude of the latter part is increased by one LSB (with appropriate adjustment for a twos–complement negative).

In each of these instructions, the exponent that results from normalization and rounding (if done) is tested for overflow or underflow. If the exponent is > 1023, set Trap 1, Overflow, and Floating Overflow; the result stored has an exponent 2048 less than the correct value. If the exponent is < -1024, set Trap 1, Overflow, Floating Overflow, and Floating Underflow; the result stored has an exponent 2048 greater than the correct value.

KL10 adds one to the least significant bit. This differs from the rounding employed in other instructions.

[30] In the KA10, KI10, and KS10 these instructions trap as unassigned codes (§2.16).

[31] Refer to the description of $E, E + 1$ on page 52.

GFAD Giant Floating Add

102	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Add the giant–format operand in location $E, E+1$[32] to the giant–format operand in AC,AC+1. If the fraction in the sum is zero, clear AC,AC+1. Otherwise, normalize the triple–length sum, bringing 0s in at the right; round the high order double length part; test for exponent overflow or underflow; and place the result in AC,AC+1.

Exponent underflow occurs when two numbers of similar small magnitude and differing signs are added to produce a non–zero result which, when normalized, results in the exponent becoming smaller than -1024. The result stored will have an exponent that is too large by 2048. If exponent underflow occurs, set Floating Underflow, Floating Overflow, Overflow, and Trap 1.

Exponent overflow occurs when two numbers of similar large magnitude and identical signs are added to produce a result which requires a right shift to normalize and which results in an exponent of $+1024$. The exponent stored will be -1024; i.e., too small by 2048. If exponent overflow occurs, set Overflow, Floating Overflow, and Trap 1.

GFSB Giant Floating Subtract

103	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Subtract the giant–format operand in location E, $E+1$ from the operand in AC,AC+1. If the fraction in the sum is zero, clear AC,AC+1. Otherwise, normalize the triple–length difference, bringing 0s in at the right; round the high order double length part; test for exponent overflow or underflow; and place the result in AC,AC+1.

Subtraction is effected by negating the subtrahend and adding. The conditions under which overflow or underflow occur correspond to those described above for GFAD.

GFMP Giant Floating Multiply

106	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Multiply the giant–format operand in AC,AC+1 by the operand in location $E, E+1$. If the product is zero, clear AC,AC+1. Otherwise, normalize the product; round the high order double length part; test for exponent overflow and underflow; and place the result in AC,AC+1.

In multiplication, the exponent of the result is computed by adding the exponents of the operands, with an adjustment for normalization of the result. Underflow occurs when two negative exponents are added to form a result smaller than -1024. Overflow occurs when two positive exponents are added to form a result larger than $+1023$.

---

[32] Refer to the description of $E, E+1$ on page 52.

GFDV Giant Floating Divide

107	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the magnitude of the fraction in the giant–format operand in AC,AC+1 is greater than or equal to twice that of the fraction in the giant–format operand in location $E, E + 1$, set Trap 1, Overflow, Floating Overflow, and No Divide and go immediately to the next instruction without affecting the original AC or memory operand in any way. Division fails when the divisor is zero. However, the no–divide condition can also occur if the divisor is unnormalized.

Otherwise, divide the AC operand by the memory operand, calculating a quotient fraction of 60 bits including one for rounding. If the fraction is zero, clear AC,AC+1. Otherwise, normalize the quotient and round it using the extra bit calculated. Test for exponent overflow or underflow, and place the quotient in AC,AC+1. The remainder is lost.

2.3.7 KA10 Software Double–Precision

These instructions are regarded as obsolete—they are solely for assisting in the KA10 software implementation of double–precision floating–point arithmetic. Hence, they exist only in the KA10, the KI10, and those KL10 processors whose microcode implements them specifically for compatibility with KA10 usage.[33] A programmer who employs these instructions must be aware that the double–length format for KA10 software double–precision is not the same as the standard double–precision format given in §1.5.2. A double–length number in KA10 software double–precision format contains a 54–bit fraction, half of which is in bits 9–35 of each word. The sign and exponent are in bits 0 and 1–8 respectively of the word containing the more significant half, and the standard twos–complement is used to form the negative of the entire 63–bit string. In the remaining part of the less significant word, bit 0 is 0, and bits 1–8 contain a number 27 less than the exponent, but this is expressed in positive form even though bits 9–35 may be part of a negative fraction. For example, the number $2^{18} + 2^{-18}$ has this two–word representation in software double–precision format:

0	10 010 011	100 000 000 000 000 000 000 000 000
0	1 8	9 35

0	01 111 000	000 000 000 100 000 000 000 000 000
0	1 8	9 35

whereas its negative is

1	01 101 100	011 111 111 111 111 111 111 111 111
0	1 8	9 35

0	01 111 000	111 111 111 100 000 000 000 000 000
0	1 8	9 35

---

[33] In KL10 processors that do not support these instructions in microcode, they trap as unassigned codes (§2.16) and are simulated, faithfully and slowly, in software.

Routines for performing software double–precision arithmetic are made possible by the six instructions described here. Four of these do the basic operations with normalization; the double–length number in software format is used as a dividend or it appears as the result in addition, subtraction, or multiplication. The remaining two instructions do not normalize: one negates a software double–length number, the other performs a special unnormalized addition for manipulating low–order parts of numbers without shifting them from their proper positions. In the instructions for the basic operations, the exponent that results from normalization is tested for overflow or underflow. If the exponent is > 127, set Trap 1, Overflow, and Floating Overflow; the result stored has an exponent 256 less than the correct value. If the exponent is < -128, set Trap 1, Overflow, Floating Overflow, and Floating Underflow; the result stored has an exponent 256 greater than the correct value.

NOTE

> The following instructions are solely for assisting in KA10 software double–precision floating–point arithmetic. In any processor that does not implement them, their codes are unassigned and they therefore execute as MUUOs rather than performing the operations given in the following descriptions.

DFN Double Floating Negate

131	A	I	X	Y
0 8	9 12	13	14 17	18 35

Negate the software double–length floating–point number composed of the contents of AC and location E with AC on the left. Do this by taking the twos complement of the number whose sign is AC bit 0, whose exponent is in AC bits 1–8, and whose fraction is the 54–bit string in bits 9–35 of AC and location E. Place the high–order word of the result in AC; place the low order part of the fraction in bits 9–35 of location E without altering the original contents of bits 0–8 of that location.

Note: Usually the double–length number is in two adjacent accumulators, and E equals $A + 1$. There is no overflow test, because negating a correctly formatted floating–point number cannot cause overflow.

DFN AC,AC is undefined.

UFA Unnormalized Floating Add

130	A	I	X	Y
0 8	9 12	13	14 17	18 35

Add the floating–point contents of location E to AC.[34] If the double–length fraction in the sum is zero, clear AC+1. Otherwise normalize the sum only if the magnitude of its fractional part is ≥ 1, and place the high–order part of the result in AC+1. The original contents of AC and E are unaffected.

If the exponent of the sum following the one–step normalization is > 127, set Trap 1, Overflow and

---

[34] The caution given in footnote 25 for FAD applies to this instruction as well.

Floating Overflow; the result stored has an exponent 256 less than the correct value.

Notes. The exponent of the sum is equal to that of the larger addend unless addition of the fractions overflows, in which case it is greater by 1. Exponent overflow can occur only in the latter case.

FADL Floating Add Long

141	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Add the floating–point contents of location E to AC.[34] If the double–length fraction in the sum is zero, clear AC,AC+1. Otherwise, normalize the double–length sum, bringing 0s into bit positions vacated at the right; test for exponent overflow or underflow as described above; and place the high order word of the result in AC. If the exponent of the sum is $< -101 \quad (-128+27)$ or the low–order half of the fraction is zero, clear AC+1. Otherwise place a low–order word for a double–length result in AC+1 by putting a 0 in bit 0, an exponent in positive form that is 27 less than the exponent of the sum in bits 1–8, and the low–order part of the fraction in bits 9–35.

FSBL Floating Subtract Long

151	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Subtract the floating–point contents of location E from AC.[35] If the double–length fraction in the difference is zero, clear AC,AC+1. Otherwise, normalize the double length difference, bringing 0s into bit positions vacated at the right; test for exponent overflow or underflow as described above; and place the high order word of the result in AC. If the exponent of the difference is $< -101 \quad (-128+27)$ or the low–order half of the fraction is zero, clear AC+1. Otherwise, place a low–order word for a double–length result in AC+1 by putting a 0 in bit 0, an exponent in positive form that is 27 less than the exponent of the difference in bits 1–8, and the low–order part of the fraction in bits 9–35.

FMPL Floating Multiply Long

161	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Multiply the floating–point operand in AC by the contents of location E. If the double–length fraction in the product is zero, clear AC,AC+1. Otherwise, normalize the double–length product, bringing 0s into bit positions vacated at the right; test for exponent overflow or underflow as described above; and place the high order word of the result in AC. If the exponent of the product is $> 154(127+27)$ or $< -101(-128+27)$ or the low order half of the fraction is zero, clear AC+1. Otherwise place a low–order word for a double–length result in AC+1 by putting a 0 in bit 0, an exponent in positive form that is 27 less than the exponent of the product in bits 1–8, and the low–order part of the fraction in bits 9–35.

---

[35] The caution given in footnote 26 for FSB applies to this instruction as well.

FDVL **Floating Divide Long**

171	A	I	X	Y
0 8	9 12	13	14 17	18 35

If the magnitude of the software–format double–length fraction in AC,AC+1 is greater than or equal to twice the magnitude of the fraction in location E, set Trap 1, Overflow, Floating Overflow, and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

Otherwise, divide the software–format double–length operand in AC,AC+1 by the contents of location E. Calculate a quotient fraction of 27 bits. If the fraction is zero, clear AC. A quotient with a non–zero fraction will already be normalized if the original operands were normalized; if it is not, normalize it, bringing 0s into bit positions vacated at the right. Test for exponent overflow or underflow as described above, and place the single–length quotient part of the result in AC.

Calculate the exponent for the fractional remainder from the division according to the relative magnitudes of the fractions in dividend and divisor: if the dividend was greater than or equal to the divisor, the exponent of the remainder is 26 less than that of the dividend, otherwise it is 27 less. If the remainder exponent is < -128 or the fraction is zero, clear AC+1. Otherwise, place the floating–point remainder (exponent and fraction) with the sign of the dividend in AC+1.

Note

> In the KL10 microcode implementation of FDVL, a negative quotient is represented by a twos–complement only when the remainder is zero; otherwise, it is a ones–complement (i.e., too small by one LSB). In the KI10 and KA10, a twos–complement is used for a negative quotient regardless of the value of the remainder.

Notes: Division fails if the divisor is zero. However, the no–divide condition can also be satisfied when the divisor is unnormalized.

A non–zero unnormalized dividend whose entire high–order fraction is zero produces a zero quotient. In this case, AC+1 is cleared in the KI10, but it may receive rubbish in other processors.

2.4 Boolean Functions

For logical operations, the PDP–10 has instructions for shifting and rotating (§2.5) as well as for performing the complete set of sixteen Boolean functions of two variables (including those in which the result depends on only one or neither variable). The Boolean functions operate bitwise on full words, so each instruction actually performs thirty–six logical operations simultaneously. Thus, in the AND function of two words, each bit of the result is the AND of the corresponding bits of the operands. The table at the end of the section lists the bit configurations that result from the various operand configurations for all instructions.

Each Boolean instruction has four modes that determine the source of the non–AC operand, if any, and the destination of the result. For an instruction without an operand (one that merely clears a location or sets it to all 1s) the modes differ only in the destination of the result, so basic and immediate modes are equivalent. The same is true also of an instruction that uses only an AC operand. When specified by the mode, the result goes to the accumulator addressed by A, even when there is no AC operand.

Mode	*Suffix*	*Source of non–AC operand*	*Destination of result*
Basic		E	AC
Immediate	I	The word $0, E^\star$	AC
Memory	M	E	E
Both	B	E	AC and E

---

* In section zero the immediate source is $0, E$ in all cases. However, in a non–zero section, setting AC to immediate memory (i.e., SETMI) instead uses the entire extended effective–address E as the source, including the section number (the left part of E).

SETZ Set to Zeros

400	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to all 0s.

`SETZ`	Set to Zeros	400
`SETZI`	Set to Zeros Immediate	401
`SETZM`	Set to Zeros Memory	402
`SETZB`	Set to Zeros Both	403

Note: SETZ and SETZI are equivalent (both clear AC). In them, I, X, and Y are reserved and should be zero. (At present E is ignored.)

AND And with AC

404	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the AND function of the specified operand and AC.

AND	And	404
ANDI	And Immediate	405
ANDM	And to Memory	406
ANDB	And to Both	407

ANDCA And with Complement of AC

410	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the AND function of the specified operand and the complement of AC.

ANDCA	And with Complement of AC	410
ANDCAI	And with Complement of AC Immediate	411
ANDCAM	And with Complement of AC to Memory	412
ANDCAB	And with Complement of AC to Both	413

SETM Set to Memory

414	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Make the contents of the destination specified by M equal to the specified operand.

SETM	Set to Memory	414
SETMI	Set to Memory Immediate	415
SETMM	Set to Memory Memory	416
SETMB	Set to Memory Both	417

If the program is running in a non–zero section, the instruction SETMI is called XMOVEI (§2.1), which performs an analogous function with an extended–immediate operand (effective–address).

Notes: SETM is equivalent to MOVE. In section zero, SETMI moves the word $0,E$ to AC and is thus equivalent to MOVEI. SETMM is a no–op that writes in memory. With non–zero A, SETMB is equivalent to MOVES. In all cases the move instruction is preferred.

ANDCM And Complement of Memory with AC

420	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the AND function of the complement of the specified operand and AC.

`ANDCM`	And Complement of Memory	420
`ANDCMI`	And Complement of Memory Immediate	421
`ANDCMM`	And Complement of Memory to Memory	422
`ANDCMB`	And Complement of Memory to Both	423

SETA Set to AC

424	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Make the contents of the destination specified by M equal to AC.

`SETA`	Set to AC	424
`SETAI`	Set to AC Immediate	425
`SETAM`	Set to AC Memory	426
`SETAB`	Set to AC Both	427

Note: SETA and SETAI are no–ops. In them, I, X, and Y are reserved and should be zero. (At present E is ignored.)

SETAM and SETAB are both equivalent to MOVEM, which is the preferred instruction (all move AC to location E).

XOR Exclusive Or with AC

430	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the exclusive OR function of the specified operand and AC.

`XOR`	Exclusive Or	430
`XORI`	Exclusive Or Immediate	431
`XORM`	Exclusive Or to Memory	432
`XORB`	Exclusive Or to Both	433

The original contents of the destination can be recovered except in XORB, where both operands are replaced by the result. In the other three modes, the replaced operand is restored by repeating the instruction in the same mode; i.e., by taking the exclusive OR of the remaining operand and the result.

IOR Inclusive Or with AC

434	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the inclusive OR function of the specified operand and AC.

`IOR`	Inclusive Or	434
`IORI`	Inclusive Or Immediate	435
`IORM`	Inclusive Or to Memory	436
`IORB`	Inclusive Or to Both	437

Note: The MACRO assembler also recognizes OR, ORI, ORM, and ORB as equivalent to the inclusive OR mnemonics.

ANDCB And Complements of Both

440	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the AND function of the complements of both the specified operand and AC. The result is the NOR function of the operands.

`ANDCB`	And Complements of Both	440
`ANDCBI`	And Complements of Both Immediate	441
`ANDCBM`	And Complements of Both to Memory	442
`ANDCBB`	And Complements of Both to Both	443

EQV Equivalence with AC

444	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the complement of the exclusive OR function of the specified operand and AC (the result has 1s wherever the corresponding bits of the operands are the same).

`EQV`	Equivalence	444
`EQVI`	Equivalence Immediate	445
`EQVM`	Equivalence to Memory	446
`EOVB`	Equivalence to Both	447

The original contents of the destination can be recovered except in EQVB, where both operands are replaced by the result. In the other three modes, the replaced operand is restored by repeating the instruction in the same mode; i.e., by taking the equivalence function of the remaining operand and the result.

SETCA Set to Complement of AC

450	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the complement of AC.

`SETCA`	Set to Complement of AC	450
`SETCAI`	Set to Complement of AC Immediate	451
`SETCAM`	Set to Complement of AC Memory	452
`SETCAB`	Set to Complement of AC Both	453

Note: SETCA and SETCAI are equivalent (both complement AC). In them, I, X, and Y are reserved and should be zero. (At present E is ignored.)

ORCA Inclusive Or with Complement of AC

454	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the inclusive OR function of the specified operand and the complement of AC.

`ORCA`	Or with Complement of AC	454
`ORCAI`	Or with Complement of AC Immediate	455
`ORCAM`	Or with Complement of AC to Memory	456
`ORCAB`	Or with Complement of AC to Both	457

SETCM Set to Complement of Memory

460	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the complement of the specified operand.

`SETCM`	Set to Complement of Memory	460
`SETCMI`	Set to Complement of Memory Immediate	461
`SETCMM`	Set to Complement of Memory Memory	462
`SETCMB`	Set to Complement of Memory Both	463

Notes: SETCMI moves the complement of the word 0,E to AC. SETCMM complements location E.

ORCM Inclusive Or Complement of Memory with AC

464	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the inclusive OR function of the complement of the specified operand and AC.

`ORCM`	Or Complement of Memory	464
`ORCMI`	Or Complement of Memory Immediate	465
`ORCMM`	Or Complement of Memory to Memory	466
`ORCMB`	Or Complement of Memory to Both	467

ORCB Inclusive Or Complements of Both

470	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to the inclusive OR function of the complements of both the specified operand and AC. The result is the NAND function of the operands.

`ORCB`	Or Complements of Both	470
`ORCBI`	Or Complements of Both Immediate	471
`ORCBM`	Or Complements of Both to Memory	472
`ORCBB`	Or Complements of Both to Both	473

SETO Set to Ones

474	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Change the contents of the destination specified by M to all 1s.

`SETO`	Set to Ones	474
`SETOI`	Set to Ones Immediate	475
`SETOM`	Set to Ones Memory	476
`SETOB`	Set to Ones Both	477

Note: SETO and SETOI are equivalent. In them, I, X, and Y are reserved and should be zero. (At present E is ignored.)

For the four possible bit configurations of the two operands, the above sixteen instructions produce the following results. In each case the result as listed is equal to bits 3–6 of the instruction word.

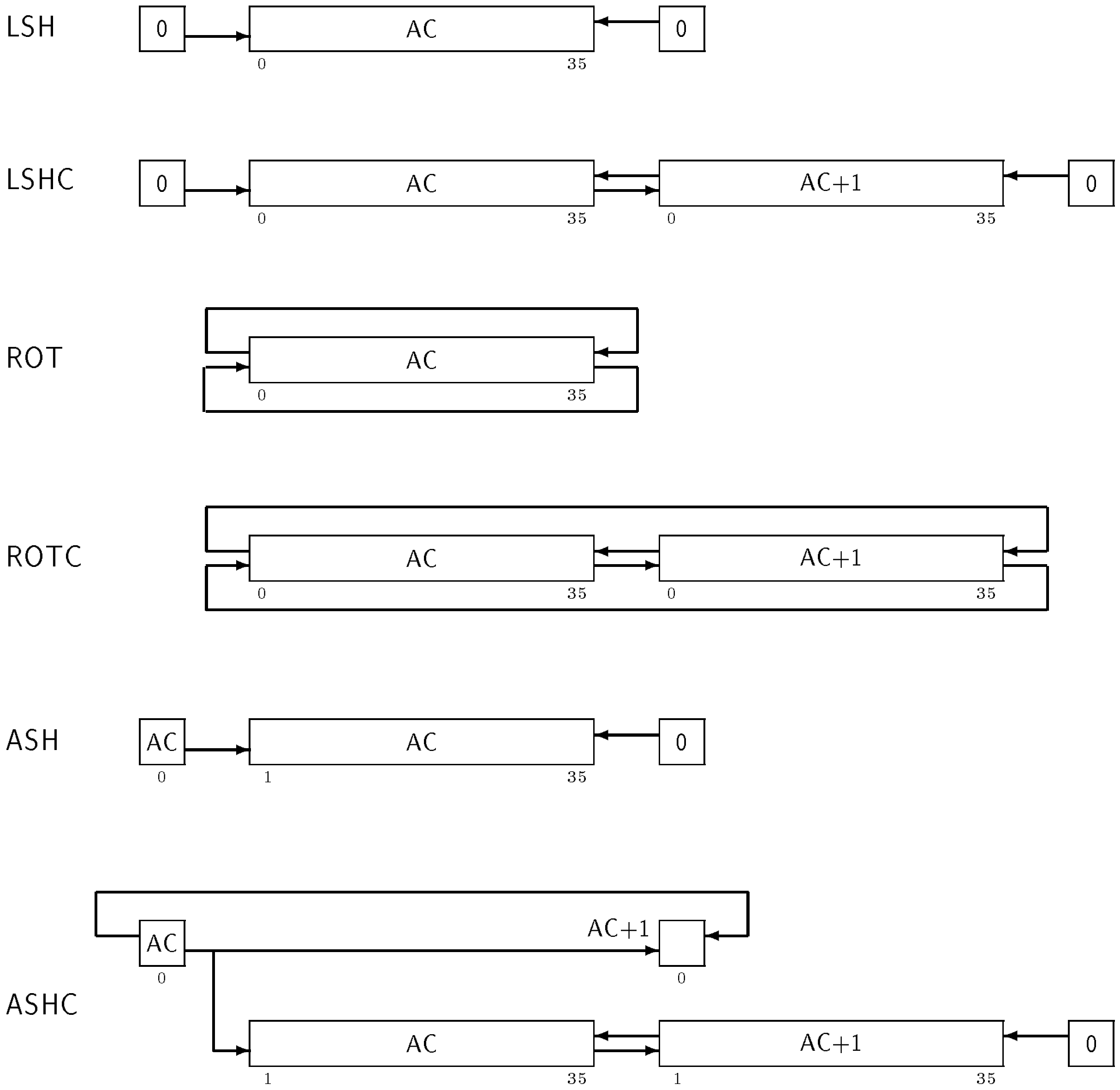
AC Operand *Mode–Specified Operand*	0 0	1 0	0 1	1 1
SETZ	0	0	0	0
AND	0	0	0	1
ANDCA	0	0	1	0
SETM	0	0	1	1
ANDCM	0	1	0	0
SETA	0	1	0	1
XOR	0	1	1	0
IOR	0	1	1	1
ANDCB	1	0	0	0
EQV	1	0	0	1
SETCA	1	0	1	0
ORCA	1	0	1	1
SETCM	1	1	0	0
ORCM	1	1	0	1
ORCB	1	1	1	0
SETO	1	1	1	1

2.5 Shift and Rotate

These instructions shift or rotate right or left the contents of AC or the contents of AC,AC+1, concatenated into a 72–bit register with AC on the left. Shifting is the movement of information bit–to–bit in a register. A logical shift involves the entire word or double word with no distinction among its bits, whereas an arithmetic shift involves only the magnitude, bypassing the sign. Figure 2.1 shows the movement of information these instructions produce in the accumulators. A logical shift moves the bits, with 0s brought in at the end being vacated; information shifted out at the other end is lost. Rotation is a cyclic logical shift where information shifted out at one end is put back in at the other. An arithmetic shift does not affect the sign; but, in a double–length number, where it operates on the 70–bit string made up of the magnitude parts of the two words, it makes bit 0 of the low–order word equal to the sign. Null bits are brought in at the end being vacated: a left shift brings in 0s at the right, whereas a right shift brings in the equivalent of the sign bit at the left. In either case, information shifted out at the other end is lost. A single shift left is equivalent to multiplying the number by 2 (provided no bit of significance is shifted out); a shift right divides the number by 2, with truncation (see footnote 36 on page page 97).

The number of places moved is specified by the result of the effective–address calculation taken as a signed number (in twos–complement notation) modulo 2^8 in magnitude. In other words, the effective shift E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence, the programmer may specify the shift directly in the instruction (perhaps indexed) or give

Figure 2.1: Accumulator Bit Flow in Shift and Rotate Instructions



an indirect address to be used in calculating the shift. A positive E produces motion to the left, a negative E to the right. E is thus the power of 2 by which the number is multiplied.

LSH Logical Shift

242	A	I	X	Y
0 8	9 12	13	14 17	18 35

Shift AC the number of places specified by E. If E is positive, shift left, bringing 0s into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right, bringing 0s into bit 0; data shifted out of bit 35 is lost.

LSHC Logical Shift Combined

246	A	I	X	Y
0 8	9 12	13	14 17	18 35

Shift AC,AC+1 the number of places specified by E. If E is positive, shift left, bringing 0s into bit 71 (bit 35 of AC+1); bit 36 is shifted into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right, bringing 0s into bit 0; bit 35 is shifted into bit 36; data shifted out of bit 71 is lost.

ROT Rotate

241	A	I	X	Y
0 8	9 12	13	14 17	18 35

Rotate AC the number of places specified by E. If E is positive, rotate left; bit 0 is rotated into bit 35. If E is negative, rotate right; bit 35 is rotated into bit 0.

ROTC Rotate Combined

245	A	I	X	Y
0 8	9 12	13	14 17	18 35

Rotate AC,AC+1 the number of places specified by E. If E is positive, rotate left; bit 0 is rotated into bit 71 (bit 35 of AC+1) and bit 36 into bit 35. If E is negative, rotate right; bit 35 is rotated into bit 36 and bit 71 into bit 0.

ASH Arithmetic Shift

240	A	I	X	Y
0 8	9 12	13	14 17	18 35

Shift AC arithmetically the number of places specified by E. Do not shift bit 0. If E is positive, shift left, bringing 0s into bit 35; data shifted out of bit 1 is lost; set Trap 1 and Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative number). If E is negative, shift right, bringing 0s into bit 1 if AC is positive, 1s if negative; data shifted out of bit 35 is lost.[36]

ASHC Arithmetic Shift Combined

244	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Shift AC,AC+1 arithmetically the number of places specified by E. Do not shift bit 0 of AC or AC+1, but make bit 0 of AC+1 equal to AC bit 0 if at least one shift occurs (i.e., if E is non–zero). If E is positive, shift left, bringing 0s into bit 71 (bit 35 of AC+1); bit 37 (bit 1 of AC+1) is shifted into bit 35; data shifted out of bit 1 is lost; set Trap 1 and Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative number). If E is negative, shift right, bringing 0s into bit 1 if AC is positive, 1s if negative; bit 35 is shifted into bit 37; data shifted out of bit 71 is lost.[36]

Note: The effect of a shift on bit 0 of the low–order word is consistent with the convention used for double–length fixed–point numbers. When there is no shift, however, the result may be inconsistent with that convention.

2.6 Arithmetic Testing

These instructions may jump or skip depending on the result of an arithmetic test and may first perform an arithmetic operation on the test word.

2.6.1 Add One to Both Halves of AC and Jump

These two instructions have no modes:

AOBJP Add One to Both Halves of AC and Jump if Positive

252	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Add 1 to each half of AC[37] and place the result back in AC. If the result is greater than or equal to zero (i.e., if bit 0 is 0, and hence a negative count in the left half has reached zero or a positive count has not yet reached 2^{17}), take the next instruction from location E and continue sequential operation from there.

---

[36] An arithmetic right shift truncates a negative result differently from IDIV if 1s are shifted out. The result of the shift is more negative by 1 than the quotient of IDIV. Hence shifting -1 (all 1s) gives -1 as a result.

To obtain the same quotient that IDIV would give with a dividend in A divided by $N = 2^K$, use

```
SKIPGE A
  ADDI A,N-1
  ASH A,-K
```

[37] In the KA10, incrementing both halves of AC together is effected by adding 1000001_8. A count of -2 in AC left is therefore increased to zero if $2^{18} - 1$ is incremented in AC right.

AOBJN Add One to Both Halves of AC and Jump if Negative

253	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Add 1 to each half of AC[37] and place the result back in AC. If the result is less than zero (i.e., if bit 0 is 1, and hence a negative count in the left half has not yet reached zero or a positive count has reached 2^{17}), take the next instruction from location E and continue sequential operation from there.

These two instructions allow the program to keep a control count in the left half of an index register and require only one data transfer to initialize. Problem: Add 3 to each location in a table of N entries starting at TAB. Only four instructions are required.

```
MOVSI     XR,-N      ;Put −N in XR left (clear XR right)
MOVEI     AC,3       ;Put 3 in AC
ADDM      AC,TAB(XR) ;Add 3 to entry
AOBJN     XR,.-1     ;Update XR and go back unless all
                     ;entries have been accounted for
```

Note that, even with extended addressing, AOBJN and AOBJP can be used for this sort of local indexing, because the left half being negative or zero satisfies the criterion for a local index.

2.6.2 Comparisons, Skips, and Jumps

The eight remaining instructions jump or skip if the operand or operands satisfy a test condition specified by the mode.

Mode	*Suffix*
Never	
Less	L
Equal	E
Less or Equal	LE
Always	A
Greater or Equal	GE
Not Equal	N
Greater	G

CAI Compare AC Immediate and Skip if Condition Satisfied

30	*M*	*A*	*I*	*X*	*Y*
0 5	6 8	9 12	13	14 17	18 35

Compare AC with E (i.e., with the word $0,E$) and skip the next instruction in sequence if the condition specified by M is satisfied.

`CAI`	Compare AC Immediate and Do Not Skip	300
`CAIL`	Compare AC Immediate and Skip if AC less than E	301
`CAIE`	Compare AC Immediate and Skip if Equal	302
`CAILE`	Compare AC Immediate and Skip if AC less than or Equal to E	303
`CAIA`	Compare AC Immediate and Always Skip	304
`CAIGE`	Compare AC Immediate and Skip if AC Greater than or Equal to E	305
`CAIN`	Compare AC Immediate and Skip if Not Equal	306
`CAIG`	Compare AC Immediate and Skip if AC Greater than E	307

Note: CAI is a no–op in which I, X, and Y are available for software use.

CAM Compare AC with Memory and Skip if Condition Satisfied

31	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Compare AC with the contents of location E and skip the next instruction in sequence if the condition specified by M is satisfied. The pair of numbers compared may be either both fixed–point or both normalized floating–point.

`CAM`	Compare AC with Memory but Do Not Skip	310
`CAML`	Compare AC with Memory and Skip if AC Less	311
`CAME`	Compare AC with Memory and Skip if Equal	312
`CAMLE`	Compare AC with Memory and Skip if AC Less or Equal	313
`CAMA`	Compare AC with Memory but Always Skip	314
`CAMGE`	Compare AC with Memory and Skip if AC Greater or Equal	315
`CAMN`	Compare AC with Memory and Skip if Not Equal	316
`CAMG`	Compare AC with Memory and Skip if AC Greater	317

Note: CAM is a no–op that references memory.

JUMP Jump if AC Condition Satisfied

32	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Compare AC (fixed or floating) with zero and, if the condition specified by M is satisfied, take the next instruction from location E and continue sequential operation from there.

JUMP	Do Not Jump	320
JUMPL	Jump if AC Less than Zero	321
JUMPE	Jump if AC Equal to Zero	322
JUMPLE	Jump if AC Less than or Equal to Zero	323
JUMPA	Jump Always	324
JUMPGE	Jump if AC Greater than or Equal to Zero	325
JUMPN	Jump if AC Not Equal to Zero	326
JUMPG	Jump if AC Greater than Zero	327

Notes: JUMP is a no–op (instruction code 320 has this mnemonic for symmetry). In it, A, I, X, and Y are available for software use. User programs in TOPS–20 employ JUMP 16, (i.e., JUMP with A set to 16, also known as ERJMP) and JUMP 17, (ERCAL) following system calls to effect error handling.

For an unconditional transfer, JRST is preferred to JUMPA.

SKIP Skip if Memory Condition Satisfied

33	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Compare the contents (fixed or floating) of location E with zero and skip the next instruction in sequence if the condition specified by M is satisfied. If A is non–zero, also place the contents of location E in AC.

SKIP	Do Not Skip, but read Memory	330
SKIPL	Skip if Memory Less than Zero	331
SKIPE	Skip if Memory Equal to Zero	332
SKIPLE	Skip if Memory Less than or Equal to Zero	333
SKIPA	Skip Always	334
SKIPGE	Skip if Memory Greater than or Equal to Zero	335
SKIPN	Skip if Memory Not Equal to Zero	336
SKIPG	Skip if Memory Greater than Zero	337

Notes: If A is zero, SKIP reads memory and discards the data: this resembles a no–op, because it neither changes memory nor changes the accumulators. The fact that SKIP reads memory is used by operating system code to reference a memory location to be sure that it exists and is accessible. When A is not zero, SKIP has the same effect as MOVE; MOVE is preferred. SKIPA is a convenient way to load an accumulator (other than accumulator zero) and skip over an instruction upon entering a loop. For unconditional skips that do not load the accumulator, TRNA is faster than SKIPA, because the former does not read memory (but a JRST is faster than any skip).

AOJ Add One to AC and Jump if Condition Satisfied

34	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Increment AC by 1 and place the result back in AC. Compare the result with zero and, if the condition specified by M is satisfied, take the next instruction from location E and continue sequential operation from there. If AC originally contained $2^{35} - 1$, set Trap 1, Overflow, and Carry 1; if AC originally contained -1, set Carry 0 and Carry 1.

`AOJ`	Add One to AC but Do Not Jump	340
`AOJL`	Add One to AC and Jump if Less than Zero	341
`AOJE`	Add One to AC and Jump if Equal to Zero	342
`AOJLE`	Add One to AC and Jump if Less than or Equal to Zero	343
`AOJA`	Add One to AC and Jump Always	344
`AOJGE`	Add One to AC and Jump if Greater than or Equal to Zero	345
`AOJN`	Add One to AC and Jump if Not Equal to Zero	346
`AOJG`	Add One to AC and Jump if Greater than Zero	347

AOS Add One to Memory and Skip if Condition Satisfied

35	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Increment the contents of location E by 1 and place the result back in E. If A is non–zero, also place the result in AC. Compare the result with zero and skip the next instruction in sequence if the condition specified by M is satisfied. If location E originally contained $2^{35} - 1$, set Trap 1, Overflow, and Carry 1; if location E originally contained -1, set Carry 0 and Carry 1.

`AOS`	Add One to Memory but Do Not Skip	350
`AOSL`	Add One to Memory and Skip if Less than Zero	351
`AOSE`	Add One to Memory and Skip if Equal to Zero	352
`AOSLE`	Add One to Memory and Skip if Less than or Equal to Zero	353
`AOSA`	Add One to Memory and Skip Always	354
`AOSGE`	Add One to Memory and Skip if Greater than or Equal to Zero	355
`AOSN`	Add One to Memory and Skip if Not Equal to Zero	356
`AOSG`	Add One to Memory and Skip if Greater than Zero	357

SOJ Subtract One from AC and Jump if Condition Satisfied

36	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Decrement AC by 1 and place the result back in AC. Compare the result with zero and, if the condition specified by M is satisfied, take the next instruction from location E and continue sequential operation from there. If AC originally contained -2^{35}, set Trap 1, Overflow, and Carry 0; if AC originally contained any other non–zero number, set Carry 0 and Carry 1.

`SOJ`	Subtract One from AC but Do Not Jump	360
`SOJL`	Subtract One from AC and Jump if Less than Zero	361
`SOJE`	Subtract One from AC and Jump if Equal to Zero	362
`SOJLE`	Subtract One from AC and Jump if Less than or Equal to Zero	363
`SOJA`	Subtract One from AC and Jump Always	364
`SOJGE`	Subtract One from AC and Jump if Greater than or Equal to Zero	365
`SOJN`	Subtract One from AC and Jump if Not Equal to Zero	366
`SOJG`	Subtract One from AC and Jump if Greater than Zero	367

SOS Subtract One from Memory and Skip if Condition Satisfied

37	M	A	I	X	Y
0 5	6 8	9 12	13	14 17	18 35

Decrement the contents of location E by 1 and place the result back in E. If A is non–zero, also place the result in AC. Compare the result with zero and skip the next instruction in sequence if the condition specified by M is satisfied. If location E originally contained -2^{35}, set Trap 1, Overflow, and Carry 0; if location E originally contained any other non–zero number, set Carry 0 and Carry 1.

`SOS`	Subtract One from Memory but Do Not Skip	370
`SOSL`	Subtract One from Memory and Skip if Less than Zero	371
`SOSE`	Subtract One from Memory and Skip if Equal to Zero	372
`SOSLE`	Subtract One from Memory and Skip if Less than or Equal to Zero	373
`SOSA`	Subtract One from Memory and Skip Always	374
`SOSGE`	Subtract One from Memory and Skip if Greater than or Equal to Zero	375
`SOSN`	Subtract One from Memory and Skip if Not Equal to Zero	376
`SOSG`	Subtract One from Memory and Skip if Greater than Zero	377

Some of these instructions are useful for determining the relative values of fixed– and floating–point numbers; others are convenient for controlling iterative processes by counting. AOSE is especially useful in an interlock procedure in a multiprogramming environment. Suppose memory contains a routine that must be available to two processes but cannot be used by both at once. When one process finishes the routine, it sets location LOCK to −1. Either process can then test the interlock and make it busy with no possibility of letting the other one in, as AOSE cannot be interrupted once it starts to modify the addressed location.

```
AOSE    LOCK    ;Skip to interlocked code only if
JRST   .-1      ;LOCK is zero after addition
.               ;Interlocked code starts here
.
.
SETOM   LOCK    ;Unlock
```

Since it takes a long time to count to 2^{36}, it is all right to keep testing the lock. (*Note*: this procedure is not suitable where multiple processors may be sharing the lock.)

There are no comparison instructions for double word comparands. For testing the sign of a normalized, double–precision floating point number, SKIPx or JUMPx can be applied to just the first word of the operand. To make other comparisons, the CAMx instructions can be combined, as shown in the example that follows. The ACs identified as `T1` and `T2` are consecutive. This example is a bit subtle; the reader is encouraged to follow the logic of the nested comparisons.

```
DMOVE   T1,X    ;Copy X (and X+1) to T1 and T2
CAMN    T1,Y    ;Compare the top parts of X and Y
CAMLE   T2,Y+1  ;Tops are equal. Test low parts of X and Y
CAMGE   T1,Y    ;
JRST    ...     ;Here if X,X+1 is less than or equal to Y,Y+1
...             ;Here if X,X+1 is greater than Y,Y+1
```

2.7 Logical Testing and Modification

These sixteen instructions (each with four modes) use a mask (a word or half word of bits) to modify and/or test selected bits in AC. The selected bits are the bits in AC that correspond to 1s in the mask; these are called the "masked bits". The programmer chooses the mask, the way in which the masked bits are to be modified, and the condition that the masked bits must satisfy (prior to being modified) to produce a skip.

The basic mnemonics are three letters beginning with "T". The second letter selects the mask and the manner in which it is used:

Mask	*Letter*	*Effect*
Right	R	AC right is masked by E (AC is masked by the word $0, E$)
Left	L	AC left is masked by E (AC is masked by the word $E, 0$)
Direct	D	AC is masked by the contents of location E
Swapped	S	AC is masked by the contents of location E with left and right halves interchanged

The third letter determines the way in which those bits selected by the mask are modified:

Modification	*Letter*	*Effect on AC*
No	N	None
Zeros	Z	Places 0s in all masked bit positions
Complement	C	Complements all masked bits
Ones	O	Places 1s in all masked bit positions

An additional letter may be appended to indicate the mode, which specifies the condition the masked bits must satisfy (prior to being modified) to produce a skip.

Mode	*Suffix*	*Effect*
Never		Never skip
Equal	E	Skip if all masked bits equal 0
Always	A	Always Skip
Not Equal	N	Skip if not all masked bits equal 0 (at least one bit is 1)

The mode names are consistent with those for arithmetic testing and derive from the test method, which ANDs AC with the mask and tests whether the result is equal to zero or is not equal to zero. The programmer may find it convenient to think of the modes as 'Every' and 'Not Every': every masked bit is 0 or not every masked bit is 0. If the mnemonic has no suffix, there is never a skip; the instruction is a no–op if there is also no modification. An "A" suffix produces an unconditional skip—the skip always occurs regardless of the state of the masked bits. The decision to skip (or not) is based on the state of the masked bits prior to any modification called for by the instruction.

TRN Test Right, No Modification, and Skip if Condition Satisfied

60	M	0	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

`TRN`	Test Right, No Modification, but Do Not Skip	600
`TRNE`	Test Right, No Modification, and Skip if All Masked Bits Equal 0	602
`TRNA`	Test Right, No Modification, but Always Skip	604
`TRNN`	Test Right, No Modification, and Skip if Not All Masked Bits Equal 0	606

Note: TRN is a no–op in which I, X, and Y are reserved and should be zero. (At present E is ignored.)

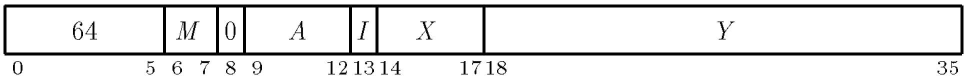
TRZ Test Right, Zeros, and Skip if Condition Satisfied

62	M	0	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

`TRZ`	Test Right, Zeros, but Do Not Skip	620
`TRZE`	Test Right, Zeros, and Skip if All Masked Bits Equaled 0	622
`TRZA`	Test Right, Zeros, but Always Skip	624
`TRZN`	Test Right, Zeros, and Skip if Not All Masked Bits Equaled 0	626

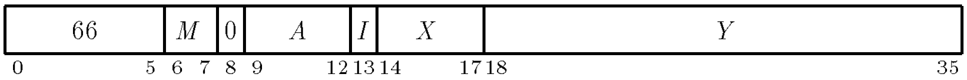
TRC Test Right, Complement, and Skip if Condition Satisfied



If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

`TRC`	Test Right, Complement, but Do Not Skip	640
`TRCE`	Test Right, Complement, and Skip if All Masked Bits Equaled 0	642
`TRCA`	Test Right, Complement, but Always Skip	644
`TRCN`	Test Right, Complement, and Skip if Not All Masked Bits Equaled 0	646

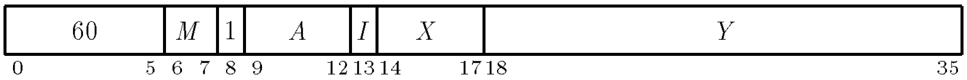
TRO Test Right, Ones, and Skip if Condition Satisfied



If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

`TRO`	Test Right, Ones, but Do Not Skip	660
`TROE`	Test Right, Ones, and Skip if All Masked Bits Equaled 0	662
`TROA`	Test Right, Ones, but Always Skip	664
`TRON`	Test Right, Ones, and Skip if Not All Masked Bits Equaled 0	662

TLN Test Left, No Modification, and Skip if Condition Satisfied



If the bits in AC left corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

`TLN`	Test Left, No Modification, but Do Not Skip	601
`TLNE`	Test Left, No Modification, and Skip if All Masked Bits Equal 0	603
`TLNA`	Test Left, No Modification, but Always Skip	605
`TLNN`	Test Left, No Modification, and Skip if Not All Masked Bits Equal 0	607

Note: TLN is a no–op in which I, X, and Y are reserved and should be zero. (At present E is ignored.)

TLZ Test Left, Zeros and Skip if Condition Satisfied

62	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC left corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

`TLZ`	Test Left, Zeros, but Do Not Skip	621
`TLZE`	Test Left, Zeros, and Skip if All Masked Bits Equaled 0	623
`TLZA`	Test Left, Zeros, but Always Skip	625
`TLZN`	Test Left, Zeros, and Skip if Not All Masked Bits Equaled 0	627

TLC Test Left, Complement, and Skip if Condition Satisfied

64	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC left corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

`TLC`	Test Left, Complement, but Do Not Skip	641
`TLCE`	Test Left, Complement, and Skip if All Masked Bits Equaled 0	643
`TLCA`	Test Left, Complement, but Always Skip	645
`TLCN`	Test Left, Complement, and Skip if Not All Masked Bits Equaled 0	647

TLO Test Left, Ones, and Skip if Condition Satisfied

66	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC left corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

`TLO`	Test Left, Ones, but Do Not Skip	661
`TLOE`	Test Left, Ones, and Skip if All Masked Bits Equaled 0	663
`TLOA`	Test Left, Ones, but Always Skip	665
`TLON`	Test Left, Ones, and Skip if Not All Masked Bits Equaled 0	667

TDN Test Direct, No Modification, and Skip if Condition Satisfied

61	*M*	0	*A*	*I*	*X*	*Y*
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

`TDN`	Test Direct, No Modification, but Do Not Skip	610
`TDNE`	Test Direct, No Modification, and Skip if All Masked Bits Equal 0	612
`TDNA`	Test Direct, No Modification, but Always Skip	614
`TDNN`	Test Direct, No Modification, and Skip if Not All Masked Bits Equal 0	616

Note: TDN has no overt effect on the contents of memory, the accumulators, or the flow of control; thus, in the usual sense, it is a no–op. However, TDN does perform a memory read operation, with all attendant, implementation–dependent, side effects.

TDZ Test Direct, Zeros, and Skip if Condition Satisfied

63	*M*	0	*A*	*I*	*X*	*Y*
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

`TDZ`	Test Direct, Zeros, but Do Not Skip	630
`TDZE`	Test Direct, Zeros, and Skip if All Masked Bits Equaled 0	632
`TDZA`	Test Direct, Zeros, but Always Skip	634
`TDZN`	Test Direct, Zeros, and Skip if Not All Masked Bits Equaled 0	636

TDC Test Direct, Complement, and skip if Condition satisfied

65	*M*	0	*A*	*I*	*X*	*Y*
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E satisfy the condition specified by M, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

`TDC`	Test Direct, Complement, but Do Not Skip	650
`TDCE`	Test Direct, Complement, and Skip if All Masked Bits Equaled 0	652
`TDCA`	Test Direct, Complement, but Always Skip	654
`TDCN`	Test Direct, Complement, and Skip if Not All Masked Bits Equaled 0	656

TDO Test Direct, Ones, and skip if Condition satisfied

67	M	0	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

`TDO`	Test Direct, Ones, but Do Not Skip	670
`TDOE`	Test Direct, Ones, and Skip if All Masked Bits Equaled 0	672
`TDOA`	Test Direct, Ones, but Always Skip	674
`TDON`	Test Direct, Ones, and Skip if Not All Masked Bits Equaled 0	676

TSN Test Swapped, No Modification, and Skip if Condition Satisfied

61	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E with its left and right halves swapped satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

`TSN`	Test Swapped, No Modification, but Do Not Skip	611
`TSNE`	Test Swapped, No Modification, and Skip if All Masked Bits Equal 0	613
`TSNA`	Test Swapped, No Modification, but Always Skip	615
`TSNN`	Test Swapped, No Modification, and Skip if Not All Masked Bits Equal 0	617

Note: TSN is a no–op, in the sense that it has no overt effect. However, it reads memory, with any attendant side effects.

TSZ Test Swapped, Zeros, and Skip if Condition Satisfied

63	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E with its left and right halves swapped satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

`TSZ`	Test Swapped, Zeros, but Do Not Skip	631
`TSZE`	Test Swapped, Zeros, and Skip if All Masked Bits Equaled 0	633
`TSZA`	Test Swapped, Zeros, but Always Skip	635
`TSZN`	Test Swapped, Zeros, and Skip if Not All Masked Bits Equaled 0	637

TSC Test Swapped, Complement, and Skip if Condition Satisfied

65	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E with its left and right halves swapped satisfy the condition specified by M, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

`TSC`	Test Swapped, Complement, but Do Not Skip	651
`TSCE`	Test Swapped, Complement, and Skip if All Masked Bits Equaled 0	653
`TSCA`	Test Swapped, Complement, but Always Skip	655
`TSCN`	Test Swapped, Complement, and Skip if Not All Masked Bits Equaled 0	657

TSO Test Swapped, Ones, and Skip if Condition Satisfied

67	M	1	A	I	X	Y
0 5	6 7	8	9 12	13	14 17	18 35

If the bits in AC corresponding to 1s in the contents of location E with its left and right halves swapped satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

`TSO`	Test Swapped, Ones, but Do Not Skip	671
`TSOE`	Test Swapped, Ones, and Skip if All Masked Bits Equaled 0	673
`TSOA`	Test Swapped, Ones, but Always Skip	675
`TSON`	Test Swapped, Ones, and Skip if Not All Masked Bits Equaled 0	677

With these instructions, any bit throughout all of memory can be used as a program flag, although an ordinary memory location containing flags must be moved to an accumulator for testing or modification. The usual procedure, since locations 1–17 are addressable as index registers, is to use AC 0 as a register of flags (often addressed symbolically as `F`).

Unless one frequently tests flags in both halves of F simultaneously, it is generally most convenient to select bits by 1s in the address part of the instruction word. A given bit selected by a half–word mask M is then set by one of these:

```
TRO F,M           TLO F,M
```

and tested and cleared by one of these:

```
TRZE F,M    TRZN F,M    TLZE F,M    TLZN F,M
```

Suppose one wishes to skip if both bits 34 and 35 are 1 in location L. The following suffices.

```
SETCM   F,L
TRNE    F,3
```

One can refer to a flag in a given bit position within a word as flag X, where X is a binary number containing a single 1 in the same bit position as the flag. This sequence determines whether flags X and Y in the right half of accumulator F are both on:

```
TRC     F,X+Y       ;Complement flags X and Y
TRCE    F,X+Y       ;Test both and restore states
...                 ;Do this if not both on
...                 ;Skip to here if both on
```

One occasionally sees a subroutine with two entry points to identify two versions of a similar function. The versions are distinguished by setting or clearing a flag, as for example:

```
LABEL1: TDZA    AC,AC       ;Set the flag (all of AC) to 0 & skip
LABEL2: MOVEI   AC,1        ;Set the entry flag to 1
        ...
```

2.8 Half–Word Data Transmission

These instructions move a half word and may modify the contents of the other half of the destination location. There are sixteen instructions; however, in a non–zero section, the immediate mode of one of them acts in a special way and is treated as a separate instruction. The sixteen forms are distinguished by which half of the source word is moved to which half of the destination and by which of four possible operations is performed on the other half of the destination. The basic mnemonics are three letters that indicate the transfer,

`HLL`	Left half of source to left half of destination
`HRL`	Right half of source to left half of destination
`HRR`	Right half of source to right half of destination
`HLR`	Left half of source to right half of destination

plus a fourth, if necessary, to indicate the operation:

Operation	*Suffix*	*Effect on Other Half of Destination*
Do nothing		None
Zeros	Z	Places 0s in all bits of the other half
Ones	O	Places 1s in all bits of the other half
Extend	E	Places the sign (the leftmost bit) of the half word moved in all bits of the other half. This action extends a right half–word number into a full–word number but is valid arithmetically only for positive left half–word numbers: the right extension of a number requires 0s regardless of sign. (Hence, the Zeros operation should be used to extend a left half–word number.)

An additional letter may be appended to indicate the mode, which determines the source and destination of the half word moved:

Mode	*Suffix*	*Source*	*Destination*
Basic		E	AC
Immediate	I	The word $0, E^\star$	AC
Memory	M	AC	E
Self	S	E	E, but full word result also goes to AC if A is non–zero

---

[⋆] In section zero, the immediate source is $0, E$ in all cases, and selecting the left half of the source clears the selected half of the destination. However, in a non–zero section, the immediate left–to–left transfer (XHLLI) instead uses the entire extended effective–address E as the source, and it thus transfers the section number (E_{L}).

HLL Half Word Left to Left

500	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by M to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination left half are lost.

`HLL`	Half Left to Left	500
`HLLI`	Half Left to Left Immediate	501
`HLLM`	Half Left to Left Memory	502
`HLLS`	Half Left to Left Self	503

If the program is running in a non–zero section, the instruction HLLI is called XHLLI (see below), which performs an analogous function with an extended–immediate operand (effective–address).

Notes: In section zero, HLLI clears AC left. If A is zero, HLLS is a no–op; otherwise, it is equivalent to MOVE.

HLLZ Half Word Left to Left, Zeros

510	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by M to the left half of the specified destination and clear the destination right half. The source is unaffected; the original contents of the destination are lost.

`HLLZ`	Half Left to Left, Zeros	510
`HLLZI`	Half Left to Left, Zeros, Immediate	511
`HLLZM`	Half Left to Left, Zeros, Memory	512
`HLLZS`	Half Left to Left, Zeros, Self	513

Notes: HLLZI clears AC. If A is zero, HLLZS clears the right half of location E.

HLLO Half Word Left to Left, Ones

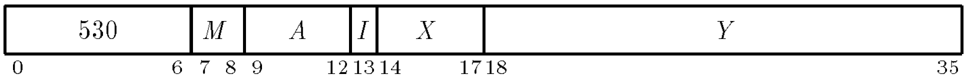
520	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by M to the left half of the specified destination and set the destination right half to all 1s. The source is unaffected; the original contents of the destination are lost.

`HLLO`	Half Left to Left, Ones	520
`HLLOI`	Half Left to Left, Ones, Immediate	521
`HLLOM`	Half Left to Left, Ones, Memory	522
`HLLOS`	Half Left to Left, Ones, Self	523

Note: HLLOI sets AC to all 0s in the left half, all 1s in the right.

HLLE Half Word Left to Left, Extend

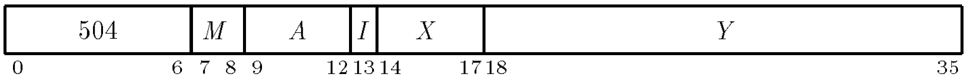


Move the left half of the source word specified by M to the left half of the specified destination and make all bits in the destination right half equal to bit 0 of the source. The source is unaffected; the original contents of the destination are lost.

`HLLE`	Half Left to Left, Extend	530
`HLLEI`	Half Left to Left, Extend, Immediate	531
`HLLEM`	Half Left to Left, Extend, Memory	532
`HLLES`	Half Left to Left, Extend, Self	533

Note: HLLEI is equivalent to HLLZI; it clears AC.

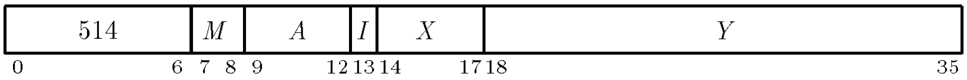
HRL Half Word Right to Left



Move the right half of the source word specified by M to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination left half are lost.

`HRL`	Half Right to Left	504
`HRLI`	Half Right to Left Immediate	505
`HRLM`	Half Right to Left Memory	506
`HRLS`	Half Right to Left Self	507

HRLZ Half Word Right to Left, Zeros



Move the right half of the source word specified by M to the left half of the specified destination and clear the destination right half. The source is unaffected; the original contents of the destination are lost.

`HRLZ`	Half Right to Left, Zeros	514
`HRLZI`	Half Right to Left, Zeros, Immediate	515
`HRLZM`	Half Right to Left, Zeros, Memory	516
`HRLZS`	Half Right to Left, Zeros, Self	517

Note: HRLZI loads the word E,0 into AC and is thus equivalent to MOVSI.

HRLO Half Word Right to Left, Ones

524	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the right half of the source word specified by M to the left half of the specified destination and set the destination right half to all 1s. The source is unaffected; the original contents of the destination are lost.

`HRLO`	Half Right to Left, Ones	524
`HRLOI`	Half Right to Left, Ones, Immediate	525
`HRLOM`	Half Right to Left, Ones, Memory	526
`HRLOS`	Half Right to Left, Ones, Self	527

HRLE Half Word Right to Left, Extend

534	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the right half of the source word specified by M to the left half of the specified destination and make all bits in the destination right half equal to bit 18 of the source. The source is unaffected; the original contents of the destination are lost.

`HRLE`	Half Right to Left, Extend	534
`HRLEI`	Half Right to Left, Extend, Immediate	535
`HRLEM`	Half Right to Left, Extend, Memory	536
`HRLES`	Half Right to Left, Extend, Self	537

HRR Half Word Right to Right

540	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the right half of the source word specified by M to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

`HRR`	Half Right to Right	540
`HRRI`	Half Right to Right Immediate	541
`HRRM`	Half Right to Right Memory	542
`HRRS`	Half Right to Right Self	543

Note: If A is zero, HRRS is a no–op (that writes in memory); otherwise, it is equivalent to MOVE.

HRRZ   Half Word Right to Right, Zeros

550	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the right half of the source word specified by M to the right half of the specified destination and clear the destination left half. The source is unaffected; the original contents of the destination are lost.

`HRRZ`	Half Right to Right, Zeros	550
`HRRZI`	Half Right to Right, Zeros, Immediate	551
`HRRZM`	Half Right to Right, Zeros, Memory	552
`HRRZS`	Half Right to Right, Zeros, Self	553

Notes: HRRZI loads the word $0, E$ into AC and is thus equivalent to MOVEI and to SETMI in section zero. If A is zero, HRRZS clears the left half of location E.

HRRO   Half Word Right to Right, One

560	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the right half of the source word specified by M to the right half of the specified destination and set the destination left half to all 1s. The source is unaffected; the original contents of the destination are lost.

`HRRO`	Half Right to Right, Ones	560
`HRROI`	Half Right to Right, Ones, Immediate	561
`HRROM`	Half Right to Right, Ones, Memory	562
`HRROS`	Half Right to Right, Ones, Self	563

HRRE   Half Word Right to Right, Extend

570	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the right half of the source word specified by M to the right half of the specified destination and make all bits in the destination left half equal to bit 18 of the source. The source is unaffected; the original contents of the destination are lost.

`HRRE`	Half Right to Right, Extend	570
`HRREI`	Half Right to Right, Extend, Immediate	571
`HRREM`	Half Right to Right, Extend, Memory	572
`HRRES`	Half Right to Right, Extend, Self	573

HLR Half Word Left to Right

544	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by M to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

`HLR`	Half Left to Right	544
`HLR1`	Half Left to Right Immediate	545
`HLRM`	Half Left to Right Memory	546
`HLRS`	Half Left to Right Self	547

Note: HLRI clears AC right.

HLRZ Half Word Left to Right, Zeros

554	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by M to the right half of the specified destination and clear the destination left half. The source is unaffected; the original contents of the destination are lost.

`HLRZ`	Half Left to Right, Zeros	554
`HLRZI`	Half Left to Right, Zeros, Immediate	555
`HLRZM`	Half Left to Right, Zeros, Memory	556
`HLRZS`	Half Left to Right, Zeros, Self	557

Note: HLRZI clears AC and is thus equivalent to HLLZI.

HLRO Half Word Left to Right, Ones

564	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source specified by M to the right half of the specified destination and set the destination left half to all 1s. The source is unaffected; the original contents of the destination are lost.

`HLRO`	Half Left to Right, Ones	564
`HLROI`	Half Left to Right, Ones, Immediate	565
`HLROM`	Half Left to Right, Ones, Memory	566
`HLROS`	Half Left to Right, Ones, Self	567

Note: HLROI sets AC to all 1s in the left half, all 0s in the right.

HLRE Half Word Left to Right, Extend

574	*M*	*A*	*I*	*X*	*Y*
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by *M* to the right half of the specified destination and make all bits in the destination left half equal to bit 0 of the source. The source is unaffected; the original contents of the destination are lost.

`HLRE`	Half Left to Right, Extend	574
`HLREI`	Half Left to Right, Extend, Immediate	575
`HLREM`	Half Left to Right, Extend, Memory	576
`HLRES`	Half Left to Right, Extend, Self	577

Note: HRLEI is equivalent to HLRZI; it clears AC.

The half–word transmission instructions are very useful for handling addresses, and they provide a convenient means of setting up an accumulator whose right half is to be used for indexing while a control count is kept in the left half. For example, this pair of instructions loads the 18–bit numbers M and N into the left and right halves, respectively, of accumulator XR.

```
HRLZI   XR,M
HRRI    XR,N
```

It is not necessary to clear the other half of XR when loading the first half word. However, any memory instruction that modifies the other half is faster than the corresponding instruction that does not, because the latter must fetch the destination word in order to save half of it. (The difference does not apply to self mode, for here the source and destination are the same.)

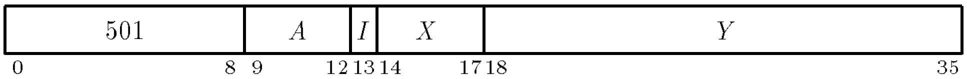
Suppose that at some point one wishes to use the two halves of XR independently as operands (taken as 18–bit positive numbers) for computations. One can begin by moving the left half of XR to the right half of another accumulator AC and isolating the right half of XR in XR.

```
HLRZM   XR,AC         ;Copy XR (left half) to AC (right half)
HLLI    XR,           ;Clear XR (left half).
```

2.8.1 Extended Half–Word Left to Left Immediate

The following instruction uses a half–word transfer for inserting the section number that results from an effective–address calculation into the left half of an accumulator.

XHLLI Extended Half Word Left to Left Immediate



If the program is running in a non–zero section, clear AC bits 0–5 and place the section number (the left part) of the effective–address E in AC bits 6–17. If E is a local AC address, the section number is 1. AC right is unaffected; the original contents of AC left are lost.

If the program is running in section zero, this instruction is called HLLI, which performs an analogous function for section zero (it moves a zero section number).

Notes: The section number given for a local AC address is that of a global AC address. Giving XHLLI with an address 20 or greater without indexing or indirection places the current PC section number in AC left. Thus, it can be used to determine in what section the program is executing.

2.9 Program Control

A program control instruction is one that in some way affects the sequence in which instructions in the program are performed. Most such instructions are actually described in some other category, such as the arithmetic and logical testing instructions above or the yet–to–be discussed stack instructions, UUOs, string–compare instructions, and the condition IO instructions that test device flags. The present section discusses the program flags, overflow trapping, and all program control instructions that do not belong to some other class. Most of these are specifically for handling subroutines. All but one are jumps, although the exception causes the processor to execute an instruction at an arbitrary location and may therefore be regarded as a jump with an immediate and automatic return. All but two of the jumps are unconditional; one exception tests several program flags, the other tests an accumulator.

When an instruction makes the processor leave the normal program sequence to jump to a subroutine or call the Monitor, it must save information sufficient to allow a later return to the original program. Such instructions generally save the states of the program flags and the location at which the disruption in the normal sequence occurred. Saving the program position is referred to as "saving PC," although the quantity actually saved may be the value currently contained in PC or an address 1 greater than that, depending on the circumstances. For example, the same instruction may be used to call a subroutine in a program or to call a service routine in an interrupt. When a return is later made using the saved address in the subroutine case, the instruction that saved PC should not be repeated—the return should be made instead to the instruction following it in normal sequence; i.e., the instruction at the address 1 greater than that originally in PC. In the interrupt case, on the other hand, a subsequent return has nothing to do with the instruction that saved PC—the return should be made to the interrupted instruction, the one PC pointed at when the interrupt occurred. Both cases are covered in the instruction descriptions by the phrase "save PC," and it is to be assumed that the address saved is the one appropriate to the situation in which the instruction is given.

Sometimes regarded as program control, in a somewhat trivial sense, are those instructions that do nothing. The most commonly used no–op is JFCL, which is described here. Other no–ops are among the testing and Boolean instructions discussed previously: SETA, SETAI, SETMM, CAI, CAM, JUMP, TRN, TLN, TDN, and TSN.[38] Of these, SETA, SETAI, CAI, JUMP, TRN, and TLN are preferred, because they do not use the calculated effective–address to reference memory.

2.9.1 The Execute Instruction

This instruction allows the programmer to execute the contents of any memory location as an instruction without altering the normal program counting sequence to do it.

XCT **Execute**

256	A	I	X	Y
0 8	9 12	13	14 17	18 35

If A is zero or the processor is in user mode or is a KA10, execute the contents of location E (the "target instruction") as an instruction.[39] Any instruction may be executed, including another XCT. If an XCT executes a skip instruction, the skip is relative to the location of the XCT (the first XCT if there are several in a chain). If an XCT executes a jump, program flow is altered as specified by the effective–address of the jump instruction. If an XCT executes a jump that saves the PC, the saved PC contains an address 1 greater than the location of the XCT (the first XCT if there are several in a chain).

In an extended processor, if the effective–address of the XCT is in a different section than the PC's section, then the target instruction's effective–address computation begins as a local address in the target instruction's section. For example, if the PC contains an address in section 7 and location 2001234 contains JRST 500 then, when the instruction XCT @[2001234] is performed, the processor will continue with 2000500 in PC; i.e., the JRST instruction specifies the local address 500 which is interpreted as being in the section containing the target instruction.[40]

In an extended processor, when an XCT executes a stack instruction that uses a local stack pointer,

---

[38] KA10 instruction codes 247 and 257 are reserved for instructions installed specially for a particular system. They execute as no–ops when run on a KA10 that contains no special hardware for them, but for program compatibility it is advised that they not be used regularly as no–ops.

[39] *Caution*: In a private program (concealed or kernel mode) on the KI10, never give an XCT that executes an instruction in a public page. It does not work.

[40] However, when the target instruction traps to the Monitor (because it is an MUUO or JSYS or because of arithmetic overflow or stack overflow), then the fact that the instruction came from a section different than the PC section is discarded by the hardware. Thus, parameters to JSYS or MUUO functions, if local addresses, are interpreted being in the PC section.

the stack pointer is interpreted as being in the PC section (rather than being in the section from which the target instruction was fetched).

In executive mode this instruction performs as stated only when A is zero.[41] Non–zero A results in a so–called "previous–context XCT" or PXCT, whose ramifications are far more widespread than indicated here. PXCT is a very special instruction for the exclusive use of the Monitor, and it is described in the section on memory management in the system operations chapter for each processor.

2.9.2 Conditional Jumps

JFFO Jump if Find First One

243	A	I	X	Y
0 8	9 12	13	14 17	18 35

If AC contains zero, clear AC+1 and go on to the next instruction in sequence. If AC contains a non–zero value, count the number of leading 0s in it (the number of 0s to the left of the leftmost 1), and place the count in AC+1. Take the next instruction from location E and continue sequential operation from there. In either case AC is unaffected; the original contents of AC+1 are lost.

Note: When AC is negative, the second accumulator is cleared, just as it would be if AC were zero. (But the instruction jumps.)

To left–normalize a positive integer in AC use

```
JFFO    AC,.+1
LSH     AC,-1(AC+1)
```

JFCL Jump on Flag and Clear

255	F	I	X	Y
0 8	9 12	13	14 17	18 35

If any flag specified by F is set, clear it and take the next instruction from location E, continuing sequential operation from there. Bits 9–12 are programmed as follows.

Bit	*Flag Selected by a 1*
9	Overflow
10	Carry 0
11	Carry 1
12	Floating Overflow

To select one or a combination of these flags, the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits. The MACRO assembler recognizes mnemonics for

---

[41] The KA10 lacks previous–context capability. On that processor and in user mode on any processor, A is ignored, but it is reserved and should be zero.

some of the 13–bit instruction codes (bits 0–12).

JFCL	JFCL 0,	No–op	25500
JOV	JFCL 10,	Jump on Overflow	25540
JCRY0	JFCL 4,	Jump on Carry 0	25520
JCRY1	JFCL 2,	Jump on Cary 1	25510
JCRY	JFCL 6,	Jump on Carry 0 or 1	25530
JFOV	JFCL 1,	Jump on Floating Overflow	25504

The flags tested by JFCL are described in detail in the next section, "Program Flags". This instruction can be used to clear the selected flags by having the jump address point to the next consecutive location, as in

```
JFCL    17,.+1
```

which clears all four flags without disrupting the normal program sequence. A JFCL that selects no flag is often used as a no–op because it neither fetches nor stores an operand; in this case, bits 18–35 of the instruction word can be used to store information.

JFCL is the only jump that can test any of the flags. However, it can test only four of them, and it saves no information for a subsequent return from a subroutine. Hence, it serves as a branch point for entry into either one of two main paths, which may or may not have a later point in common. For example, it may test the carry flags for the purpose of taking appropriate action in a multiple–precision fixed–point routine.

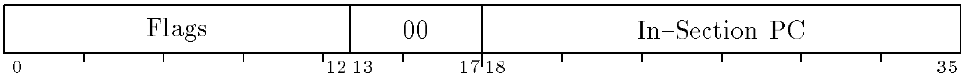
2.9.3 Program Flags

When an instruction saves the program flags, it loads their states into bits 0–12 of a word as shown here

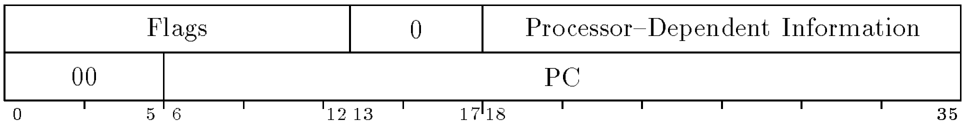
Overflow	Carry 0	Carry 1	Floating Overflow	First Part Done	User	User In–Out	Public	Address Failure Inhibit	Trap 2	Trap 1	Floating Under–flow	No Divide
Previous Context Public						Previous Context User						
0	1	2	3	4	5	6	7	8	9	10	11	12

where the upper part of a double box indicates the flag saved in user mode and the lower part indicates that saved in KL10 and KI10 executive mode. The flag listed in the lower part for bit 6 also applies to KS10 and XKL-1 processor executive mode; however, because these processors have no public mode, bit 0 always receives the state of the Overflow flag and bit 7 is not used. The KS10 also lacks a flag for bit 8. (In KA10 executive mode, bits 0 and 6 receive the Overflow state and the (meaningless) User In–Out state, and bits 7–10 are not used because their flags do not exist.)

Where the flags are saved (in an accumulator or memory location) and what other information is saved with them depends on the instruction and the circumstances of its execution. Whenever the flags are saved, their states are always stored in bits 0–12 of a word in the configuration shown. Some instructions when executed in section zero save the flags and the in–section part of PC in a so–called “PC word” like this.



Note that nothing is stored in bits 13–17; when the PC word is addressed indirectly, it can produce neither indexing nor further indirect addressing. When such instructions are performed in a non–zero section, they generally save only the extended PC without flags. Other instructions, executable only in the XKL-1 processor, the KS10, and the extended KL10, combine the flags and the full PC in what is called a “flag–PC double word” with this format:



In a manner analogous to the PC word, nothing is stored in bits 13–17 of the first word or in bits 0–5 of the second. Hence, when the second word is addressed indirectly, it is interpreted as global address–word which produces neither indexing nor further indirect addressing. Note that, if the second word is used in an index register, it is taken as global or local depending on whether or not bits 6–17 are zero. Nothing is saved in the right half of the flag word except in an extended processor. In the KL10, if the instruction is in an executive program or an interrupt of an executive program, bits 31–35 save the previous–context section for that program (see §4.1.5). In the XKL-1 processor, if the instruction is in an executive program, an interrupt, or a trap, bits 18–35 save the context of the interrupted program: the current and previous accumulator selection and the previous context section.

Certain instructions can use bits 0–12 of a word to set up the program flags to restore them to their original states following an interruption or to control specific situations. Restoration, of course, assumes the flags are being restored from a word in which they were previously saved. When the flags are saved, the flag bits reflect the states and flags appropriate to the current situation. At a transition from one mode to another, the flags saved are those of the mode the processor is leaving, and the flags restored are those for the mode the processor is entering. For example, when the user calls the Monitor, bit 5 of the flag word is set and the User flag must be cleared, either automatically or by a 0 in bit 5 of a restoring flag word. Moreover, Overflow and User In–Out are saved, but the flag bits used for restoration are adjusted to produce the correct states for the previous–context flags. No conflict can result concerning bit 6, because User In–out exists only in user mode, and Previous Context User exists only in executive mode. On the other hand, although only one flag is ever saved in bit 0, at restoration, bit 0 conditions the states of both Overflow and Previous Context Public (if present). The latter is irrelevant in user mode, but the executive programmer must be aware that, if he wishes to use Overflow or give a `JFCL` to test it, its initial state is that assigned to Previous Context Public rather than that resulting from an arithmetic operation. When a return is made to an interrupted executive program via a flag–PC double word in an extended processor, the previous–context is restored from the right half of the flag word. In the KL10 the previous–context section is restored from bits 31–35 of the flag word. In the XKL-1 processor, the the accumulator selection and previous–context section are restored from bits 18–35 of the flag word.

By manipulating the bits used to restore the flags, the programmer can set them up in any way desired, except that the hardware contains interlocks so that a user program cannot clear User or set User In–Out, and no public program can clear Public for itself. As an example, restoring flags from a word in which a trap flag is set will result in an immediate trap.

The following lists the meaning of the information contained in bits 0–12 of a flag word at the time the flags are saved. Bits 0 and 6 are given only for user mode, as the special executive flags are relevant only to the previous–context XCT instruction and are left for the discussion of system operations. Remember (§2.2) that overflow is determined directly from the carries, not the carry flags, which give useful information only if no more than one instruction that can set them occurs between clearing and reading them. The explanations assume the flags reflect normal circumstances—not arbitrary manipulation. An x in a mnemonic indicates any letter (or none) that may appear in the given position to specify the mode; e.g., ADDx comprises ADD, ADDI, ADDM, and ADDB.

Bit *Meaning of a 1 in the Bit*

0 Overflow – any of the following has occurred and set Trap 1:

- A single instruction has set one of the carry flags (bits 1 and 2) without setting the other.
- An ASH or ASHC has left–shifted a 1 out of bit 1 in a positive number or a 0 out of bit 1 in a negative number.
- An MULx has multiplied -2^{35} by itself (product 2^{70}).
- A DMUL has multiplied -2^{70} by itself (product 2^{140}).
- An IMULx has multiplied two numbers with product $\geq 2^{35}$ or $< -2^{35}$.
- An FIX, FIXR, GFIX, or GFIXR has fetched an operand with exponent > 35.
- A GDFIX or GDFIXR has fetched an operand with exponent > 70.
- A GFIXR has fixed a number with exponent 35 and fraction $\geq 1 - 2^{-36}$.
- Floating Overflow has been set (bit 3).
- No Divide has been set (bit 12).

1 Carry 0 – if set without Carry 1 (bit 2) being set, causes Overflow to be set and indicates that one of the following has occurred:

- An ADDx has added two negative numbers with sum $< -2^{35}$.
- A DADD has added two negative numbers with sum $< -2^{70}$.
- An SUBx has subtracted a positive number from a negative number with difference $< -2^{35}$.
- A DSUB has subtracted a positive number from a negative number with difference $< -2^{70}$.
- An SOJx or SOSx has decremented -2^{35}.

If set with Carry 1, indicates that one of these non–overflow events has occurred:

- In an ADDx or DADD, both addends were negative, or their signs differed and their magnitudes were equal or the positive one was the greater in magnitude.
- In a SUBx or DSUB, the signs of the operands were the same and AC was the greater or the two were equal, or the signs of the operands differed and AC was negative.
- An AOJx or AOSx has incremented -1.
- A SOJx or SOSx has decremented a non–zero number other than -2^{35}.

- A MOVNx has negated zero.
- A DMOVN or DMOVNM has negated zero (this condition does not affect the flags in the KI10).

2 Carry 1 – if set without Carry 0 (bit 1) being set, causes Overflow to be set and indicates that one of the following has occurred:

- An ADDx has added two positive numbers with sum $\geq 2^{35}$.
- A DADD has added two positive numbers with sum $\geq 2^{70}$.
- An SUBx has subtracted a negative number from a positive number with difference $\geq 2^{35}$.
- A DSUB has subtracted a negative number from a positive number with difference $\geq 2^{70}$.
- An AOJx or AOSx has incremented $2^{35} - 1$.
- An MOVNx or MOVMx has negated -2^{35}.
- A DMOVN or DMOVNM has negated -2^{70} (this condition does not affect the flags in the KI10).

If set with Carry 0, indicates that one of the non–overflow events listed under Carry 0 has occurred.

3 Floating Overflow – any of the following has set Trap 1 and Overflow:

- In a floating–point instruction (other than FLTR, DFN, or the giant–format instructions) the exponent of the result was > 127.
- In GSNGL the exponent of the memory operand was > 127
- In a giant–range floating–point instruction, GFAD, GFSB, GFMP, GFDV, or GFSC, the exponent of the result was > 1023.
- Floating Underflow (bit 11) has been set.
- No Divide (bit 12) has been set in an FDVx, FDVRx, DFDV or GFDV.

4 First Part Done – the processor is responding to a priority interrupt between the parts of a two–part instruction or to a page failure in the second part. A 1 in this bit indicates that the first part has been completed, and this fact should be taken into account when the processor restarts the instruction at the beginning upon the return to the interrupted program. For example, if an ILDB or IDPB is interrupted after incrementing the pointer but before processing the byte, the pointer now points not to the previous byte, but rather to the byte that should be handled at the return. Thus when the processor restarts the instruction, it must retrieve the pointer but *not* increment it. Note, however, that this flag is solely for use by the hardware: it is saved and restored by the Monitor, and the user should never touch it.[42]

The First Part Done flag is **not** an architectural characteristic; it is an artifact of a specific processor implementation. Different processors implement and use this flag in different ways. No programmer should depend on any particular behavior being repeatable on a different system or with different microcode.

In the present implementation of the XKL-1 and its microcode, First Part Done affects the following instructions: ILDB, IDPB, MOVSRJ, SWPxA, and CVTBDx. First Part Done is cleared

---

[42] On the other hand, if a trap handler (which may be supplied by the user) does any byte operations, the state of this flag must be taken into account; for details refer to the discussion of "special considerations" at the end of each of the sections on the interrupt.

at the start of any instruction, except that the flag restoring instructions are crafted to bypass the clear operation when starting the next instruction. First Part Done cannot be saved by a flag saving instruction. Interrupts and traps are not performed as actual instructions so they can save First Part Done.

5 User – the processor is in user mode.

6 User In–Out – even with the processor in user mode, the program is allowed to use In–Out instructions.

7 Public[43] – the last instruction performed was fetched from a public area of memory; i.e., the processor is in user mode public or executive mode supervisor.

8 Address Failure Inhibit[44] – if an address failure occurs during the execution of the very next instruction, that failure will be ignored. This flag is cleared whenever an instruction (except a flag–restoring JRST) is completed. The purpose of this flag is to let the processor complete the execution of the instruction for which an address failure trap has just been processed.

9 Trap 2[45] – if bit 10 is not also set, stack overflow has occurred. Unless the pager is disabled, the setting of this flag immediately causes a trap as explained in §2.9.6. At present, no hardware condition sets both bits 9 and 10 at the same time.

10 Trap 1[45]—if bit 9 is not also set, arithmetic overflow has occurred. Unless the pager is disabled, the setting of this flag immediately causes a trap as explained in §2.9.6. At present, no hardware condition sets both bits 9 and 10 at the same time.

11 Floating Underflow—any of the following has set Trap 1, Overflow, and Floating Overflow:

- In a single– or double–precision floating–point instruction, the exponent of the result was < -128.
- In a GSNGL instruction, the exponent of the memory operand was < -128.
- In a GFAD, GFSB, GFMP, GFDV, or GFSC instruction, the exponent of the result was < -1024.

12 No Divide—any of the following has set Overflow and Trap 1:

- In a DIVx or DDIV, the high–order half of the dividend was greater than or equal to the divisor.
- In an IDIVx the divisor was zero, or the dividend was -2^{35} and the divisor was ± 1.
- In an FDVx, FDVRx, DFDV, or GFDV, the divisor was zero or the dividend fraction was greater than or equal to twice the divisor fraction in magnitude; in either case Floating Overflow has been set. If normalized operands are used, only a zero divisor can cause floating division to fail.
- In an ADJBP the number of bytes per word was zero.

---

[43] Available only in the KI10 and KL10.

[44] Not available in the KA10 or KS10.

[45] Not available in the KA10

2.9.4 The JRST Instruction

The basic use of this instruction is as a straightforward jump—it is the fastest jump and is the preferred instruction for such use. However, it also allows the programmer to select individual functions by means of bits 9–12 of the instruction word. All KI10 and KA10 functions are included in the instruction set of the more modern processors, but the method of decoding the individual functions is so different that the instruction is described twice, first for the XKL-1/KL10/KS10, then for the earlier processors. Most of the functions are illegal in some circumstances on at least some processors; when a function is illegal, the instruction traps as an MUUO (§2.16) instead of performing the given function. The instruction descriptions explain what each function does when it is legal. Between the two descriptions is a table that indicates which of the functions are legal in which processors and under what circumstances.

JRST **Jump and Restore (XKL-1/KL10/KS10)**

254	*F*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the function specified by F, if it is legal. At present thirteen functions are defined, and for all but one of these the MACRO assembler recognizes individual mnemonics for generating the combined 13–bit instruction codes (including bits 9–12). The defined functions, with their function codes, mnemonics, and combined instruction codes, are as follows.

F	*Mnemonic and Instruction Code*	*Function*
00	JRST 25400	Jump to Location E
01	PORTAL 25404	If the instruction has been taken from a nonpublic area, clear Public; then jump to location E. A location containing a PORTAL is the only valid entry to a nonpublic area. The instruction places the processor in concealed or kernel mode. Note that this function is equivalent to function 0 except when the instruction is taken from a private area by a public program, an event that cannot occur in a KS10 or XKL-1 because they have no public mode.
02	JRSTF 25410	Restore the program flags from bits 0–12 of the final word used in the effective–address calculation (indirect or index word), and jump to location E.

Caution

Restoring the flags requires that the instruction use indexing or indirect addressing. Without indexing or indirection the result is indeterminate.

When executed in a non–zero section, this function traps as an unassigned code (see §2.16).

All flags, excepting the User and Public flags, are restored according to the contents of the corresponding bits in the flag word: a flag is set by a 1 in the bit or cleared by a 0. A 1 in bit 5 sets User, but a 0 has no effect: the Monitor can continue a user program by restoring flags but the user cannot leave User mode by this method. A 0 in bit 6 clears User In–Out, but a 1 sets it only if the JRSTF is being performed by the Monitor; i.e., if User is clear. A 1 in bit 7 sets Public, but a 0 clears it only if the JRSTF is being performed in executive mode with a 1 in bit 5; i.e., User is being set. These conditions imply that the processor is entering user mode: hence, the user cannot enter concealed mode by clearing Public. Although the supervisor can place the processor in User mode concealed, it cannot use this procedure to enter Kernel mode.

Notes: The flag bits are assumed to be in a previously stored PC word. If the PC word was stored in AC (as in a JSP), a common procedure is to use AC to index a zero address; e.g., JRSTF (AC), so that its right half becomes the effective–address (the jump address). If the PC word was stored in memory (as in a JSR), one must address it indirectly (remember, bits 13–17 of the PC word are clear, so, again, its right half is the effective–address). A JRSTF (AC) is considerably faster than a JRSTF @PCWORD.

04 HALT
25420

Load E into PC and halt the processor. While the KL10 is halted, the microcode runs in the halt loop, in which it will handle interrupts on level 0 and will respond to console and diagnostic functions from the front end. The KS10 microcode performs the halt sequence discussed in §4.2.7 and then runs in the halt loop, in which it responds only to commands from the console. The XKL-1 runs its microcode halt loop; the console terminal will respond to console commands as described in §3.2.

Note: The halt occurs, of course, only when the function is legal. For debugging purposes, this function is often used when illegal, in which event it traps as an MUUO (see §2.16).

05 XJRSTF
25424

Restore the program flags and PC (and the previous context, if appropriate) from a flag–PC double word in location $E, E+1$,[46] and continue performing instructions in normal sequence beginning at the location then addressed by PC. User mode restrictions on the manipulation of the flags by the flag bits are the same as those for JRSTF given above. When performed in executive mode, this instruction restores the processor context from the right half of the word in location E.

06 XJEN
25430

Restore the level on which the highest priority interrupt is currently being held (i.e., dismiss the interrupt as described in §3.4, §4.1.1, and §4.2.1), then perform an XJRSTF.

Note: This instruction can be used in any section. It is the only way to dismiss an interrupt routine or restore an interrupted program in a

---

[46] Refer to the description of $E, E+1$ on page 52.

		non–zero section.
07	XPCW 25434	Save the program flags and PC (and the previous–context section, if relevant) in a flag–PC double word in location $E, E + 1$.[46] Then restore the flags and PC from a flag–PC double word in location $E + 2, E + 3$ and continue performing instructions in normal sequence beginning at the location then addressed by PC. Restrictions on the manipulation of the flags by the flag bits are the same as those for JRSTF given above. *Notes*: In a KS10 or an extended KL10, this instruction can be used only for calling an interrupt routine. In the extended KL10, this is the recommended instruction for entering an interrupt routine. The four–word block at location E must be in section zero, because that is the default section for instructions executed in interrupt locations. The return from the routine would typically be made by an XJEN that addresses the same block (i.e., that uses the first double word in the block). In the XKL-1, XPCW is allowed in programs. In the XKL-1, the effect of XPCW is simulated by the processor with respect to the appropriate Interrupt Control Block; see §3.4.4. When executed in executive mode or when simulated for interrupt acceptance, XPCW loads the new processor context from the right half of $E + 2$.
10	*(no mnemonic)* 25440	Restore the level on which the highest priority interrupt is currently being held; i.e., dismiss the interrupt as described in §3.4, §4.1.1, and §4.2.1.
11	XJRSTP 25444	Restore the program flags, context, PC, and the state of the Priority Interrupt system. Perform the XJRSTF function using the contents of E and $E+1$ as data. Then perform WRPI (§3.4.8) using the right–half contents of $E + 2$ as data. This instruction restores processor state as an indivisible operation. This operation is implemented only in the XKL-1; it is legal only in executive mode.
12	JEN 25450	Restore the level on which the highest priority interrupt is currently being held (i.e., dismiss the interrupt, as described in §3.4, §4.1.1, and §4.2.1), then perform a JRSTF.
13	HALTRM 25454	Halt the processor and reload the microcode. This instruction is implemented only in the XKL-1, where it is legal only in executive mode. The I, X, and Y fields are reserved.
14	SFM 25460	Save the program flags in bits 0–12 of memory location E (clear bits 13–17). If the instruction is given in executive mode in an extended processor, save the previous context (in the KL10, the previous–context section, in bits 31–35, clearing bits 18–30; in the XKL-1, the accumulator selection and previous–context section in bits 18–35), otherwise clear bits 18–35. (This instruction is also known to some software as XSFM.)

15	XJRST 25464	Jump to the location given by bits 6–35 of the word addressed by E. Bits 0–5 of the word addressed by E are ignored. This instruction is not implemented in the KS10, where it is handled as an MUUO.

The remaining undefined functions execute as MUUOs, as does any defined function when it is illegal.

One can program a function by giving JRST with the equivalent of an AC address that specifies the function code. For the sixteen forms of the instruction, Table 2.1 lists the individual mnemonic, if any, and indicates where that form of the instruction is legal in each of the different processors.

JRST Jump and Restore (KI10/KA10)

254	F	I	X	Y
0 8	9 12	13	14 17	18 35

Perform the functions specified by F if they are legal; then, if the function was performed and the processor is not halted, take the next instruction from location E and continue sequential operation from there. Bits 9–12 are programmed as follows.

Bit	*Function Produced by a 1 if Legal*
9	Restore the level on which the highest priority interrupt is currently being held; i.e., dismiss the interrupt (§4.3.2, §4.3.5).
10	Halt the processor. When it stops, the AR lights on the KI10 and the MA lights on the KA10 display an address 1 greater than that of the location containing the instruction that caused the halt, and PC displays the jump address (the location from which the next instruction will be taken if the operator causes the processor to resume operation without changing PC). AR or MA actually displays the address of the location that would have been executed next had the JRST been replaced by a no–op. Thus, except for a JRST in an interrupt, the lights point to the location 1 beyond that containing the instruction that caused the halt. This instruction is ordinarily the JRST; however, it could be an XCT or an MUUO.
11	Restore the program flags from bits 0–12 of the final word used in the effective–address calculation. Hence, to restore flags requires that the instruction use indexing or indirect addressing. Restrictions on the manipulation of the flags by the flag bits are the same as those for the KL10 JRSTF given above. (The notes on addressing given there also apply.)
12	*KA10*: Enter user mode. The user program starts at relocated location E. *KI10*: This is the PORTAL instruction. It is simply a jump except when fetched from a nonpublic area, in which case it clears Public. In other words, a location containing a PORTAL is the only valid entry to a nonpublic area, and the instruction places the processor in concealed or kernel mode.

While the KA10 is in user mode, if JRST is executed as an interrupt instruction or by an MUUO, the processor leaves user mode.

Notes: To produce a combination of these functions, the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits; however, MACRO recognizes mnemonics for

Table 2.1: Domains in which JRST Functions are Legal

The meanings of the symbols used to define the legal domains of the functions are as follows.

Yes	Legal everywhere.
Z	Legal only in section zero.
IO	Legal wherever IO instructions are legal; i.e., in user IO mode (User and User In–Out both set) and in kernel mode (executive mode in the XKL-1, KS10, and KA10).
K	Legal only in kernel mode (in the XKL-1 and the KS10, executive mode is kernel mode).
No	Legal nowhere (always executes as an MUUO).
–H	Legal where indicated by first symbol but causes a halt.
–RM	Legal where indicated by first symbol but causes the machine to halt and reload its microcode.

		XKL-1	*Extended KL10*	*Single–section KL10*	*KS10*	*KI10*	*KA10*
JRST 0,	JRST	Yes	Yes	Yes	Yes	Yes	Yes
JRST 1,	PORTAL	Yes	Yes	Yes	Yes	Yes	Yes
JRST 2,	JRSTF	Z	Z	Yes	Yes	Yes	Yes
JRST 3,		No	No	No	No	Yes	Yes
JRST 4,	HALT	K–H	K–H	K–H	K–H	K–H	IO–H
JRST 5,	XJRSTF	Yes	Yes	No	Yes	K–H	IO–H
JRST 6,	XJEN	IO	IO	No	K	K–H	IO–H
JRST 7,	XPCW	IO	IO	No	K	K–H	IO–H
JRST 10,		IO	IO	IO	IO	K	IO
JRST 11,	XJRSTP	K	No	No	No	K	IO
JRST 12,	JEN	Z ∧ IO*	Z ∧ IO*	IO	IO	K	IO
JRST 13,	HALTRM	K–RM	No	No	No	K	IO
JRST 14,	SFM	Yes	Yes	No	K	K–H	IO–H
JRST 15,	XJRST	Yes	Yes	Yes	No	K–H	IO–H
JRST 16,		No	No	No	No	K–H	IO–H
JRST 17,		No	No	No	No	K–H	IO–H

---

* JEN is legal only where IO is legal in section zero.

Any JRST function executed in a domain where the function is not legal is handled as an MUUO; see §2.16.

the most important 13–bit instruction codes (bits 0–12).

`JRST`	`JRST 0,`	Jump	25400
	`JRST 10,`	Jump and Restore Interrupt Level	25440
`HALT`	`JRST 4,`	Halt	25420
`JRSTF`	`JRST 2,`	Jump and Restore Flags	25410
`PORTAL`	`JRST 1,`	Allow Nonpublic Entry (KI10) Jump to User Program (KA10)	25404
`JEN`	`JRST 12,`	Jump and Enable	25450

JEN completes an interrupt by restoring the level and restoring the flags for the interrupted program. It is a combination of JRST 10, and JRSTF.

Caution

> Giving a JRSTF or JEN without indexing or indirect addressing restores the flags from the instruction code itself.

2.9.5 Subroutine Calling

Currently the stack instructions PUSHJ and POPJ, described in §2.10, are employed almost universally for handling subroutines. Described here are four traditional subroutine–handling instructions, the first two of which still enjoy some popularity.

JSR Jump to Subroutine

264	A	I	X	Y
0 8	9 12	13	14 17	18 35

A is not used.[47]

In section zero, save the program flags and PC in a PC word in location E; in a non–zero section, save PC in bits 6–35 of location E (clear bits 0–5). In either case jump to location $E + 1$.[48] The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared.

While the processor is in user mode, if this instruction is executed as an interrupt instruction (or by a KA10 MUUO), the processor leaves user mode, clearing Public. (An interrupt that is not dismissed automatically returns control to kernel mode.)

JSP Jump and Save PC

265	A	I	X	Y
0 8	9 12	13	14 17	18 35

---

[47] The A portion of this instruction is reserved and should be zero.

[48] Refer to the description of $E, E + 1$ on page 52.

In section zero, save the program flags and PC in a PC word in AC; in a non–zero section, save PC in AC bits 6–35 (clear bits 0–5). In either case jump to location *E*. The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared.

While the KI10 or KA10 is in user mode, if this instruction is executed as an interrupt instruction (or by a KA10 MUUO), the processor leaves user mode, clearing Public. (An interrupt that is not dismissed automatically returns control to kernel mode.)

When a subroutine is called in section zero by a JSR M, the typical method of returning from it is to give a JRSTF @M, which not only returns to the original program but also restores the original states of the program flags using the PC word saved by the JSR. In a non–zero section, there is an analogous procedure using a flag–PC double word. The subroutine is called by

```
SFM     M
JSR     M+1
```

and the return is made by XJRSTF M. A similar analogy holds for JSP. The following discussion of subroutine calling is geared to section zero. The application of these ideas to non–zero sections requires such substitutions as a flag–PC double word for a PC word, XJRSTF for JRSTF, and so forth.

JSR and JSP are unconditional, but the execution of such an instruction can be the result of a decision made by any conditional skip or jump. In the case of the flags, a basic overflow test and subroutine call can be made as follows.

```
JOV    .+2         ;jump over the next instruction if overflow is set
JRST   .+2         ;Overflow is clear, jump over the next instruction
JSR    OVRFLO      ;execute this instruction if Overflow is set
.
.
```

Because the No Divide flag is not among the flags that are testable by JFCL, to test for the No Divide condition, one must first read the flags into an accumulator. The following example shows how to call the DIVERR subroutine when No Divide is set:

```
JSP   T,.+1       ;Store flags but continue in sequence
TLNE  T,40        ;40 in the left half selects bit 12
JSR   DIVERR      ;execute this if No Divide is set.
.
.
```

A subroutine called by a JSR must have its entry point reserved for the PC word. Hence, it is non–reentrant: the JSR modifies memory so the subroutine cannot be shared with other processes. The JSP requires an accumulator, but it is faster and is convenient for argument passing. To return from a JSR–called subroutine, one usually addresses the PC word indirectly, returning to the location following the JSR. However, there are two ways to get back from a JSP. One can address the PC word indirectly with a `JRST @AC` (or `JRSTF @AC`, if the flags must be restored). Alternatively, one can return by addressing AC as an index register: `JRST (AC)`. By using the second return method, one can place N words of data for the subroutine immediately after the call and return to the location following the data by giving a JRST N(AC).

Suppose one wishes to call a print subroutine and supply the words from which the characters are to be taken. The main program would contain:

```
JSP   T,PRINT     ;Put PC word in accumulator T
.                 ;Text inserted here by ASCIZ pseudo-instruction,
.                 ;which automatically places a zero (null)
.                 ;character at the end
...               ;Next instruction here
```

The subroutine can use T as a byte pointer (§2.11) that already addresses the first word of data. For the print routine, characters are loaded into another accumulator CH:

```
PRINT:   HRLI    T,440700        ;Initialize left half of pointer for
                                 ;size 7, position 36
         ILDB    CH,T            ;Increment pointer and load byte
         JUMPE   CH,1(T)         ;Upon reaching zero character
                                 ;return to 1 beyond last data word
                                 ;Print routine
         .
         .
         .
         JRST    PRINT+1         ;Get next character
```

The next two instructions use a format that is incompatible with extended addressing. Because they are also considered an obsolete method for subroutine call/return (they have been supplanted by the stack instructions), no attempt has been made to find an alternate format for these instructions when executed in a non-zero section.

For compatibility with section–zero programs, these two instructions continue to work in non–zero sections. However, their use is restricted to intra–section operation, and all inter–section use is undefined.

JSA Jump and Save AC

266	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Save AC in location E, the in–section part of E in AC left, and the in–section part of PC in AC right. Then jump to location $E + 1$.[49] The original contents of E are lost.

---

[49] Refer to the description of $E, E + 1$ on page 52.

While the KA10 is in user mode, if this instruction is executed as an interrupt instruction or by an MUUO, the processor leaves user mode.

If E or $E + 1$ specifies a section other than the PC section, the effect of this instruction is indeterminate.

JRA **Jump and Restore AC**

267	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction effects the return from a subroutine called by a JSA instruction.

Place the contents of the location (in the PC section) addressed by AC left into AC and jump to the location (in the PC section) specified by E.

After the normal effective–address calculation is performed, the PC section is appended to the in–section address in AC left to form the address of where the old contents of AC were stored. The PC section replaces the section component of E, so the next instruction will be fetched from the same section as the current PC. Thus, all references made by this instruction are in the PC section.

Notes: The normal usage of JRA is of the form

```
JRA AC,(AC)     or     JRA AC, k(AC)
```

The first of these returns to the instruction following the JSA that called this subroutine. The second form skips k locations past the normal return point, thus avoiding argument words that may follow the JSA.

(The following paean to the virtues of JSA and JRA notwithstanding, PUSHJ and POPJ are now the most usual instructions for subroutine call and return.) A JSA combines advantages of the JSR and JSP. JSA does modify memory, but it saves PC in an accumulator without losing its previous contents (at the cost of not saving the flags). It is thus convenient for multiple–entry subroutines. In a subroutine called by a JSR, the returning JRST must refer to the (single) entry point. Since a JRA can retrieve the original PC by addressing AC as an index register, it is independent of any entry point without tying up an accumulator to the extent a JSP would. The accumulator contents saved by a JSA are restored by a JRA paired with it, despite intervening JSA—JRA pairs. Hence, these instructions are especially useful for nesting subroutines.

2.9.6 Overflow Trapping[50]

In the performance of a program, there are many events that cannot be foreseen and whose occurrence requires special action by the program. Among these events are arithmetic overflow and stack overflow.

Although there are instructions that test for such events, in a long string of computations, it would

---

[50] This feature is not available in the KA10. That processor is limited to the use of internal conditions that can act through the interrupt system (§4.3.5).

be both cumbersome and time consuming to test for overflow at every step. It is far better to allow an event such as overflow to cause a break from the normal program sequence. A “trap” is a break from the normal program sequence, attributed to a specific action of the program and synchronized with the execution of the program. (Contrast this to an “interrupt” which is an asynchronous break from the normal program sequence.)

Although traps are also used to handle the restrictions that play a role in program and memory management (as explained in later chapters), the present discussion is specifically concerned with the action by the processor in response to overflow.

2.9.6.1 Overflow Trapping in the KL10, KS10, and KI10 Processors

An instruction in which an arithmetic overflow condition occurs sets Overflow and Trap 1, and an instruction in which a stack overflow occurs sets Trap 2. Note that it is the overflow condition that sets Trap 1—not the state of the Overflow flag. Hence, an overflow is trapped even if Overflow is already set. Note also that the trap flags have no effect at all when paging is disabled. Otherwise, at the completion of an instruction in which either trap flag is set, rather than going on to the next instruction as specified by PC, the processor instead executes an instruction, the “trap instruction”, which is taken from a particular location in the process table for the program (user or executive). The location, as a function of the trap flag settings, is as follows:

Trap Flags Set	*Trap Type*	*Trap Number*	*Location*
Trap 1 only	Arithmetic overflow	1	421
Trap 2 only	Stack overflow	2	422
Trap 1 and 2	Not used by hardware[51]	3	423

A trap instruction is executed in the same address space and section as the instruction that caused it. When a trap condition occurs in a user instruction, the CPU refers to a location in the user process table, and any addresses used in the instruction in that location are interpreted in the user address space. Thus a user program can handle its own traps; e.g., by requesting that the Monitor to place a PUSHJ to a user routine in the trap location. An MUUO must be used in the trap location if the Monitor is to handle a user–caused trap.

The location of the instruction that caused the overflow can be determined from PC unless the instruction jumped, in which case its location is indeterminate. (However, the location of a PUSHJ can be determined from the data stored on the stack.) The trap instruction (either the instruction in the trap location or the final instruction in an XCT and/or LUUO string) clears the trap flags, so the processor returns to the trapped–from program unless the trap instruction changes PC. Thus, the trap instruction can be a no–op (which ignores the trap), a skip, a jump, or anything else. However, should the trap instruction itself set a trap flag (not necessarily the same one), a second trap occurs. An arithmetic instruction that overflows on every iteration produces an infinite loop if used as a trap instruction for arithmetic overflow. A stack instruction in a stack overflow trap can overflow only once. (The memory allocated to a stack should have at least one extra location to handle this case—two extra locations if the program and the trap both use the same stack pointer.)

An interrupt can occur between an instruction that overflows and the trap instruction, but the latter

---

[51] A trap can be produced artificially by simply setting up the trap flags from bits in a flag word. In this way the program can also use trap number 3, which at present cannot result from any hardware–detected condition and is reserved.

will be performed correctly upon the return provided the interrupt is dismissed automatically or the interrupt routine restores the flags properly. If a single instruction causes both overflow and a page failure, the latter has preference; the overflow trap will be taken care of after the trapped–from instruction has been restarted and completed successfully. A trap instruction that causes a page failure does not clear the trap flags; hence, after the page failure is taken care of, the trap instruction will correctly handle the trap when it is restarted.

2.9.6.2 Overflow Trapping in the XKL-1 Processor

An instruction in which an arithmetic overflow condition occurs sets Overflow and Trap 1. Note that it is the overflow condition that sets Trap 1—not the state of the Overflow flag. Hence an overflow is trapped even if Overflow is already set. An instruction in which a stack overflow occurs sets Trap 2.

Following an instruction that sets either trap flag, after handling any interrupts that are pending, the processor selects a trap vector in which to store information pertaining to the trap. (Traps from executive mode are disabled when the Executive Base Register is not valid; traps from user mode are disabled when the User Base Register is not valid. See §3.7.2.) If the selected trap vector is not enabled, the processor clears the trap flag(s) and continues with the next instruction as specified by PC. If the selected trap is enabled, the processor stores the PC and flags (with the trap flags clear) in the trap vector and takes a new PC and flags from the trap vector. The trap flags determine the location of the trap vector within the process table for the program (user or executive), as follows:

Trap Flags Set	*Trap Type*	*Trap Number*	*Location*
Trap 1 only	Arithmetic overflow	1	450 (`UP.TP1`)
Trap 2 only	Stack overflow	2	460 (`UP.TP2`)
Trap 1 and 2	Not used by hardware[52]	3	470 (`UP.TP3`)

The format of a trap vector is as depicted in Figure 2.2.

Following the instruction that sets the trap condition, the processor examines the state of the Trap Enable flag, bit 0 of word 0 in the trap block (T in the diagram, `UP%TEN==:1B0`, offset `UP.BFL==:0`). If the Trap Enable flag is 0, the trap is disabled. The processor clears the Trap1 and Trap2 flags; processing continues as specified by the present PC and flags.

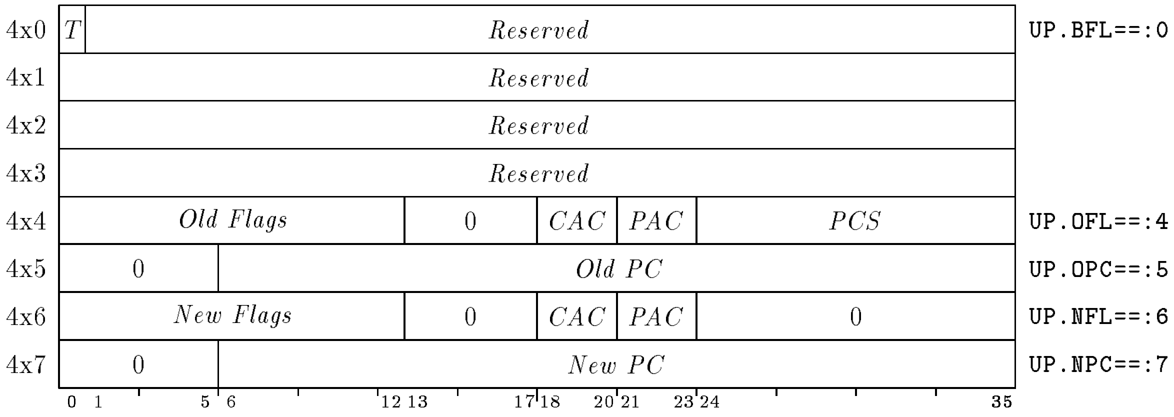
If the Trap Enable flag is 1, the processor stores the present Flags and Context in word 4 of the trap block (offset `UP.OFL`) and the present PC in word 5 (offset `UP.OPC`). The stored PC is generally the address following the trapping instruction; but, in the case of an instruction that jumps or skips, the stored PC reflects the effect of the jump or skip. In the stored flags, the Trap1 and Trap2 flags will be clear (to facilitate return to the trapped–from process).

The processor continues to handle the trap by loading new flags, context, and PC from the double word (at offsets `UP.NFL` and `UP.NPC`). For a trap that occurs while the processor is in user mode, the new flags may specify either an executive mode PC or a user mode PC. For a trap that occurs while the processor is in executive mode, the new flags specify an executive mode PC; if User is set in the new flags, it will be ignored. If the new flags specify an executive mode PC, then the current– and previous–context AC block selection will be set from bits 18–23 of the new flags word;

---

[52] A trap can be produced artificially simply by setting up the trap flags from bits in a flag word. In this way the program can also use trap number 3, which at present cannot result from any hardware–detected condition and is reserved.

Figure 2.2: XKL-1 Trap Vector



the previous–context section will be set according to the section of the stored PC. In all cases, execution continues at the location specified by the new PC and in the mode specified by the new flags.

Note: it is contemplated that the new PC will be in executive mode, because the user generally has no way to get at the information stored in the trap vector except by an operating system call (which would be more time–consuming than trapping through the executive to the user). The ability to specify a user mode address in the new PC is provided for special situations; e.g., real–time programming, for which arrangements could be made to make the user process table visible to the user program.

The location of the instruction that caused the overflow can be generally be determined from the stored PC unless the instruction jumped or skipped. Only in the case of PUSHJ can a jump instruction that causes a trap be located (by looking at the data on the stack).

If a single instruction causes both a trap and a page failure (e.g., a PUSHJ that causes stack overflow and which references a stack location that is not presently in memory), the page failure has precedence; the trap will be taken care of after the page failure has been resolved and the instruction has been completed.

2.10 Stack Operations

A stack, or pushdown list, is simply a set of consecutive memory locations from which words are read in the order opposite that in which they are written. In more general terms, it is any list in which the only item that can be removed at any given time is the last item in the list. This is usually referred to as “first in, last out” or “last in, first out.” Suppose locations a, b, c, ... are set aside for a stack. One can deposit data in a, b, c, d, then read d, then write in d and e, then read e, d, c, etc. Adding an item to the stack is referred to as “pushing” or “pushing down”; removing an item is “popping.” The stack is used in two ways: for handling data, and for saving and restoring PC, as in calling and returning from a subroutine.

The mechanism for keeping track of the list is a stack pointer, which specifies the position of the last item stored in the stack. This pointer is always kept in an accumulator. In section zero, the pointer has two parts: the right half contains the address of the last item, and the left half can contain a control count. An instruction that pushes an item onto the list increments both parts of the pointer by 1 and then places the item in the newly specified location; an instruction that pops an item takes it from the currently specified last position and then decrements both parts of the pointer by 1 so it points to what has become the last item. To help prevent mismanagement of the stack, the control count in the left half is monitored for overflow. The overflow condition, which sets the Trap 2 flag, is a change in the count from negative to zero on a push or from zero to negative on a pop. The KA10 lacks the trapping feature; in the KA10, stack overflow sets the Pushdown Overflow flag, which requests an interrupt on the level assigned to the processor (§4.3.6).

By keeping a control count in AC left, the programmer can set a limit to the size of the list by starting the count negative, or he can prevent the program from extracting more items than there are in the list by starting the count at zero, but he cannot do both at once. The common practice is to limit the size of the list. If only jump addresses are kept in the stack, the size limitation restricts the depth of nesting. A technique to catch extra popping of jump addresses is to put the address of an error routine at the bottom of the stack.

In a non–zero section there are two pointer formats: local and global. A local pointer is exactly like the one used in section zero, with the same manipulation in pushing and popping, except that the left half must be negative or zero (like a local index register). Restriction to a negative control count means that the control count can be used only to limit the size of the list, because the only meaningful overflow condition is the change to zero on a push. AC right contains a local address that is interpreted as being in the same section as the instruction. Note that a local stack wraps around in the local section (including the accumulators).

A global stack pointer is one in which bit 0 is zero and bits 6–35 contain a global address in a non–zero section; i.e., bits 6–17 are non–zero. Manipulation of a global pointer by pushing and popping is simply incrementing and decrementing the 30–bit address by 1; a global stack can therefore cross section boundaries. There is no control count, but the program can limit the stack size by making the pages at either end of the stack area inaccessible. Note that pushing on a local stack whose stack pointer has already overflowed (i.e., a stack pointer whose control count has become zero) changes the pointer to the global format, and it then addresses a location in section 1. Similarly, adjusting a global stack pointer into the "section" beyond 7777 changes it to the local format. (A pointer with a 0 in bit 0 and any arbitrary configuration in bits 1–5 is interpreted as local or global depending on whether or not bits 6–17 are zero.)

The processor provides five stack instructions for programs to use: two pairs for pushing and popping and one for making arbitrary adjustments of the pointer. One of the pairs handles data; the other pair are jumps that use the stack for handling subroutines.

PUSH Push

261	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the program is running in section zero, or AC left is negative (or AC bits 6–17 are zero), add 1 to each half of AC, then move the contents of location *E* to the location now addressed by AC right.

If the addition causes the count in AC left to reach zero, set Trap 2.[53] If the program is running in a non–zero section with a 0 in AC bit 0 and AC bits 6–17 non–zero, add 1 to AC, then move the contents of location E to the location now addressed by AC bits 6–35. The contents of E are unaffected; the original contents of the location added to the stack are lost.

Note: Do not allow the pointer to address AC, as the result of the instruction is then indeterminate.

POP Pop

262	A	I	X	Y
0 8	9 12	13	14 17	18 35

If the program is running in section zero, or AC left is negative (or AC bits 6–17 are zero), move the contents of the location addressed by AC right to location E, then subtract 1 from each half of AC. If the subtraction causes the count in AC left to reach -1, set Trap 2.[53] If the program is running in a non–zero section with a 0 in AC bit 0 and AC bits 6–17 non–zero, move the contents of the location addressed by AC bits 6–35 to location E, then subtract 1 from AC. The original contents of location E are lost.

Notes: Do not use the instruction POP AC,AC, because its result is indeterminate. To decrement the pointer by 1 position (in other words to throw away the last item in the stack), use either POP AC,(AC) or ADJSP AC,–1.

Example: In section zero, a POP can be used to implement a reverse BLT; i.e., to transfer a block of words from one area of memory to another, starting at the largest addresses and proceeding to the smallest. To move a block of N words from a source area to a destination area whose maximum addresses are S and D respectively, the program must first set up a stack pointer in accumulator T, where T left contains $N - 1 + 400000$ and T right contains S. The transfer is then effected by this pair of instructions:

```
POP       T,D-S(T)
JUMPL     T,.-1
```

where the jump returns to the POP until T left is less than 400000; i.e., until it looks positive. The 400000 added into T left prevents stack overflow but also limits the block to 2^{17} words.

PUSHJ Push and Jump

260	A	I	X	Y
0 8	9 12	13	14 17	18 35

Take one of these three courses of action.

If the program is running in section zero, add 1 to each half of AC, then save the program

---

[53] In the KA10, incrementing and decrementing both halves of AC together is effected by adding and subtracting 1000001_8. Hence a count of -2 in AC left is increased to 0 if $2^{18} - 1$ is incremented in AC right; conversely, 1 in AC left is decreased to -1 if 0 is decremented in AC right. In the KA10, there are no trap flags, so Pushdown Overflow (an APR interrupt condition) is set instead.

> flags and PC in a PC word in the location now addressed by AC right. If the addition causes the count in AC left to reach zero, set Trap 2.[53]
>
> If the program is running in a non–zero section but AC left is negative (or AC bits 6–17 are zero), add 1 to each half of AC, then save PC in bits 6–35 of the location now addressed by AC right (clear bits 0–5). If the addition causes the count in AC left to reach zero, set Trap 2.
>
> If the program is running in a non–zero section with a 0 in AC bit 0 and AC bits 6–17 non–zero, add 1 to AC, then save PC in bits 6–35 of the location now addressed by AC (clear bits 0–5).

Then jump to location E.

The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared. However, stack overflow overrides the Trap 2 clear, so if the list overflows, Trap 2 sets and the processor traps instead of jumping. The original contents of the location added to the list are lost.

While the KI10 or KA10 is in user mode, if this instruction is executed as an interrupt instruction (or by a KA10 MUUO), the processor leaves user mode, clearing Public. (An interrupt that is not dismissed automatically returns control to kernel mode.)

POPJ Pop and Jump

263	A	I	X	Y
0 8	9 12	13	14 17	18 35

E is not used.[54]

Take one of these three courses of action.

> If the program is running in section zero, subtract 1 from each half of AC. If the subtraction causes the count in AC left to reach -1, set Trap 2.[53] Then jump to the location addressed by the right half of the location that was addressed by AC right *prior* to the decrementing.
>
> If the program is running in a non–zero section but AC left is negative (or AC bits 6–17 are zero), subtract 1 from each half of AC. If the subtraction causes the count in AC left to reach -1, set Trap 2. Then jump to the location addressed by bits 6–35 of the location that was addressed by AC right *prior* to the decrementing.
>
> If the program is running in a non–zero section with a 0 in AC bit 0 and AC bits 6–17 non–zero, subtract 1 from AC, and jump to the location addressed by bits 6–35 of the location that was addressed by AC bits 6–35 *prior* to the decrementing.

---

[54] I, X and Y are reserved and should be zero.

Caution

> The jump is completed before the processor responds to stack overflow. Hence, it is impossible to determine the location of the POPJ that caused the overflow.

ADJSP Adjust Stack Pointer[55]

105	A	I	X	Y
0 8	9 12	13	14 17	18 35

If the program is running in section zero, or AC left is negative (or AC bits 6–17 are zero), add E_{R} (the in–section part of E, bit 18 is the sign) algebraically to each half of AC. If a negative E_{R} changes the count in AC left from positive or zero to negative, or if a positive E_{R} changes the count from negative to positive or zero, set Trap 2. If the program is running in a non–zero section with a 0 in AC bit 0 and AC bits 6–17 non–zero, add E_{R} (with bit 18 extended into bits 0–17) algebraically to AC.

Notes: When an ADJSP changes the control count in a local pointer in a non–zero section from negative to positive, the result will appear to be a global pointer. Similarly, an overflow to negative can occur only from zero, as otherwise the original would have been taken as global (excluding the irrelevant case of AC left being greater than zero only because of bits 1–5 being non–zero).

A stack is very convenient for a program that can use data stored in this manner because the pointer is initialized only once and only one accumulator is required for the most complex stack operations. To initialize a local pointer P for a list to be kept in a block of memory beginning at BLIST and to contain at most N items, the following suffices.

```
MOVSI   P,−N
HRRI    P,BLIST-1
```

The programmer must define BLIST and set aside locations BLIST to BLIST+$N-1$. Using MACRO to full advantage one could instead give

```
MOVE    P,[IOWD N,BLIST]
```

where the pseudo–instruction

```
IOWD J,K
```

is replaced by a word containing $-J$ in the left half and $K-1$ in the right. Elsewhere there would

---

[55] In the KI10 and KA10, this instruction traps as an unassigned code (§2.16).

appear

```
BLIST:  BLOCK    N
```

which defines BLIST as the current contents of the location counter and sets aside the N locations beginning at that point.

Since the stack is independent of the subroutine called, PUSHJ–POPJ can be used for multiple entries. Moreover, ordering by level is inherent in the structure of a stack, so paired PUSHJ–POPJ instructions are excellent for nesting subroutines: there can be any number of subroutines at any level, each with more subroutines nested within it. Recursive subroutines are also easily programmed.

The stack instructions tie up an accumulator, but the usual procedure is to keep both data and jump addresses in a single list so only one accumulator is required for most operations. The programmer must keep track of whether a given entry in the list is data or a saved PC; in other words, generally every item inserted by a PUSH should be removed by a POP or ADJSP and every PUSHJ should be matched by a POPJ.

If flag restoration is desired in section zero, the returning

```
        POPJ    P,
```

can be replaced by

```
        POP     P,AC
        JRSTF   (AC)
```

which requires another accumulator.

In section zero only, if the flags are not important, data may be stored in the left halves of the PC words in the stack, reducing the required pushdown depth.

The stack is kept in a random–access memory, so the restrictions on order of entry and removal of items apply only to the standard addressing by the pointer in stack instructions—other addressing methods can reference any item at any time. The most convenient way to do this is to use the address part of the pointer as an index. To move the last entry to accumulator AC, one need simply give

```
        MOVE   AC,(P)
```

This does not shorten the list—the word moved remains the last item in it.

One usually regards an index register as supplying an additive factor for a basic address contained in an instruction word, but the index register can supply the basic address and the instruction can supply the additive factor. Thus one can retrieve the next–to–last item by giving

```
        MOVE    AC,-1(P)
```

and so forth. Similarly

```
        PUSH    P,-3(P)
```

appends the third–to–last item to the end of the list (remember that E is calculated before the contents of P are changed).

```
        POP     P,-2(P)
```

removes the last item and inserts it in place of the next–to–last item in the shortened list.

An ADJSP can delete an entire block from a stack; and, in combination with a BLT, it can be used to add a whole block.

It is not very practical to use PUSHJ to call a subroutine in section zero from a non–zero section. The PUSHJ, executed in a non–zero section, will store a 30–bit PC on the stack; a corresponding POPJ, executed in section zero, interprets only the rightmost 18–bits of PC when returning from the subroutine. Further, if the stack is local, it is impractical to make inter–section calls; if the stack is global, it will be misinterpreted while the program executes in section zero. For these reasons, programs that use extended addressing tend to avoid using section zero.

2.11 Byte Manipulation[56]

This set of six instructions allows the programmer to pack or unpack bytes of any length anywhere within a word. Movement of a byte is always between AC and a memory location: a deposit instruction takes a byte from the right end of AC and inserts it at any desired position in the memory location; a load instruction takes a byte from any position in the memory location and places it right–justified in AC.

The byte manipulation instructions have the standard memory reference format, but the effective–address E is used to retrieve a pointer, which is used in turn to locate the byte or the place that will receive it. There are three formats for byte pointers: one–word local, one–word global, and two–word. Only the first of these applies in unextended processors. The first two apply in section zero of an extended processor.[57] All three formats are valid in non–zero sections.

A one–word local pointer has the format

---

[56] In a KA10 without byte manipulation hardware, all of the instructions presented in this section trap as unassigned codes (§2.16). However, all KA10s were delivered with byte manipulation hardware.

[57] In early versions of the extended KL10 microcode, one–word global byte pointers were legal only in non–zero sections. Current KL10 microcode removes this restriction, except, the KL10 fails to honor one–word global byte pointers when they appear in string instructions executed in section zero.

P	*S*	0	*I*	*X*	*Y*
0 5	6 11	12	13	14 17	18 35

where S is the size of the byte as a number of bits (with zero S specifying a null byte) and P is its position as the number of bits remaining at the right of the byte in the word (e.g., if P is 3 the rightmost bit of the byte is bit 32 of the word). The rest of the pointer is interpreted in the same way as in an instruction: I, X, and Y are used to calculate the address of the location that is the source or destination of the byte; the address calculation begins in the section containing the pointer.

Unextended processors support only the one–word local–format byte pointer. Unextended processors ignore bit 12 of the byte pointer; for compatibility with extended processors, bit 12 should be set to zero.

In an extended processor, the P field of a one–word local byte pointer must contain a value $\leq 36_{10}$ ($\leq 44_8$). In an extended processor, bit 12 of a one–word local byte pointer must be zero when the byte pointer is used in a non–zero section. Bit 12 is ignored by the processor when executing in section zero; for compatibility with extended uses, bit 12 should be set to zero.

The one-word global byte pointer, valid only in extended processors, has this format:

PS	*Y*
0 5	6 35

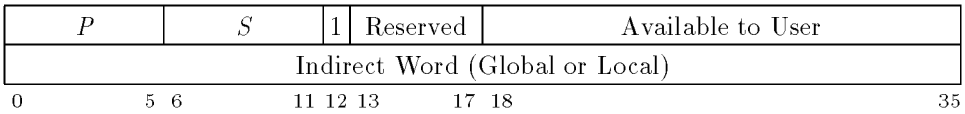
In this format, the Y field contains the 30–bit address of the word containing the byte (or into which the byte will be placed); the PS field, which must contain a value $> 36_{10}$ ($> 44_8$), encodes both the position and the size of the byte. These encodings are displayed in the following table, which gives the P and S values corresponding to each value of PS. The value of PS is expressed in octal. the value of S, the size of the byte, and the value of P (the number of bits remaining at the right of the byte in the word) are expressed in decimal.

PS	*P*	*S*	*PS*	*P*	*S*	*PS*	*P*	*S*
45	36	6	56	20	8	67	36	9
46	30	6	57	12	8	70	27	9
47	24	6	60	4	8	71	18	9
50	18	6	61	36	7	72	9	9
51	12	6	62	29	7	73	0	9
52	6	6	63	22	7	74	36	18
53	0	6	64	15	7	75	18	18
54	36	8	65	8	7	76	0	18
55	28	8	66	1	7	77	Illegal	

The two–word byte pointer, valid only in non–zero sections, in locations $E, E+1$,[58] has this format:

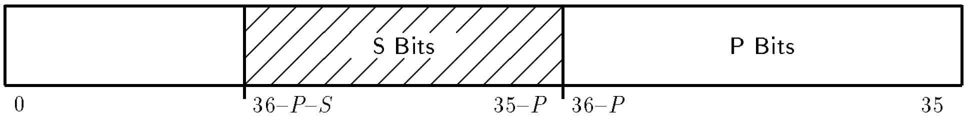
---

[58] Refer to the description of $E,E+1$ on page 52.



This two–word arrangement allows for global pointing, because the second word can be local or global as specified by bit 0 (see the discussion of indirect words in §1.7.2). An extended processor determines the number of words in a pointer by the state of bit 12 in the first word of byte pointers read from non-zero sections. (The processor ignores bit 12 of byte pointers read from section zero; however, bit 12 should be 0.)

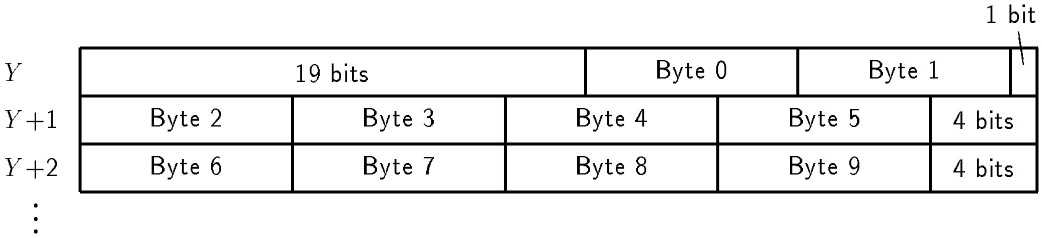
Any of the three types of byte pointers aim at a word whose format is



where the shaded area is the byte.

Bytes are always contiguous within a word, and the forward order is left to right in words and from low to high addresses. The position of the byte area in a word is called the "byte alignment." Let P be the position of a specified byte; $36 - P$ is then the number of bits in the left part of the word including the given byte and all byte positions at the left of it. Dividing $36 - P$ by S gives the number of byte positions in this left part, and the remainder is those extra bits at the left end that are not in any byte position. This number of extra bits, $(36 - P) \bmod S$, is the byte alignment.

A block of 8–bit bytes might look like this.



In the first word, the first byte can occupy any position, and as many bytes as will fit are packed into the rest of the word at the right. In the second and all succeeding words, the byte alignment is zero no matter where the bytes may start in the first word, and as many as will fit are packed into every word, although the last may run short. In this example the byte alignment in the first word is 3, even though two byte positions are not used: the alignment is always less than S and is the number mod S of bits at the left of the first byte. Bytes are assumed to be handled consecutively in the forward direction only, and for this type of processing the pointer is "incremented." Since bytes are contiguous and are processed from left to right, incrementing merely replaces the current value of P by $P - S$, unless there is insufficient space in the present location for another byte of the specified size ($P - S < 0$). In this case, Y is increased by 1 to point to the next consecutive location, and P is set to $36 - S$ to point to the first byte at the left in the new location.[59]

To facilitate processing a series of bytes, two of the byte handling instructions increment the pointer before handling the byte. A typical procedure for using these instructions is to set up the pointer initially to point at the byte position preceding the first byte.

The pointer is referred to as being "at location E," which means that it is either a single word in location E or a double word in location $E, E + 1$.[58] Local and global pointers and the operations associated with them, as described above, are also utilized in handling byte strings, which are discussed in the three sections following this one.

CAUTION

> On an extended processor, a pointer with P greater than 36_{10} signifies a one–word global byte pointer in any context; an instruction that uses a byte pointer in which P is equal to 63 will trap as an MUUO. On an unextended processor, giving a byte pointer in which P is greater than 36 produces an indeterminate result in any instruction that uses it.
>
> Giving a byte pointer in which S is greater than 36 produces an indeterminate result in any instruction that uses it. A P of 36 should be used only for initial incrementing by an ILDB or IDPB (its effect on an LDB or DPB is indeterminate).
>
> If both P and S are less than 36 but $P + S > 36$, a byte of size $36 - P$ is loaded from position P, or the right $36 - P$ bits of the byte are deposited in position P.

LDB **Load Byte**

135	A	I	X	Y
0 8	9 12	13	14 17	18 35

Retrieve a byte of S bits from the location and position specified by the pointer at location E, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected; the original contents of AC are lost.

Note: If S is zero, LDB clears AC.

DPB **Deposit Byte**

137	A	I	X	Y
0 8	9 12	13	14 17	18 35

Deposit the right S bits of AC into the location and position specified by the pointer at location E. The original contents of the bits that receive the byte are lost; AC and the remaining bits of the deposit location are unaffected.

---

[59] *Caution*: In the KA10, do not allow Y to reach maximum value. The whole pointer is incremented, so, if Y is $2^{18} - 1$, it becomes zero and X is also incremented. The address calculation for the pointer uses the original X; but, if an interrupt should occur before the calculation is complete, the incremented X is used when the instruction is repeated.

IBP Increment Byte Pointer

133	00	I	X	Y
0 8	9 12	13	14 17	18 35

Increment the byte pointer at location E, setting the byte alignment to zero if the incrementing crosses a word boundary, as explained above.

Note: Giving this instruction code with bits 9–12 non–zero produces the **ADJBP** instruction described at the end of this section. In the KI10 and KA10, only the **IBP** form is available and bits 9–12 are ignored (but should be zero).

ILDB Increment Pointer and Load Byte

134	A	I	X	Y
0 8	9 12	13	14 17	18 35

Increment the byte pointer at location E, setting the byte alignment to zero if the incrementing crosses a word boundary, as explained above. Then retrieve a byte of S bits from the location and position specified by the newly incremented pointer, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected; the original contents of AC are lost.

IDPB Increment Pointer and Deposit Byte

136	A	I	X	Y
0 8	9 12	13	14 17	18 35

Increment the byte pointer at location E, setting the byte alignment to zero if the incrementing crosses a word boundary, as explained above. Then deposit the right S bits of AC into the location and position specified by the newly incremented pointer. The original contents of the bits that receive the byte are lost; AC and the remaining bits of the deposit location are unaffected.

Note that, in the pair of instructions that both increment the pointer and process a byte, it is the *modified* pointer that determines the byte location and position. Hence, to unpack bytes from a block of memory, the program should set up the pointer to point to a byte just *before* the first desired, and then load the bytes with a loop containing an **ILDB**. If the first byte is at the left end of a word, this is most easily done by initializing the pointer with a P of 36 (44_8). Incrementing then replaces the 36 with $36 - S$ to point to the first byte. For the convenience of the programmer, MACRO has a pseudoinstruction for setting up such a pointer: in assembly language,

`POINT` S, Y

is replaced by a pointer that points to a byte of size S at position 36 in location Y. At any time that the program might inspect the pointer during execution of a series of **ILDB**s or **IDPB**s, it points to the last byte processed (this may not be true when the pointer is tested from an interrupt routine).

ADJBP Adjust Byte Pointer[60]

133	A	I	X	Y	$A \neq 0$
0 8	9 12	13	14 17	18 35	

Take one of these three courses of action depending on the value of S in the pointer at location E.

> If S is 0, place an unmodified copy of the pointer in AC or AC,AC+1.[61]
>
> If S is greater than 36 minus the byte alignment given by the pointer—so not even one byte will fit in a word—set Trap 1, Overflow, and No Divide and go on to the next instruction without affecting the ACs or memory.
>
> If S is greater than 0 but less than 36 minus the byte alignment, make a copy of the pointer from location E or $E, E+1$ and "adjust" the copy, forward or backward, by the number of byte positions specified by AC, *preserving the byte alignment across word boundaries*. If AC contains a positive number N, adjust the copy by N bytes forward; if AC contains a negative number $-N$, adjust the copy by N bytes backward. Place the revised pointer copy in AC or AC,AC+1 as appropriate. The original pointer is unaffected; the original contents of AC or AC,AC+1 are lost.[62]

Notes: The adjustment always produces a pointer that specifies an actual byte. For example, adjusting a pointer with a P of 36 by zero bytes results in a pointer that specifies the rightmost byte (of appropriate alignment) in the preceding word. When the pointer specifies a byte alignment of zero, there is no difference between "adjusting" it by N and "incrementing" it N times (except that the latter actually modifies the pointer). Since the result goes to AC, it is not generally useful to adjust a local pointer that is in a different section from the instruction.

Giving this instruction code with a zero A field or in a KI10 or KA10 produces the IBP instruction described above. Note that, if $S = 0$, this instruction is equivalent to MOVE.

The ADJBP instruction facilitates selection of individual bytes at arbitrary positions in an array whose format differs from the linear format used by the incrementing instructions, in that the adjustment preserves the byte alignment across word boundaries. As an example of this format, let us again use 8–bit bytes where the pointer specifies a byte in the same position as byte 0 in our linear example at the beginning of this section. Such an array would look like this.

---

[60] In the KA10 and KI10, this instruction is not implemented; it is performed as IBP.

[61] As of KL10 microcode 2.1[442], this case results in the processor setting Trap 1, Overflow, and No Divide and going on to the next instruction without affecting the ACs or memory.

[62] The KL10 does not provide a correct result when AC initially contains 400000000000.

	3 bits					1 bit
⋮						
Y-2		Byte −10	Byte −9	Byte −8	Byte −7	
Y-1		Byte −6	Byte −6	Byte −4	Byte −3	
Y		Byte −2	Byte −1	Byte 0	Byte 1	
Y+1		Byte 2	Byte 3	Byte 4	Byte 5	
Y+2		Byte 6	Byte 7	Byte 8	Byte 9	
⋮						

Here the bytes are ordered in either direction from the zero position, and the byte alignment specified by the pointer is preserved throughout all words in the block. Within the restriction that the alignment be preserved, as many bytes as will fit are packed in all words. (Except, the first and last words of the block need not be filled.) For example, with 10–bit bytes there are always three per interior word in the linear format, but in the array format with an alignment of 8, there are only two, occupying bits 8–17 and 18–27. Specification of an arbitrary byte anywhere in the array is accomplished by using an ADJBP. The microcode makes the adjustment by changing Y to the location containing the byte and then setting up a new P for the specific byte.

Suppose bytes are packed five to a word, a pointer at location E now points to the third byte in a given location, and one wishes to retrieve the thirty–first (the fourth byte from the sixth location) beyond that. This routine loads the desired byte into AC.

```
MOVEI   AC,37     ;Adjust by $31_{10}$
ADJBP   AC,E      ;Form the adjusted pointer in AC
LDB     AC,AC     ;now, get the byte itself
```

2.12 String Manipulation[63]

This section and two sections that follow treat the instructions that handle strings. All string instructions are in the extended instruction set, and all therefore have a two–word format, the first word being EXTEND. The second instruction word, whose own effective–address is *E1*, is at location *E0*, which is the effective–address of the EXTEND. An instruction that "offsets" uses *E1* as a signed offset, in which bit 18 is the sign. An instruction that "translates" or "edits" makes use of a translation table that begins at *E1*.

A string is a sequence of bytes as specified by successive states of a standard byte pointer of the type described in the preceding section, the first page or so of which the reader should reread if he does not remember in detail the format of the pointer, the way it is incremented, and the way bytes are organized in consecutive words (specifically with zero byte alignment). The program defines a string by giving its length in number of bytes and an initial value for the pointer. Initially the

---

[63] In the KI10 and KA10, these instructions trap as unassigned codes (§2.16).

pointer must point to the byte position preceding the first byte in the string, because every string instruction acts in a manner similar to a series of ILDBs or IDPBs, or in some cases both. Hence, all string operations are from left to right because of the way byte pointers are incremented. A string byte pointer and length may define a string of bytes or define a string space that will receive bytes. In an instruction that moves a string, the actual string moved is referred to as the source string, and the receiving space is referred to as the destination string, even though initially the latter is a string of positions rather than bytes. Note that source and destination strings need not be the same length. When the source string is longer, only part of it will fit in the destination space. Conversely, when the source is shorter, it can be inserted into part of the destination space, either starting at the left (left justified) or placed so that its final byte is in the last destination position (right justified).

Bytes may be of any size from zero bits to thirty–six. However, in a given string, all bytes are the same size, as specified by the pointer. The relationship between source and destination byte sizes is a function of the way the programmer uses his data and the meaning he assigns to it. Depending on circumstances, it may be desirable to spread out a source string into a destination space whose positions are larger than the source bytes (data is always right justified in a given byte position); or source bytes may be truncated to fit into smaller destination positions (the truncation being always from the left).

Most string operations make some use of bytes other than those in the strings themselves. Such bytes may be special characters found in locations $E0{+}1$ and $E0{+}2$ or substitutions supplied by a translation table. A byte from any location not in a string defined by the pointers and lengths associated with the instruction is always from the right end of the word or half–word containing it and has the same number of bits as the bytes in the string in which it will be used.

The "interior" of a string space is all of those bits in the words containing the string that lie between the first byte in the first word and the last byte in the last word. Since byte alignment is zero, the string is packed solid (with no unused interior bits) if 36 is an integral multiple of the byte size. For sizes that do not pack solid, there will be unused interior bits except in the last word, and they will lie at the right of the bytes in the words. In a destination string space of a string instruction, if all unused interior bits are 0s initially, they are guaranteed to be 0s at the completion of the instruction. If such bits are not all 0s initially, the subsequent states of unused interior destination bits are indeterminate. In other words, the implementation is free to use a mechanism other than repeated IDPB instructions to store into the destination string. (Source strings are unaffected by the instructions.)

Bytes in a string may represent anything—digits, letters, or special characters. This section discusses the basic operations: those that compare two strings or that move a string to a new position with optional offsetting or translating of its bytes. The next section covers special operations for converting between binary and decimal, where a decimal number is a string of bytes representing decimal digits. §2.14 considers an instruction that is, effectively, a whole routine for complex editing of a text string.

All string instructions skip the next instruction in the PC sequence if all operations are carried out as expected or a compare condition is satisfied, etc. Failure of a compare condition to be satisfied or something being amiss (such as loss of bytes because the source string will not fit in the destination space) usually causes the processor to perform the next instruction. Note that the "next instruction" is relative to the EXTEND (or an XCT that executes it)—in other words, relative to the actual instruction to which PC points. The location of the second instruction word, which is actually the operand of the EXTEND, does not affect the PC value.

Every string instruction uses a block of accumulators which contain one or two byte pointers. A pointer may be one word or two (local or global), as explained at the beginning of §2.11. In the illustrations of the AC block format for the EXTEND instructions, pointers are always shown as a pair of words in AC+N,AC+N+1, where the actual byte pointer used may be in the first accumulator or in both. The reader should note that, when a pointer is one word, the instruction does not in any way affect the contents of the second accumulator in the pair.[64]

CAUTION

> For the instructions described in this and the next two sections, the format illustrations show various parts of the accumulators and instruction words as being zero. These parts are reserved and *must be zero*. Failure to comply with this requirement will cause an EXTEND instruction to give an indeterminate result.
>
> Moreover, there can be no overlapping of the various quantities used in any EXTEND instruction. The source and destination spaces must never overlap; and under no circumstances should any string overlap anything else used by the instruction, such as the AC block, a translation table, an edit pattern, special character locations following *E0*, or even the instruction words themselves; and unused ACs in the specified block (such as that following a one-word byte pointer) cannot be used for any other purpose (such as an index register). Any such overlapping will cause the result of the instruction to be indeterminate.
>
> This caution applies not only to the basic instructions discussed here but also to those of the two sections that follow.

There are four string move instructions. One right–justifies the source string in the destination space, without otherwise modifying it. The others move the source string directly (i.e., left justified), with the bytes unmodified, all offset by a constant, or translated where every byte of a given value is replaced by a corresponding substitution. The six string compare instructions do not affect the specified strings; instead, the strings are compared according to a collating sequence based on the algebraic relationships of their bytes taken as unsigned binary numbers. All of these are two–word instructions, where the first has the EXTEND code 123 and all use a block of six accumulators.

MOVSLJ Move String Left Justified

123	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

E0	016	00	*I*	*X*	*Y*	Bits 9–12 = 0
E0+1					Fill	*E1* is not used.[65]
	0 8	9 12	13	14 17	18 35	

---

[64] However, in the KL10, when a one–word global byte pointer is used in a string instruction, the processor converts it to an equivalent two–word byte pointer; the two–word byte pointer is returned in the AC block upon completion of the instruction. For this reason, the KL10 fails to honor one–word global byte pointers when they appear in string instructions executed in section zero.

Other processors may deal with the one–word global byte pointer without conversion.

[65] I, X, and Y are reserved and should be zero.

Move the source string left–justified into the destination string space.

Source and destination are defined by the contents of a block of six accumulators.

	0 8	9 35	
AC	000	Source String Length	Bits 0–8 = 0
AC+1 AC+2	Source String Byte Pointer		
AC+3	000	Destination String Length	Bits 0–8 = 0
AC+4 AC+5	Destination String Byte Pointer		

Beginning at the left, copy as many bytes from the source string as will fit into the destination string space. If any source bytes are left over (i.e., if the source string is longer than the destination string), go to the next instruction. Otherwise, place the fill character from *E0*+1 in the remaining destination byte positions (if any) and skip the next instruction.

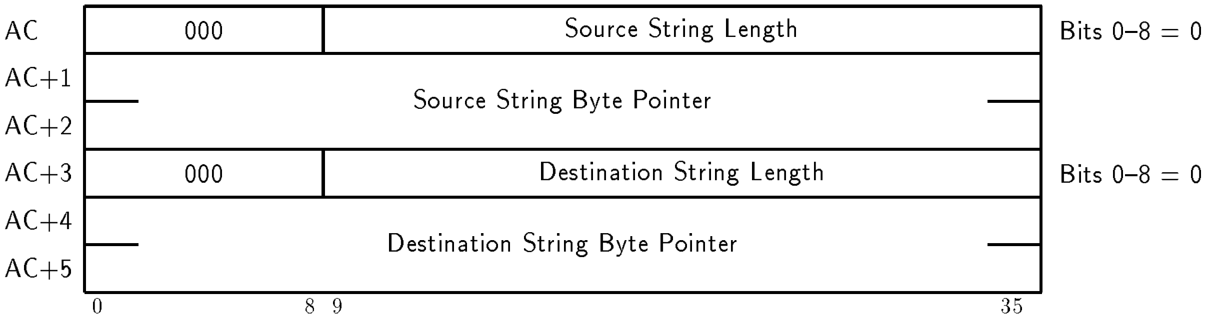
At the end, the byte pointers point to the last positions referenced in source and destination, AC+3 contains zero, and AC bits 9–35 contain the number of source bytes not copied (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise, unused interior destination bits are indeterminate. The source string is unaffected.

MOVSO Move String Offset

0 8	9 12	13	14 17	18 35
123	*A*	*I*	*X*	*Y*

	0 8	9 12	13	14 17	18 35	
E0	014	00	*I*	*X*	*Y*	Bits 9–12 = 0
E0+1					Fill	

Move the source string, with each byte offset by *E1*, left justified into the destination string space. Source and destination are defined by the contents of a block of six accumulators.



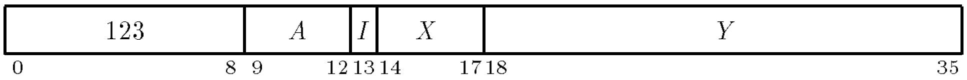
Beginning at the left, read each byte from the source string, add *E1* to it algebraically (bit 18 is the sign), and place the offset byte in the corresponding position in the destination string space, provided it is not larger than the specified byte size (i.e., there are no 1s outside the area containing the offset byte in the register holding it). Continue in this fashion for each source byte until an oversize offset byte is encountered or until either the source string or the destination space is exhausted, whichever occurs first. Then, if there are any source bytes not moved (because an offset byte is oversize or the source string is too long), go to the next instruction. Otherwise, place the fill character from *E0*+1 in the remaining destination byte positions (if any) and skip the next instruction.

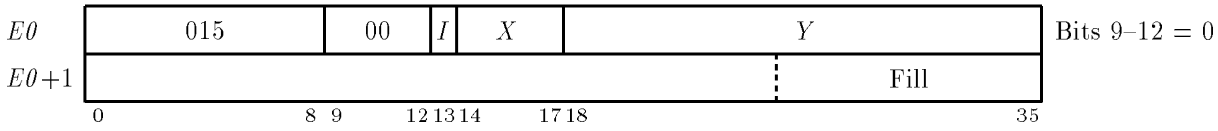
At the end, the byte pointers point to the last positions referenced in source and destination,[66] AC bits 9–35 contain the number of source bytes not moved (if any), and AC+3 bits 9–35 contain the number of destination byte positions not used (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise, unused interior destination bits are indeterminate. The source string is unaffected.

Note: MOVSO with a zero offset is similar to MOVSLJ, but the latter is preferred in applicable situations because it is faster.

Offset can be used to change a string of capitals to lower case by adding 40 octal to every byte.

MOVST Move String Translated

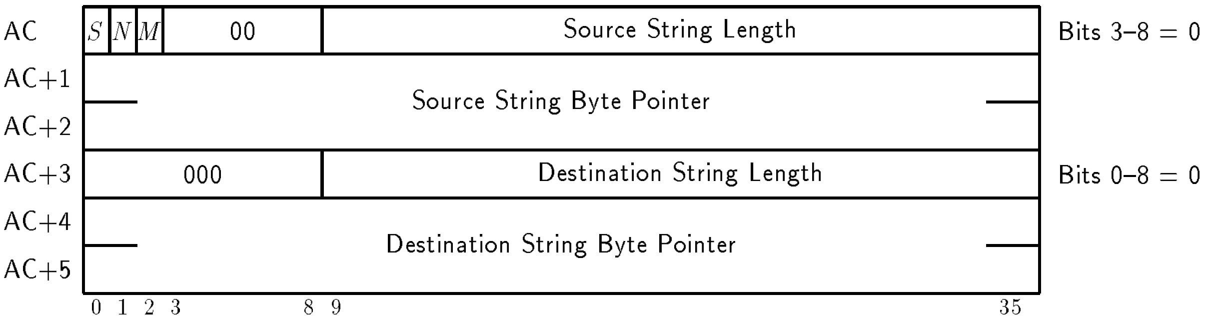




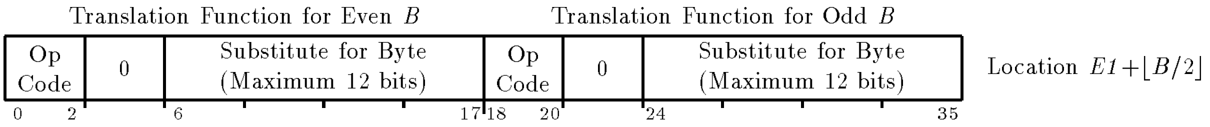
Move the significant part of the source string, with its bytes replaced by bytes from a translation table at *E1*, left justified into the destination string space. Source and destination are defined by the contents of a block of six accumulators. *S* is the significance bit: setting it signals the start of that part of the source string that has significance, and bytes read while it is on are regarded as significant.

---

[66] If the instruction terminates because of a condition relating to source data (e.g., an oversize offset byte in MOVSO, a termination byte in MOVST) then the source pointer addresses the source byte that caused the termination and the destination pointer addresses the last byte actually stored in the destination field.



Beginning at the left, read each byte from the source string and carry out the corresponding translation function given in the appropriate half—word at location $E1 + \lfloor B/2 \rfloor$ in the translation table, where B is the value of the source byte.[67] Each word in the table has this format:



Perform the function specified by the op code in the half-word corresponding to the source byte, as follows.

0 If S is 1, take the substitute in place of the source byte.

1 Terminate translation.

2 If S is 1, take the substitute in place of the source byte. (Also clear M.)

3 If S is 1, take the substitute in place of the source byte. (Also set M.)

4 Set S and take the substitute in place of the source byte. (Also set N.)

5 Terminate translation. (Also set N.)

6 Set S and take the substitute in place of the source byte. (Also set N and clear M.)

7 Set S and take the substitute in place of the source byte. (Also set N and M.)

Then take one of these three courses of action:

> If the function makes no substitution and does not terminate, read the next byte from the source string and continue as described above.
>
> If the function makes a substitution, place the substituted byte in the next position in the destination string space, read the next byte from the source string, and continue as described above.
>
> If the function terminates the translation, go on to the next instruction.

---

[67] The notation $\lfloor x \rfloor$ signifies the largest integer contained within x.

Unless the translation is terminated by a translation function, continue the above procedure until either all source bytes are processed or the destination string is filled, whichever occurs first. Then, if any source bytes are left over, go to the next instruction. Otherwise, place the fill character from *E0*+1 in the remaining destination byte positions (if any) and skip the next instruction.

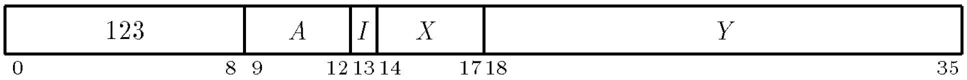
At the end, the byte pointers point to the last positions referenced in source and destination,[66] AC bits 9–35 contain the number of unprocessed bytes in the source string (if any), and AC+3 bits 9–35 contain the number of destination byte positions not used (if any). If unused interior bits in both strings are clear initially, they are left clear; otherwise, unused interior destination bits are indeterminate. The source string is unaffected.

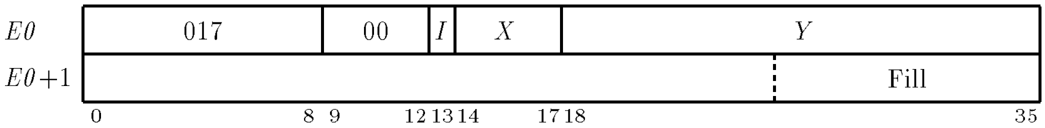
Notes: The translation table starts at location *E1* and, since there are two functions per word, it contains 2^{n-1} locations, where n is the number of bits in a byte. The address is generated by adding the left $n - 1$ bits of a byte to *E1*.

Of the three flags in AC bits 0–2, only S is relevant to this instruction; the translation functions also manipulate M and N, but their states have no effect on the result. S being set means the translation has started. The programmer can make the translation start at the beginning of a string by having S already set when the instruction is given or he can skip any number of initial bytes in the source string and have the translation started by the first occurrence of some byte whose associated function sets S. Hence, by use of S and terminating functions, the programmer can have an MOVST translate any contiguous subset of the source string.

Text in upper case and lower case can be converted to all upper case by an MOVST with a translation table that substitutes capitals for both.

MOVSRJ Move String Right Justified

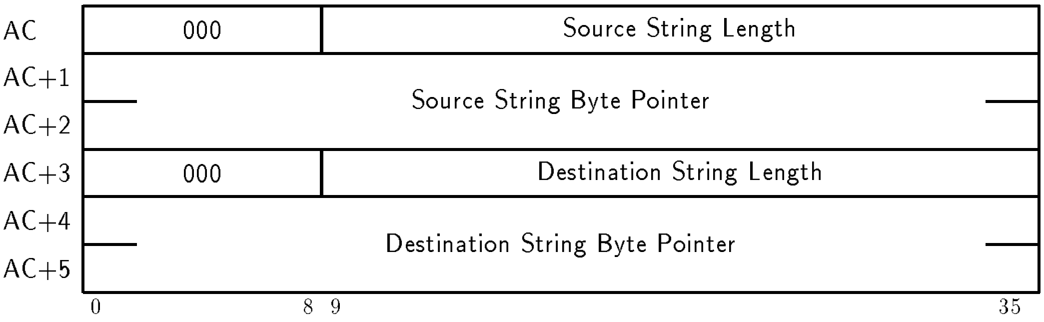




Bits 9–12 = 0

E1 is not used.[65]

Move the source string right justified into the destination string space. Source and destination are defined by the contents of a block of six accumulators.



Bits 0–8 = 0

Bits 0–8 = 0

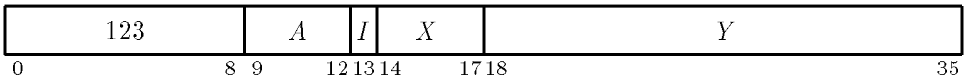
Check the relation between the source and destination lengths to select one of the following three courses of action:

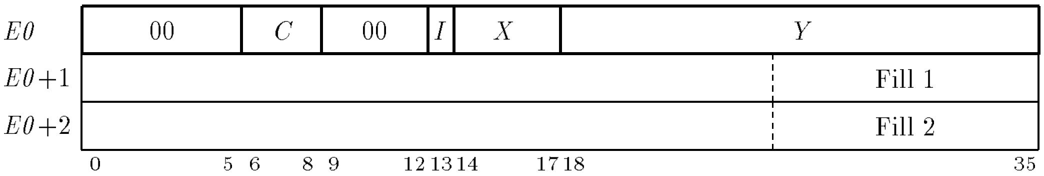
> If the source and destination strings are the same length, move the source string into the destination space.
>
> If the source string is shorter, place the fill character from *E0*+1 in destination byte positions beginning at the left until there are just enough places remaining in the destination space to accept the source string. Move the source string into the remaining destination positions at the right.
>
> If the source string is longer, skip over enough source bytes at the left so the remaining source substring will fit in the destination space. Move the remaining source bytes into the destination space.

After completing the selected course of action, skip the next instruction.

At the end, the byte pointers point to the last positions referenced in source and destination, AC+3 contains zero, and AC bits 9–35 contain the number of source bytes skipped over (if any).[68] If unused interior bits in both strings are clear initially, they are left clear; otherwise, unused interior destination bits are indeterminate. The source string is unaffected.

CMPS– Compare Strings



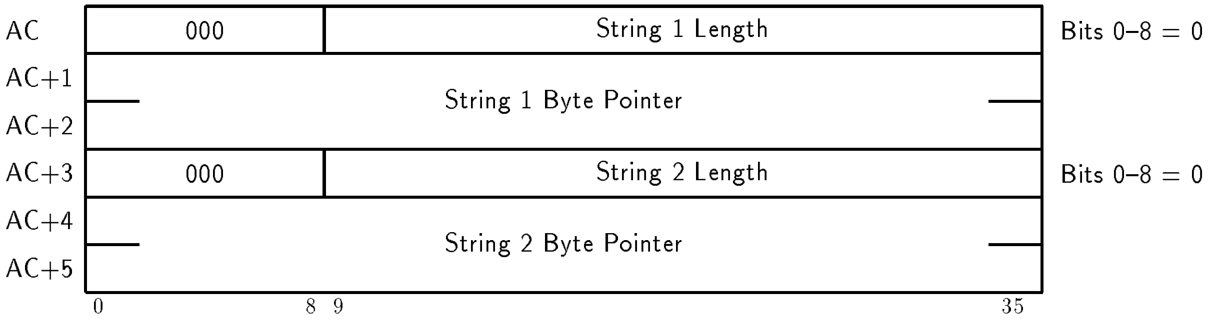


$C \neq 0, 4$
Bits 9–12 = 0
E1 is not used.[69]

Compare two strings and skip the next instruction if the condition specified by C is satisfied. The two strings are defined by the contents of a block of six accumulators. The strings are compared according to a collating sequence based on the algebraic relationships of their bytes taken as unsigned binary integers.

---

[68] However, as of microcode version 2.1[442], the KL10 will return 0 in AC bits 9–35 even if source bytes have been skipped.



Beginning at the left, compare string 1 with string 2, byte by byte, until a pair of bytes that are not identical is encountered. If a string runs out before an inequality is found, continue the comparison using a byte from *E0*+1 in lieu of bytes from string 1, or a byte from *E0*+2 in lieu of bytes from string 2, whichever is shorter.

Upon either encountering an inequality between corresponding bytes of the two strings or reaching the end of the longer string, stop the comparison and skip the next instruction if condition *C* is satisfied. The various values of *C* select different conditions and, therefore, specific forms of this instruction, as follows.

`CMPSL`	Compare Strings and Skip if String 1 Less than String 2	001
`CMPSE`	Compare Strings and Skip if String 1 Equal to String 2	002
`CMPSLE`	Compare Strings and Skip if String 1 Less than or Equal to String 2	003
`CMPSGE`	Compare Strings and Skip if String 1 Greater than or Equal to String 2	005
`CMPSN`	Compare Strings and Skip if String 1 Not Equal to String 2	006
`CMPSG`	Compare Strings and Skip if String 1 Greater than String 2	007

At the end, the byte pointers point to the last positions referenced in the strings and bits 9–35 of AC and AC+3 contain the number of bytes left in the strings beyond the unequal pair. The strings themselves are not affected. Note that, except in the case where the inequality occurs at the last byte, the comparison continues to the end of the strings only if they are equal; in both of these cases, the final states of the pointers and lengths are the same.

If an interrupt or page failure occurs during execution of a string move or compare, the accumulators are adjusted for what has already been done. Afterwards, the instruction resumes as though starting at the beginning but manipulates substrings that are simply those parts of the original strings left from where the instruction was interrupted.

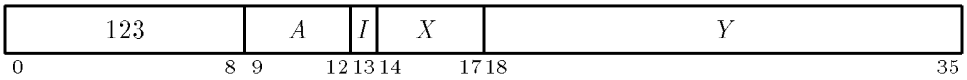
The string compare instructions are useful for such applications as alphabetizing strings that represent words.

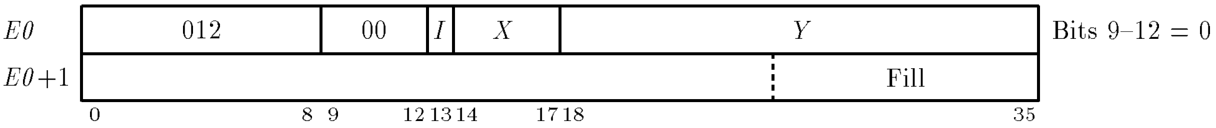
2.13 Decimal Conversion[63]

Included among the string instructions are four for converting between binary and decimal. The binary is always a twos complement, double–length binary integer in the format given in §1.5.1: the magnitude is the 70–bit string in bits 1–35 of the two words, bit 0 of the high–order word is the sign, and bit 0 of the low–order word is a copy of the sign but is never used in any operation. The decimal is a string of bytes representing decimal digits (*the reader should be familiar with the general information and cautions about strings presented at the beginning of the previous section*). To be capable of conversion to double–length binary, a decimal string can have a maximum of twenty–two significant digits, although the string may be longer because of the presence of leading zeros or nonnumeric characters. The decimal value corresponding to the binary maximum of 2^{70} is 1 180 591 620 717 411 303 424.

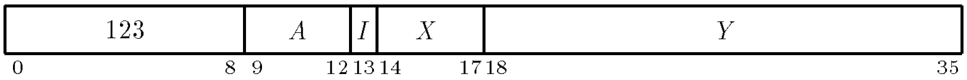
The four instructions are for converting with offset or translation in the two directions. All are two–word instructions, where the first has the EXTEND code 123, and all use a block of accumulators. Decimal–to–binary uses five accumulators, and binary to decimal requires a block of six, but one within the block is not used.

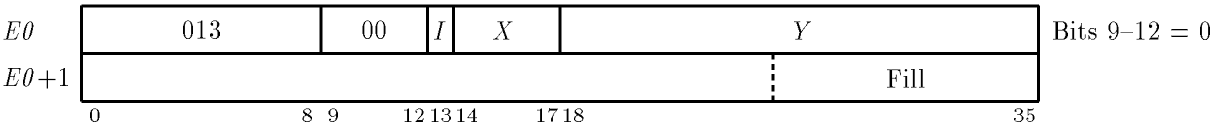
CVTBDO Convert Binary to Decimal Offset



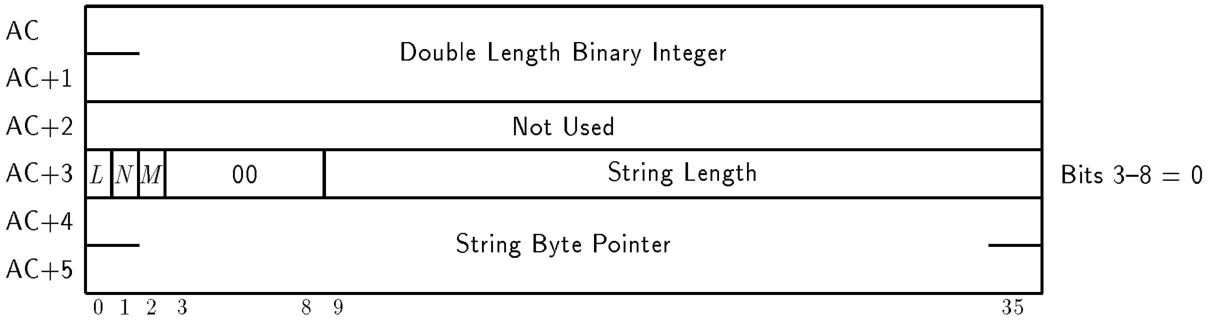


CVTBDT Convert Binary to Decimal Translated





Convert the magnitude of a double–length binary integer into a decimal digit string, offset or translated. The integer is given, and the string space is defined by the contents of a block of six accumulators.



Determine the number of decimal digits required to convert the binary integer; if this number is greater than the string length given by AC+3 bits 9–35, go on to the next instruction without affecting the string space or the accumulators in any way.[70] Note that the string length must specify a minimum of one digit byte, even if the binary number is zero, because representing zero in decimal requires at least the digit “0” (a string with no bytes cannot represent anything—not even zero). If the converted integer will fit in the defined string space, continue as follows.

If the binary integer in AC,AC+1 is not zero, set *N*; if it is less than zero, set *M* (minus). If the number of digits required is less than the given string length and *L* is 1, place the leading fill character from *E0*+1 in the excess positions at the left in the string space. This action causes the result to be right justified. Clear AC+3 bits 9–35.

Compute each decimal digit for a positive representation of the magnitude of the binary integer (highest order first); and, for each, do one or the other of the following two operations, depending on which instruction is being performed.

> If the instruction is CVTBDO, add *E1* to the computed digit algebraically (bit 18 is the sign).
>
> If the instruction is CVTBDT, for the digit substitute a byte from the right half of location *E1*+*D* in the translation table, where *D* is the value of the digit, unless this is the last digit in the conversion, in which case make the substitution from the right half of the location if *M* is 0 or from the left half if *M* is 1.

Place each offset or translated byte in the next position in the string space, compute the next digit, and continue as described above. When the conversion is complete—all digits computed, offset or translated, and deposited—clear AC and AC+1 and skip the next instruction.

At the end, the byte pointer points to the last byte deposited in the string space and AC, AC+1, and AC+3 bits 9–35 all contain zero. If unused interior bits in the string are clear initially, they are left clear; otherwise, unused interior destination bits are indeterminate. The source string is unaffected.

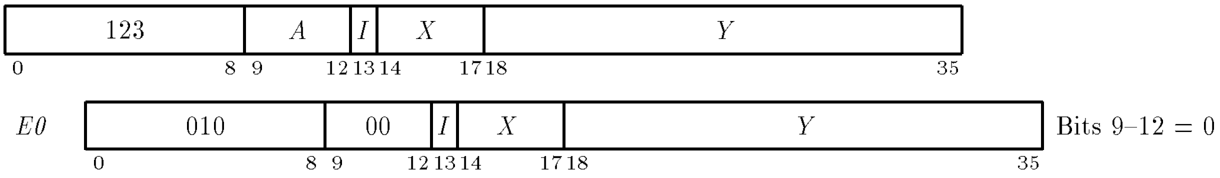
Notes: The translation table, which starts at *E1*, contains ten locations for the decimal digits, each with substitute bytes in both half-words, but the left half is used only for the final digit. This allows

---

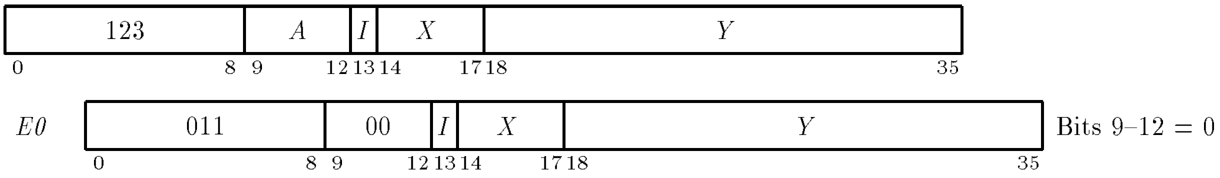
[70] *Caution*: In the KL10 the *N* and *M* flags are set up first and may therefore be affected even by an instruction that is aborted because the binary integer is too large.

the program to use a different final byte for a decimal string converted from a negative number. Setting N indicates that the number converted is not zero; the state of the flag has no effect on the execution of the instruction.

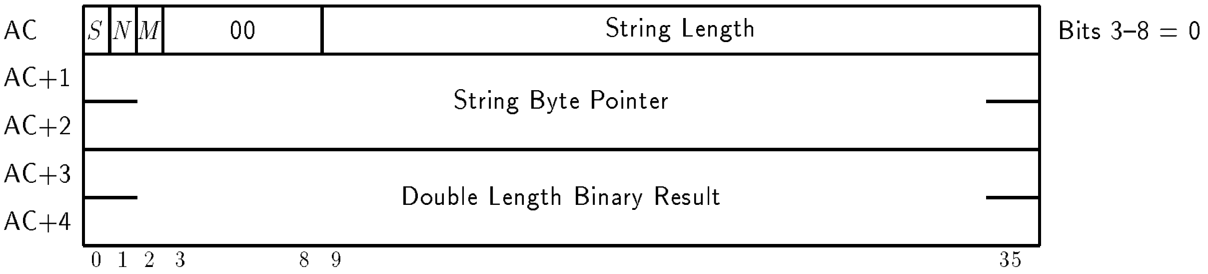
CVTDBO Convert Decimal to Binary Offset



CVTDBT Convert Decimal to Binary Translated



Convert a decimal string, offset or translated, to a double–length binary integer. A block of five accumulators is used for defining the decimal string and receiving the binary result.

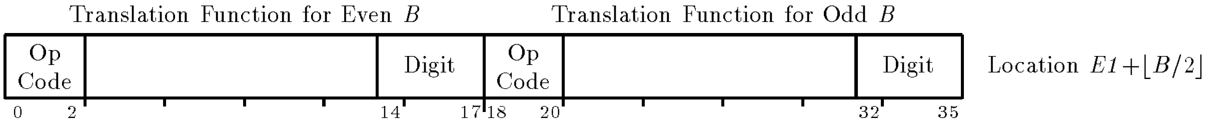


If S is 1 initially, there is already a binary number of significance in AC+3,AC+4: use it as a base for further accumulation of the digits derived from the decimal string. Otherwise, begin with a zero base.

If the instruction is CVTDBO, set S to indicate the conversion has started.

Beginning at the left, read each byte from the string and, for each, do one or the other of the following two operations, depending on which instruction is being performed.

> If the instruction is CVTDBO, add $E1$ to the byte algebraically (bit 18 is the sign).
>
> If the instruction is CVTDBT, carry out the corresponding translation function given in the appropriate half–word at location $E1 + \lfloor B/2 \rfloor$ in the translation table, where B is the value of the byte. Each word in the table has this format:



Perform the function specified by the op code in the half–word corresponding to the byte, as follows (setting S signals the start of significant digits in the the decimal string):

0 If S is 1, substitute the table digit for the byte. If S is 0, ignore this byte and go on to the next.

1 Terminate the conversion.

2 Clear M and, if S is 1, substitute the table digit for the byte. If S is 0, ignore this byte and go on to the next.

3 Set M and, if S is 1, substitute the table digit for the byte. If S is 0, ignore this byte and go on to the next.

4 Set S and N and substitute the table digit for the byte.

5 Set N and terminate the conversion.

6 Set S and N, clear M, and substitute the table digit for the byte.

7 Set S, N, and M and substitute the table digit for the byte.

If the translation function terminates the conversion, or the offset or translated digit is greater than 9, put the number of bytes remaining in the string in AC bits 9–35, put the partial binary result accumulated so far in AC+3,AC+4, and go on to the next instruction. Otherwise multiply the current binary value by 10_{10}, add in the current digit, and read the next byte from the string to continue as described above until the conversion is finished.

CAUTION

It is up to the programmer to keep track of the size of the decimal number—the hardware runs no test on the string. If there are too many significant digits, the most significant part of the binary is lost, and the processor gives no indication of it.

The conversion is regarded as complete only when all bytes of the decimal string have been processed without causing a termination or generating a digit outside the range 0–9. Upon completion, negate the accumulated binary if M is 1, place the result (negated or not) in AC+3,AC+4, and skip the next instruction.

At the end, the byte pointer points to the last byte read from the decimal string and AC bits 9–35 contain the number of unprocessed bytes left in the decimal string (if any). The string itself is unaffected. The translation table starts at location *E1*; and, since there are two functions per word, it contains 2^{n-1} locations, where n is the number of bits in a byte. The address is generated by adding the left $n-1$ bits of a byte to *E1*.

Notes: CVTDBO always sets S immediately, but in CVTDBT its setting is controlled by the translation functions. Hence, an instruction can skip over leading fill characters or nonnumeric characters preceding the decimal part of a string. If an interrupt or page failure occurs during this instruction, the number of bytes yet to be processed is put in AC bits 9–35 and the partial binary accumulated so far is placed in AC+3,AC+4. Thus, when the instruction resumes after an interruption with S set, it simply continues where the conversion left off, adding the next digit to ten times the binary previously saved. If the programmer wishes to preset S to add the decimal string to a significant binary base already in AC+3,AC+4, he must be aware that the base is multiplied by ten before the first digit is added.

For a decimal string $abcde$, the evaluation procedure is

$$(((a \times 10 + b) \times 10 + c) \times 10 + d) \times 10 + e$$

which is equivalent to

$$\begin{aligned} e \times 1 \\ + d \times 10 \\ + c \times 100 \\ + b \times 1000 \\ + a \times 10000 \end{aligned}$$

The operations are all done in binary arithmetic.

Translation functions manipulate M, but the program can set it prior to either instruction to indicate that the decimal string represents a negative number. N can also be preset or manipulated through the translation table, but its state has no effect on the execution of the instruction.

For decimal strings with 4–bit digits, conversion can be done by CVTBDO or CVTDBO with a zero offset. However, note that decimal bytes need not be four bits: they can be larger, using any decimal code, provided only that, on conversion to binary, they are in the range 0–9 (0–1001 binary) after offset or translation.

In ASCII numeric strings, the bytes representing the digits are 60–71 octal. Conversion to ASCII decimal would be by CVTBDO with offset 60 (48 decimal), and CVTDBO with offset −60 would convert in the opposite direction. Consider an ASCII string containing decimal numbers of various unknown lengths separated by semicolons (ASCII code 73). The program could convert all of these numbers to binary by specifying a constant, suitably large string length, while giving a sequence of CVTDBOs with offset −60. Each conversion would terminate (nonskip) upon encountering a semicolon, because its offset value is 11 decimal. Between conversions, the program would have to store away the result and clear S by a sequence like this:

```
EXTEND  AC,[CVTDBO 0,-60]         ;Convert
DMOVEM  AC+3,VALUE1               ;Store result
TLZ     AC,700000                 ;Reset SNM
EXTEND  AC,[CVTDBO 0,-60]
DMOVEM  AC+3,VALUE2
TLZ     AC,700000
 ...
```

If there were very many numbers, the program would naturally use only one of the above sets of three instructions in a loop, along with some mechanism to change the storage address and test whether to reiterate. The procedure cannot provide a negative result. If the same situation were handled by translation, the table would not actually start at *E1*—it would run from *E1*+30 to *E1*+35.

2.14 String Editing[63]

The EDIT instruction implements more complex operations on strings than merely moving or translating; before investigating EDIT, the reader should be familiar with the general characteristics of strings (and cautions about them) as presented at the beginning of §2.12. EDIT provides the facilities needed, particularly in COBOL and PL/I, to create formatted character strings for output. Typical features are the ability to suppress leading zeros, insert special symbols such as decimal points or currency symbols, and recognize different types of numbers for operations like adding "CR" or "DB" after them. When numbers appear in running text, leading zeros are usually deleted; when they are lined up in columns (such as in a financial statement), the practice is to substitute spaces.

EDIT uses the usual source and destination byte pointers, but no string lengths are given. Instead the source bytes are processed by commands in a pattern command string, whose structure is determined by the expected length of the source. The pattern commands are 9–bit bytes packed four to a word. They are executed according to a pattern pointer, which supplies the address of a memory location[71] and a 2–bit byte number, wherein the numbers 0–3 identify the bytes from left to right in the word. The destination string space is assumed to be large enough for whatever string EDIT creates.

Available to the procedure are a translation table at *E1* like that of MOVST and a message insertion table following *E0*. *E0*+1 contains the fill character—typically a space—for suppression of leading zeros; but, if the whole word containing the fill character is zero, the fill is not inserted in the destination space, thus deleting leading zeros. *E0*+2 contains a float character—typically a currency symbol or plus sign—which, if the word containing it is non–zero, is inserted before the first significant byte. The table can extend to *E0*+100, thus supplying an additional sixty–two characters for insertion in the string being generated. Insert characters are typically decimal point, comma, "C" and "R", and so forth.

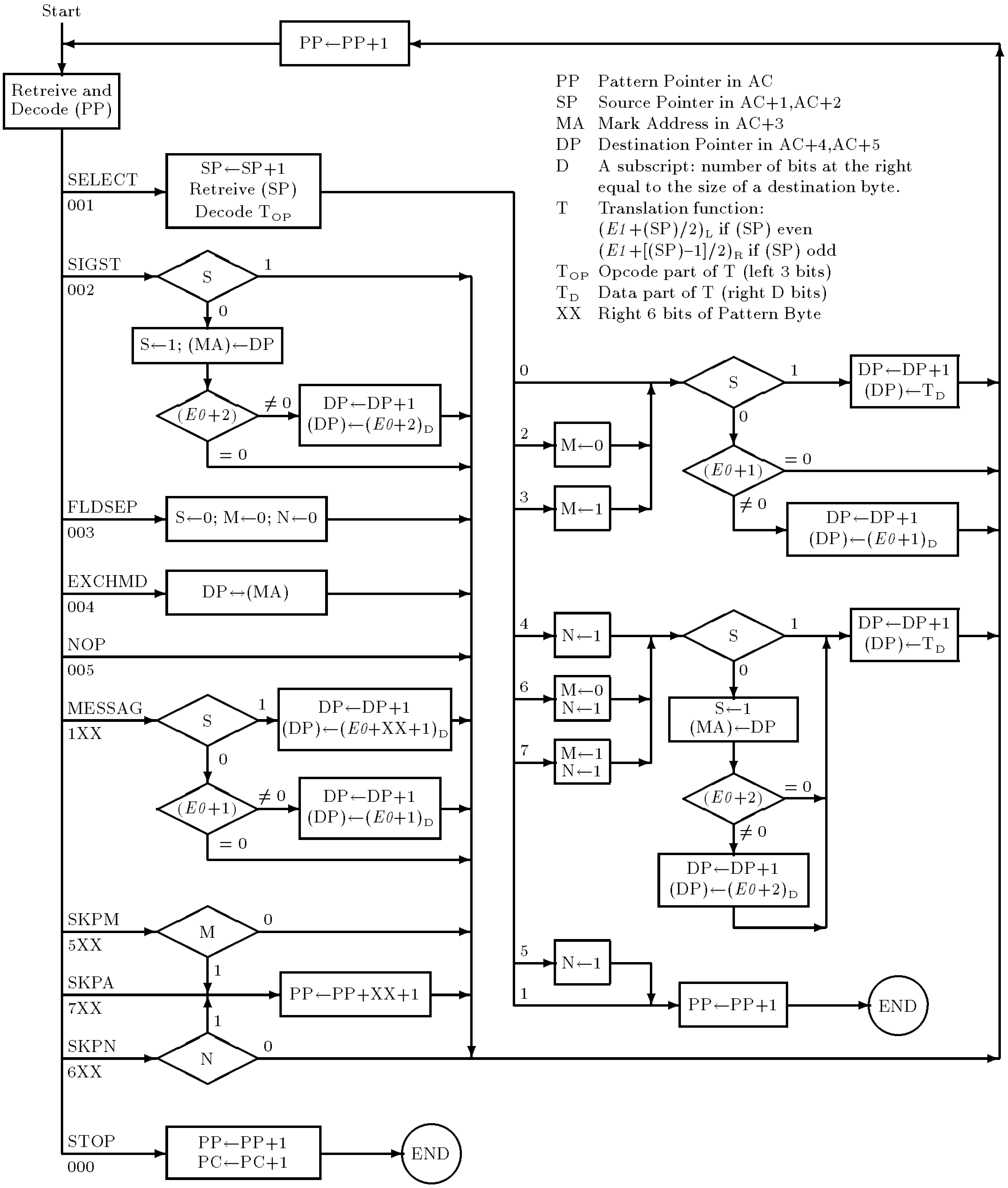
For signaling significance, AC has an S bit, which can be set from the translation table when significance starts. At this point the destination string position is marked by storing the current value of the destination pointer at a location specified by a mark address. This provides a record of where significance started, so the instruction can go back to make revisions if needed after receiving more information from the source.

EDIT is a two–word instruction, where the first has the EXTEND code 123, and it uses a block of six accumulators. The description is accompanied by a flowchart in Figure 2.3.

---

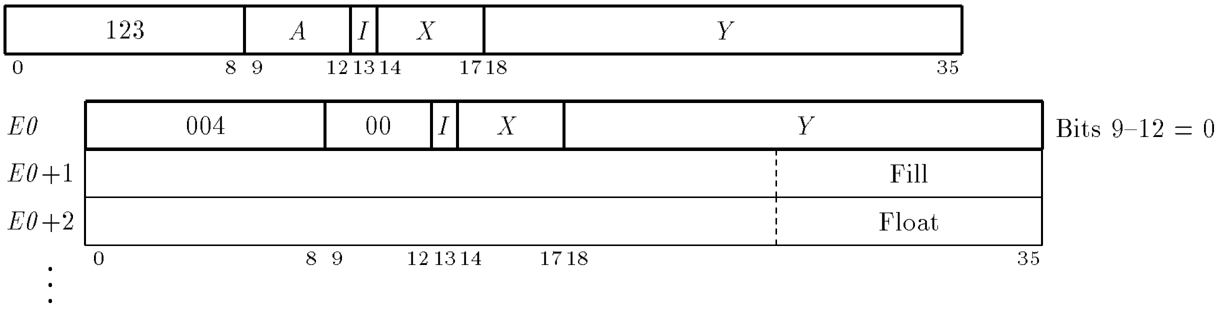
[71] When EDIT is executed in section zero, bits 6–17 of the pattern pointer should be zero. When EDIT is executed in a non–zero section, the pattern pointer is a global address.

Figure 2.3: EDIT Instruction Flowchart

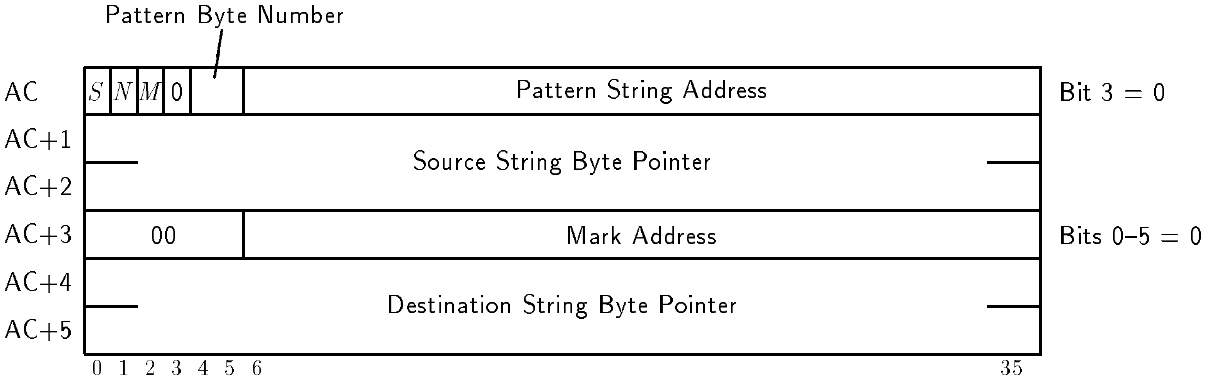


PP	Pattern Pointer in AC
SP	Source Pointer in AC+1,AC+2
MA	Mark Address in AC+3
DP	Destination Pointer in AC+4,AC+5
D	A subscript: number of bits at the right equal to the size of a destination byte.
T	Translation function: $(E1+(SP)/2)_L$ if (SP) even $(E1+[(SP)-1]/2)_R$ if (SP) odd
T_{OP}	Opcode part of T (left 3 bits)
T_D	Data part of T (right D bits)
XX	Right 6 bits of Pattern Byte

EDIT Edit String



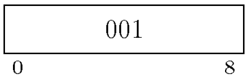
To edit a source string, execute the commands in the pattern string, employing byte substitutions from a translation table at *E1* and inserting characters from a message insertion table at *E0*+1; place the result in the destination string space. Source, destination, and pattern are defined by the contents of a block of six accumulators.



Definitions: Initially the pattern pointer, which comprises the pattern string address and byte number, points to the first pattern command. Pattern byte counting is effected by incrementing the byte number unless it is 3, in which case it is changed to 0 and the address is incremented. The address is limited to bits 18–35 if the program is running in section zero. The mark address is simply the address of the first in a pair of locations for receiving the destination string byte pointer as a mark. Of course, if the destination pointer is local, only one location is used to store it. Furthermore, if the program is running in section zero, the mark address is limited to bits 18–35 and always points to a single location. In the following, any reference to reading a source byte shall be taken to mean that the source string byte pointer is incremented first, and any reference to placing a character in the next position in the destination string space shall be taken to mean that the destination string byte pointer is incremented first.

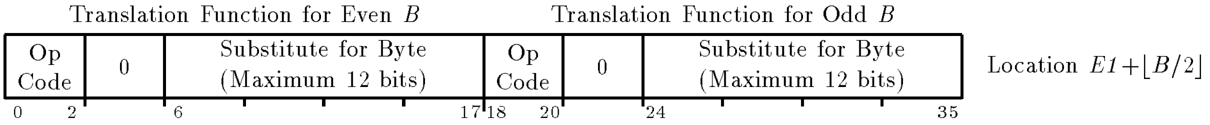
Execute the pattern command specified by the pattern pointer. At the completion of any pattern command, unless the edit has been ended by a STOP command or a terminating translation function, increment the pattern pointer and execute the pattern command then specified by it. There are ten such commands, as follows (all other command bytes are reserved and must not be used).

SELECT



Select Next Source Byte

Read the next byte from the source string and carry out the corresponding translation function given in the appropriate half–word at location $E1+\lfloor B/2\rfloor$ in the translation table, where B is the value of the source byte. Each word in the table has this format.



Perform the function specified by the op code in the half–word corresponding to the source byte as follows.

0 If S is 1, place the substitute in the next position in the destination string space. Otherwise, if location $E0+1$ is non–zero, place the fill character from it in the next destination position.

1 Increment the pattern pointer and go on to the next instruction.

2 Clear M and then perform function 0.

3 Set M and then perform function 0.

4 Set N. If S is 1, place the substitute in the next position in the destination string space. Otherwise, do the following: set S; put the current value of the destination byte pointer at the location specified by the mark address; if location $E0+2$ is non–zero, put the float character from it in the next destination position; then place the substitute in the next destination position after that.

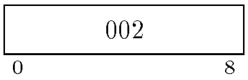
5 Set N, increment the pattern pointer, and go on to the next instruction.

6 Clear M and then perform function 4.

7 Set M and then perform function 4.

Notes: The translation table starts at location $E1$, and, since there are two functions per word, it contains 2^{n-1} locations, where n is the number of bits in a byte. The address is generated by adding the left $n-1$ bits of a byte to $E1$.

SIGST



Start Significance

If S is 0, do the following: set S; put the current value of the destination pointer at the location specified by the mark address; if location $E0+2$ is non–zero, put the float character from it in the next destination position.

Notes: A typical use of this command might be before a final character to guarantee that zero is represented by one “0” or, if the number of cents is 00004, to put in a decimal point and generate a result of .04.

MESSAG+n *Insert Message Character*

1	n
0 2	3 8

If S is 1, place the character from $E0+n+1$ in the next destination position. Otherwise, if location $E0+1$ is non–zero, place the fill character from it in the next destination position.

FLDSEP *Separate Fields*

003
0 8

Clear S, M, and N.

Notes: Essentially this instruction causes the procedure to start over on a new substring. A typical use would be in handling a series of numbers (separated by some character) where one would want to suppress leading zeros in all of them.

EXCHMD *Exchange Mark and Destination Pointers*

004
0 8

Interchange the destination pointer presently held in AC+4,AC+5 with the mark pointer at the location specified by the mark address.

Notes: This makes it possible to go back to where significance began in order to revise the destination string in light of further processing of the source, but at the same time saving the present position. A return forward can be made simply by repeating the instruction.

It is unlikely to be very useful for the programmer to set up an initial mark pointer. In any normal procedure, a mark is created from the destination pointer and is simply a particular state of it. Hence, the destination and mark pointers have the same number of words. The result is indeterminate if the programmer deliberately sets up mark and destination pointers of different types.

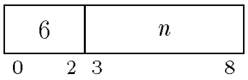
SKPM+n *Skip on M*

5	n
0 2	3 8

If M is 1, skip over the next $n+1$ pattern commands by incrementing the pattern pointer $n+1$ times.

Notes: M is generally used as a minus sign (i.e., to indicate a string is negative), but the programmer can use it for any purpose. A typical use would be to determine whether “CR” or “DB” should be inserted after a number.

SKPN+n

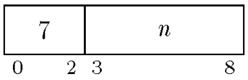


Skip on N

If N is 1, skip over the next $n+1$ pattern commands by incrementing the pattern pointer $n+1$ times.

Notes: N is generally set to mean the string is non–zero, but the programmer can use it for any purpose. Suppose we wish to output a blank on zero, but use of SIGST to handle cents–only quantities has produced ".00". We could use SKPN after the last source byte, so that, if the output is non–zero, we would skip over commands that would otherwise go back and blank the output.

SKPA+n



Skip Always

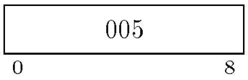
Skip over the next $n+1$ pattern commands by incrementing the pattern pointer $n+1$ times.

Notes: This command is used mostly to reverse the meaning of the other skips. For example, the sequence "SKPN,X" skips command X if N is 1, but the sequence "SKPN,SKPA,X" executes it if N is 1. SKPA can also be used to extend a conditional skip beyond sixty–four commands, as in

SKPN+77,...*63 bytes*...,SKPA+0,SKPA+3,...*4 bytes*...,X

in which N being 1 causes a skip over sixty–seven significant commands to get to X.

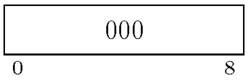
NOP



No–op

Do nothing.

STOP



Stop Edit

Increment the pattern pointer, end the edit, and skip the next instruction.

At the end, the byte pointers point to the last positions referenced in source and destination and the pattern pointer points to the command byte following the last one executed. However, if the pattern gives an EXCHMD after the final byte is placed in the destination string, the "destination pointer" is actually at the mark location rather than in AC+4,AC+5. If unused interior bits in both strings are clear initially, they are left clear; otherwise, unused interior destination bits are indeterminate. The source string is unaffected.

Notes: If an interrupt or page failure occurs during EDIT, the accumulators are adjusted for restarting at the beginning of the current pattern command.

Example: The following program uses binary–to–decimal conversion and editing to translate a binary number into a message of seventeen characters containing a decimal string with appropriate nomenclature for commercial billing purposes. A positive result has the form

$12,345.46 DUE US

whereas a negative result has the form

$12,345.46 CREDIT

but, if the number is zero, the entire field is blank (all spaces). The maximum number the routine can handle is $99,999.99.

This program employs seven accumulators, of which P is for the stack pointer and a block of six, labeled AC1–AC6, is for the EXTEND instructions. In the block, however, AC3 and AC6 are never actually used because the program is entirely local, employing only one–word byte pointers. Beginning at TEMP and FIELD are blocks of eight locations set aside for the EDIT source and destination strings. The routine is called by PUSHJ P,PNTFLD with the amount as a binary number of cents in AC1,AC2. It returns the result beginning at the left in FIELD.

```
PNTFLD: MOVE   AC4,[400000,,7]     ;Convert up to 7 digits with leading fill
        MOVE   AC5,[POINT 7,TEMP]  ;Store decimal in edit source area
        EXTEND AC1,[CVTBDO 60      ;Convert to decimal with leading zeros 60
          JRST ERROR]              ;Here if need too many digits (binary too large)
        MOVEI  AC1,PATTRN          ;Set pattern pointer to first command
        TLNE   AC4,100000;         ;Copy M flag from AC4 (CVTBDO result)
        TLO    AC1,100000          ;to AC1
        MOVE   AC2,[POINT 7,TEMP]  ;Pointer for source string (CVTBDO result)
        MOVEI  AC4,MARK            ;Address of mark pointer
        MOVE   AC5,[POINT 7,FIELD] ;Pointer for destination string
        EXTEND AC1,EDTINS          ;Edit the item
          HALT .                   ;Should never get here
        POPJ   P,0                 ;Return

;Here is the edit instruction
EDTINS: EDIT   TABLE-30   ;Need only digit part of translation table
        " "               ;Fill character is space
        "$"               ;Float character is dollar sign
        ","               ;Message 2 is comma
        "."               ;Message 3 is decimal point
        "D"
        "U"
        "E"
        "S"
        "C"
        "R"
        "I"
        "T"
```

```
;Here is the translation table.  Digits 1--9 set S and N flags
; 0 does not affect the flags

TABLE:   60,,400061
         400062,,400063
         400064,,400065
         400066,,400067
         400070,,400071

;Here is the pattern

PATTRN:  001001,,102001  ;SELECT SELECT MESSAG+2 SELECT
                         ; 2 digits, comma, digit
         001001,,002103  ;SELECT SELECT SIGST MESSAG+3
                         ; 2 more digits, then start significance and insert
                         ; a decimal point
         001001,,100506  ;SELECT SELECT MESSAG+0 SKPM+6
                         ; 2 more digits (cents) and a space, then skip
                         ; next 7 commands if number was negative
         104105,,106100  ;Append the message ''DUE US''
         105107,,705110  ;Then skip 6 pattern commands
         111106,,104112  ;Append the message ''CREDIT''
         113613,,004100  ;If number is non-zero skip 12 commands
         100100,,100100  ;Else exchange mark and destination pointers
         100100,,100100  ;and blank out result
         100100,,0       ;Then stop

MARK:    BLOCK 1
TEMP:    BLOCK 10
FIELD:   BLOCK 10
```

2.15 Programming Examples

Before continuing to more system–related subjects, let us consider some simple programs that demonstrate the use of a variety of the instructions described thus far.

2.15.1 Processor Identification

The instruction repertoires of all PDP–10 processors and the 166 processor used in the PDP–6 are very similar; most programs require no changes to run on any of them. Because of minor differences and the fact that some instructions are not available on the earlier machines, a program that is to be compatible with all should have some way of distinguishing which machine it is running on. The following test routine suffices.[72]

---

[72] It is beyond the scope of this manual to document the behavior of other vendors' versions of the PDP–10 architecture. PDP–10 architectures have also been built by Foonly and by Systems Concepts.

```
JFCL    17,.+1          ;Clear flags
JRST    .+1             ;Change PC
JFCL    1,PDP6          ;PDP-6 has PC Change flag
MOVNI   AC,1            ;Others do not.  Make AC all 1s
AOBJN   AC,.+1          ;Increment both halves
JUMPN   AC,KA10         ;KA10 carries to left half
BLT     AC,0            ;Try BLT.  Source=0; Dest=0.  AC must not be 0
JUMPE   AC,KI10         ;KI10 if AC = 0
MOVSI   AC,400000       ;Largest negative number
ADJBP   AC,[430100,,0]  ;Check what this does
CAMN    AC,[430100,,0]  ;The KL won't change this
JRST    KL10            ;This must be a KL10
MOVSI   AC,450000       ;A one-word global byte pointer
IBP     AC              ;What does this do?
CAME    AC,[450000,,0]  ;The KS doesn't change this
JRST    XKL1            ;This must be an XKL-1
JRST    KS10            ;Otherwise, it's a KS10
```

2.15.2 Parity

Parity procedures are used regularly to check the accuracy of stored information. Parity generation and checking are generally handled automatically by memory and high–speed, block–oriented peripheral devices but must be handled by the program for character–oriented devices. Consider 8–bit characters, for which the program carries out two procedures: for output it generates a parity bit from seven data bits to produce an 8–bit character with parity; following input it checks the parity of the eight bits received. In either case, however, the program can simply find the parity of an 8–bit character by regarding the seven output data bits as eight, including an irrelevant extra bit. The two procedures then differ only in the final action. In the first case, the program uses the result to adjust the eighth bit for correct parity, whereas, in the second, it checks the result for an indication of error.

Assuming the character is right–justified in accumulator A and the rest of A is clear, as it would be were the character placed in A by a LDB instruction or a DATAI, the simplest and quickest procedure would be to use A to index an XCT into a table, each of whose locations contains an instruction that adjusts the parity for output or jumps to a routine for erroneous input. This procedure would normally be unacceptable because of the very large memory requirements. However, the table can be reduced to sixteen entries without excessive loss in speed, by exclusive–ORing the left and right halves of the character and indexing on the result (parity is invariant under the exclusive–OR function, which essentially disposes of pairs of 1s). This example, which uses a second accumulator T for character manipulation, requires six memory references to generate odd parity. (Numbers of memory references and locations given do not include those for the POPJ, which will be regarded as subroutine overhead. Similarly every example also requires that the program give a PUSHJ to get to the subroutine. This example is counted as five memory references for instruction fetches and one memory reference for a data fetch, the instructions in PARTAB are considered to be data.)

```
PARITY:    MOVEI    T,(A)        ;Copy character in T
           LSH      T,-4         ;Line up halves
           XORI     T,(A)        ;Reduce paritywise to 4 bits
           ANDI     T,17         ;Wipe out unwanted bits
           XCT      PARTAB(T)    ;Execute indicated table item
           POPJ     P,

PARTAB:    XORI     A,200        ;0 --- change high bit
           JFCL                  ;1 --- no--op
           JFCL                  ;2
           XORI     A,200        ;3
           JFCL                  ;4
           XORI     A,200        ;5
           XORI     A,200        ;6
           JFCL                  ;7
           JFCL                  ;10
           XORI     A,200        ;11
           XORI     A,200        ;12
           JFCL                  ;13
           XORI     A,200        ;14
           JFCL                  ;15
           JFCL                  ;16
           XORI     A,200        ;17
```

To handle even parity, interchange the JFCLs and XORIs in the table or change the MOVEI T,(A) to MOVEI T,200(A).

The next example does exactly the same thing, but it substitutes a little more computation for use of a table. In other words, it takes a little more time (7.5 memory references average) but less than half the memory space.

```
PARITY:  MOVEI   T,200(A)  ;Copy character with high bit
         LSH     T,-4      ;complemented, then fold copy into 4
         XORI    T,(A)     ;bits with opposite parity
         TRCE    T,14      ;Are left two both 0?
         TRNN    T,14      ;Or both 1?
         XORI    A,200     ;Yes, change high bit
         TRCE    T,3       ;Are right two both 0?
         TRNN    T,3       ;Or both 1?
         XORI    A,200     ;Yes, change for even, restore for odd
         POPJ    P,
```

Note that this example does not require the rest of A to be clear. For even parity, change the address in the MOVEI from 200 to 0.

Finally, let us consider the extreme of substituting computation for memory. Starting with the character $abcdefgh$ right–justified in A, first copy it to T and then duplicate it twice to the left, producing

$$abc \qquad def \qquad gha \qquad bcd \qquad efg \qquad hab \qquad cde \qquad fgh$$

where the bits (in positions 12–35) are grouped corresponding to the octal digits in the word. ANDing this with

$$001 \qquad 001 \qquad 001 \qquad 001 \qquad 001 \qquad 001 \qquad 001 \qquad 001$$

retains only the least–significant bit in each 3–bit set, the result can be represented by

$$cfadgbeh$$

where each letter represents an octal digit having the same value (0 or 1) as the bit originally represented by the same letter. Multiplying this by 11111111_8 generates the following partial products:

$$\begin{array}{ccccccccccccccc}
 & & & & & & & c & f & a & d & g & b & e & h \\
 & & & & & & c & f & a & d & g & b & e & h & \\
 & & & & & c & f & a & d & g & b & e & h & & \\
 & & & & c & f & a & d & g & b & e & h & & & \\
 & & & c & f & a & d & g & b & e & h & & & & \\
 & & c & f & a & d & g & b & e & h & & & & & \\
 & c & f & a & d & g & b & e & h & & & & & & \\
c & f & a & d & g & b & e & h & & & & & & &
\end{array}$$

Since any digit is at most 1, there can be no carry–out from any column with fewer than eight digits unless there is a carry–in to it. Hence, the octal digit produced by summing the center column (the one containing all the bits of the character) is even or odd as the sum of the bits is even or odd. Thus, its least significant bit (bit 14 of the low–order word in the product) is the parity of the character: 0 if even, 1 if odd.

The above may seem a very complicated procedure to do something trivial, but it is effected by this quite simple sequence:

```
PARITY:   MOVEI    T,(A)     ;Copy in T
          IMULI    T,200401  ;Duplicate twice
          AND      T,ONES    ;Pick LSBs
          IMUL     T,ONES    ;Generate product
          TLNN     T,10      ;Is bit 14 odd?
          XORI     A,200     ;No, change parity
          POPJ     P,
          .
          .
          .
ONES:     11111111
```

This procedure uses a minimum of both memory references and memory space but takes considerably more time, because the multiplications are relatively slow.

The following table shows the trade–off of memory references against memory space for the above four procedures. The time is proportional to the number of references, except in case 4.

	References	*Locations*
1. Table Lookup	2	257
2. Folded Lookup	6	21
3. Folded Computation	7.5	9
4. Computation	7.5	7

2.15.3 Reversing the Order of Digits

Suppose one wishes to reverse the order of the digits in the 6–bit character *abcdef*, where the letters represent the bits of the character. One can first duplicate it three times to the left and shift the result left 1 place, producing

$$a \quad bcd \quad efa \quad bcd \quad efa \quad bcd \quad efa \quad bcd \quad ef0$$

where the bits are grouped corresponding to the octal digits in the word. ANDing this with

$$1 \quad 000 \quad 100 \quad 100 \quad 010 \quad 010 \quad 000 \quad 001 \quad 000$$

gives

$$a \quad 000 \quad e00 \quad b00 \quad 0f0 \quad 0c0 \quad 000 \quad 00d \quad 000$$

This number is configured such that the residues of the values of its bits modulo $2^8 - 1$ are in exactly the opposite order from the bits of the original character and have the binary orders of magnitude 0–5. In other words, this number is equal to the sum of the numbers in the upper row below, and dividing each of these addends by 255_{10} gives the remainder listed in the lower row.

Dividend	$f \times 2^{13}$	$e \times 2^{20}$	$d \times 2^3$	$c \times 2^{10}$	$b \times 2^{17}$	$a \times 2^{24}$
Remainder	$f \times 2^5$	$e \times 2^4$	$d \times 2^3$	$c \times 2^2$	$b \times 2^1$	$a \times 2^0$

The remainder in a division is equal to the sum, modulo the divisor, of the remainders left by dividing each of the dividend addends by the same divisor, and the sum of the terms in the lower row is obviously *fedcba*. The above procedure is implemented by this sequence (attributed to Schroeppel[73]) where the character is right–justified in accumulator A (with the rest of A clear) and its reverse is right–justified in accumulator A+1.

```
IMUL   A,[2020202]      ;4 copies shifted left 1
AND    A,[104422010]    ;Pick bits for reverse
IDIVI  A,377            ;Divide by 2^8 - 1
```

To reverse eight bits, one can use a similar procedure (also attributed to Schroeppel) where, again, the original character is right–justified in A and its reverse is right–justified in A+1. However, this

---

[73] HAKMEM, item 167, page 78 (*Artificial Intelligence Memorandum, No. 239*, February 29, 1972, MIT Artificial Intelligence Laboratory).

time the manipulation cannot be managed within a single–length word, so different forms of multiply and divide are used.

```
MUL     A,[100200401002]  ;5 copies in A and A+1
AND     A+1,[20420420020] ;Pick bits for reverse via
ANDI    A,41              ;residues mod 2^10 - 1
DIVI    A,1777            ;Divide by 2^10 - 1
```

2.15.4 Counting Ones

Suppose one wishes to count the number of 1s in a word. One could. of course, check every bit in the word. However, there is a quicker way if one remembers that, in any word and its twos complement, the rightmost 1 is in the same position, both words are all 0s to the right of this 1, and no corresponding bits are the same to the left (the parts of both words to the left of the rightmost 1 are complements). Hence, using the negative of a word as a mask for the word in a test instruction selects only the rightmost 1 for modification. The example uses three accumulators: the word being tested (which is lost) is in T, the count is kept in CNT, and the mask created in each step is stored in TEMP.

```
MOVEI   CNT,0          ;Clear CNT
MOVN    TEMP,T         ;Make mask to select rightmost 1
TDZE    T,TEMP         ;Clear rightmost 1 in T
AOJA    CNT,.-2        ;Increase count and jump back
...                    ;Skip to here if no 1s left in T
```

CNT is increased by 1 every time a 1 is deleted from T. After all 1s have been removed, the TDZE skips.

The preceding example uses little memory but contains a loop, so the time it takes is proportional to the number of 1s. The next example takes more memory but it takes constant time; hence, it is slower than the above when there are few 1s (fewer than eight) but is much faster when there are many. The word, which is lost, is in accumulator A, and the answer appears in accumulator A+1 (for convenience in nomenclature, let B = A+1). The routine (attributed to Gosper, Mann, and Leonard[74]) has three distinct parts and is based on the fact that, in a binary word, counting 1s is equivalent to calculating the sum of the digits. The first part, of seven instructions, manipulates the octal digits of the word so as to replace each digit by the number of 1s in it. Taking D as an octal digit and $\lfloor x \rfloor$ to mean the largest integer contained in x, the algorithm used to make the substitution is

$$D - \lfloor D/2 \rfloor - \lfloor D/4 \rfloor$$

Of course, the computer always acts in binary terms regardless of programmer interpretation. In this case the procedure carried out on each 3–bit piece abc is

---

[74] Ibid, item 169, page 79.

$$abc - ab - a$$

The instructions effect this algorithm by shifting a copy of the word right 1 place, masking out the LSB of each shifted octal digit to prevent it from interfering with the next digit at the right (i.e., to isolate the digits), and subtracting the shifted word from the original. The same process is then repeated, this time masking out what was originally the middle bit in each digit. That this algorithm gives the correct substitution is evident from the following table, in which it is seen that the bottom number in a given column is the sum of the bits in the octal digit given at the top of the column.

Original digit	0	1	2	3	4	5	6	7
Subtract $\lfloor D/2 \rfloor$	$\underline{-0}$	$\underline{-0}$	$\underline{-1}$	$\underline{-1}$	$\underline{-2}$	$\underline{-2}$	$\underline{-3}$	$\underline{-3}$
	0	1	1	2	2	3	3	4
Subtract $\lfloor D/4 \rfloor$	$\underline{-0}$	$\underline{-0}$	$\underline{-0}$	$\underline{-0}$	$\underline{-1}$	$\underline{-1}$	$\underline{-1}$	$\underline{-1}$
Number of 1s	0	1	1	2	1	2	2	3

The original word has been replaced with a set of twelve numbers whose sum is equal to the number of 1s in the original. The next three instructions add pairs of adjacent numbers so as to replace the original twelve by six numbers whose sum is still the same. These new numbers are isolated in 6–bit pieces of the word, so they can be regarded as digits in a number in base 64. Any number is simply the sum of the values of its digits; i.e., of its digits each multiplied by an appropriate power of the base. Dividing each such addend by 1 less than the base gives the digit itself as remainder. Hence, the third part of the routine simply divides the 6–digit number by 63, producing in B a remainder that is the sum of the remainders from the individual digits; i.e., the sum of the digits.[75]

```
MOVE    B,A                 ;Copy in B
LSH     B,-1                ;Right 1
AND     B,[333333,,333333]  ;Mask out LSBs
SUB     A,B                 ;D − ⌊D/2⌋
LSH     B,-1                ;Right 1 again
AND     B,[333333,,333333]  ;Mask out middle bits
SUBB    A,B                 ;D − ⌊D/2⌋ − ⌊D/4⌋; two copies
LSH     B,-3                ;Shift right 1 octal digit
ADD     A,B                 ;Add numbers in digit pairs
AND     A,[070707,,070707]  ;Throw out extra pair sums
IDIVI   A,77                ;Divide by 63, sum in B
```

If it is known that the 1s in the word are entirely contained within bits 22–35 (the rightmost fourteen bits), the following somewhat shorter routine can be used. This is faster than the loop for more than seven 1s. It first treats the number in quaternary, replacing each digit with the number of 1s in it, and then converts from quaternary to hexadecimal.

---

[75] In general terms, this is the statement that the sum S of the digits in any number N in base b is N mod $(b-1)$—*provided* b is deliberately chosen such that $S < b - 1$. The condition holds here, of course, because the number of 1s in a PDP–10 word is at most 36. It is, in fact, to make this condition hold that the routine converts from base 8 to base 64.

```
MOVEI   B,(A)
LSH     B,-1
ANDI    B,12525          ;Mask out LSBs
SUBB    A,B              ;D − ⌊D/2⌋; two copies
LSH     B,-2             ;Right 1 quaternary digit
ANDI    A,31463          ;Mask out some of digits in A
ANDI    B,31463          ;The rest in B
ADDI    A,(B)            ;Now combine digit pairs
IDIVI   A,17             ;Divide by 15, sum in B
```

Note that the pair of ANDIs gets rid of one out of each set of two identical bit pairs before adding. This is done because there can be digit overflow; i.e., a resulting hexadecimal digit can have more than two significant bits.

2.15.5 Number Conversion

In the standard algorithm for converting a number N to its equivalent in base b, one performs the series of divisions

$$\begin{aligned} N/b &= q_1 + r_1/b, \ \ r_1 < b \\ q_1/b &= q_2 + r_2/b, \ \ r_2 < b \\ q_2/b &= q_3 + r_3/b, \ \ r_3 < b \\ &\vdots \\ q_{n-1}/b &= 0 + r_n/b, \ \ r_n < b \end{aligned}$$

The number in base b is then $r_n \ldots r_3 r_2 r_1$. For example, the octal equivalent of 61 decimal is 75:

$$\begin{aligned} 61/8 &= 7 + 5/8 \\ 7/8 &= 0 + 7/8 \end{aligned}$$

The following decimal print routine converts a 36–bit positive integer in accumulator T to decimal and types it out. The contents of T and T+1 are destroyed. The routine is called by a PUSHJ P,DECPNT, where P is the stack pointer.

```
DECPNT: IDIVI   T,12            ;12₈ = 10₁₀
        PUSH    P,T+1           ;Save remainder
        SKIPE   T               ;All digits formed?
        PUSHJ   P,DECPNT        ;No, compute next one
DECPN1: POP     P,T             ;Yes, take out in opposite order
        ADDI    T,60            ;Convert to ASCII (60 is code for "0")
        JRST    TTYOUT          ;Type out
```

This routine repeats the division until it produces a zero quotient. Hence, it suppresses leading zeros; but, since it is executed at least once, it outputs one "0" if the number is zero. The TTYOUT routine returns with a POPJ P, to DECPN1 until all digits are typed, then to the calling program.

In section zero only, space can be saved in the stack by storing the computed digits in the left halves of the locations that contain the jump addresses. This is accomplished in the decimal print routine by changing

```
            PUSH   P,T+1   to   HRLM  T+1,(P)   ;section zero only
 and
            POP    P,T     to   HLRZ  T,(P)     ;section zero only
```

The routine can handle a 36–bit unsigned integer if the IDIVI T,12 is replaced by

```
        LSHC    T,-^D35         ;Shift right 35 bits into T+1
        LSH     T+1,-1          ;Vacate the T+1 sign bit
        DIVI    T,12            ;Divide double length integer by 10
```

2.15.6 Table Searching

Many data processing situations involve searching for information in tables and lists of all kinds. Suppose one wishes to find a particular item in a table beginning at location TAB and containing N items.[76] Accumulator T contains the item. The right half of A is used to index through the table, while the left half keeps a control count to signal when a search is unsuccessful.

```
        MOVSI   A,-N            ;Put -N,,0 in A
        CAMN    T,TAB(A)        ;Skip if current item not the one
        JRST    FOUND           ;Item found
        AOBJN   A,.-2           ;Try next item until left count = 0
        ...                     ;Item not in list
```

---

[76] N is restricted: it must be $\leq 2^{17}$. Other search methods are advisable before N becomes this large. Because this example uses local addresses, the table, TAB, must fit entirely in one section.

The location of the item (if found) is indicated by the number in the right half of A (its address is that quantity plus TAB). A slightly different procedure would be

```
        MOVSI   A,-N
        CAME    T,TAB(A)        ;Skip if current item is the one
        AOBJN   A,.-1
        JUMPL   A,FOUND         ;Jump if left count < 0
        ...                     ;Item not found
```

2.15.7 List Manipulation

Locations used for a list can be scattered throughout one section of memory if data is kept in the left half of each location and if the right half addresses the next location in the list. The final location in the list is indicated by a zero right half. The following routine finds the last half–word item in the list. It is entered at FIND with the first location in the list addressed by the right half of accumulator T and zero in the left half of T. At the end, the final item is in the right half of T.

```
        HRRZ    U,T             ;Copy the right half of T for local address
        MOVE    T,(U)           ;Move next data item and address to T.
FIND:   TRNE    T,-1            ;Skip if T right = 0; -1 = 777777
        JRST    .-3             ;not 0, not end of list.  Get next item
        HLRZ    T,T             ;Move final item to right
```

The following counts the length of the list in accumulator CNT.

```
        MOVEI   CNT,0           ;Clear CNT
        JUMPE   T,OUT           ;Jump out if T contains 0
        HRRZ    T,(T)           ;Get next address
        AOJA    CNT,.-2         ;Count and go back
```

2.15.8 Extended Addressing

For simplicity the preceding examples have employed only local addressing, because this is mostly what a typical program would use even when running in a non–zero section. The following are straightforward examples to show the differences between local and extended addressing, with and without indexing and indirection. For the following examples, the program is assumed to be running in section 22.

Local reference without indexing or indirection:

```
        MOVE    T,1000
```

loads accumulator T with the contents of location 1000 in section 22.

Local indexing:

```
        MOVEI   X,100
        MOVE    T,1000(X)
```

loads T with the contents of location 1100 in section 22. This would typically be used to access the array element number 100_8, where the array origin (element number 0) is located at 1000 in the current section.

```
        SETZ    T,
        MOVNI   X,100
LOOP:   ADD     T,1000(X)
        AOJL    X,LOOP
```

adds the contents of locations 700–777 in section 22; the sum is in T.

```
        SETZ    T,
        MOVSI   X,-LENGTH
LOOP:   ADD     T,511000(X)
        AOBJN   X,LOOP
```

adds the contents of all locations in an array of length LENGTH starting at location 511000 in section 22. For AOBJN to work properly, LENGTH must not exceed 400001. Note that, since local indexing is used, the references to the array cannot cross into section 23. If LENGTH is greater than 267000 ($1000000 - 511000 = 267000$) the array reference at LOOP wraps around, first into the accumulators, and then continuing from location 20 in section 22.

Global indexing:

```
        MOVE    X,[30,,1000]
        ADD     T,100(X)
```

adds the contents of location 1100 in section 30 to T. Note that if the literal were "22,,1000" the ADD would address location 1100 in the current section, even though the indexing is global.

```
        MOVE    X,[30,,1000]
        ADD     T,-100(X)
```

adds the contents of location 700 in section 30 to T. Were the address portion (Y) of the ADD instruction -1000, it would reference storage location 0 in section 30 (not a fast–memory location).

Furthermore, if the address portion were −2000, it would address location 777000 in section 27, because global indexing can cross the section boundary.

Local indirection:

```
MOVEI   T1,100
MOVEM   T1,1000
ADD     T,@1000
```

adds the contents of location 100 in section 22 to T.

Global indirection.

```
MOVE    T,@[30,,1000]
```

loads T with the contents of location 1000 in section 30. If location 3000 in section 30 contained

```
MOVE    T,2000
```

then, in the current section (22), performing the instruction

```
XCT     @[30,,3000]
```

would load T with the contents of location 2000 in section 30, because the effective–address computation of the target of the XCT is performed in that instruction's section rather than in the section where the XCT appears. If location 4000 in section 30 were to contain

```
JSR     SUBR
```

then an

```
XCT     @[30,,4000]
```

performed in location 100 in section 22 would transfer control to SUBR+1 in section 30, but the PC saved in 30,,SUBR would be 22,,101, because the XCT itself is performed in the current PC section, which is 22.

Global indirection with indexing:

```
MOVEI   X,100
MOVE    T,@[BYTE(1)0,0(4)X(12)30(18)1000]
```

loads T with the contents of location 1100 in section 30. The BYTE operator has created a global indirect word in which the number X has been placed in bits 2–5 of the word and in which 30,,1000 has been placed in bits 6–35.

```
        MOVE    X,[2000000-1]          ;2 sections worth
LOOP:   ADD     T,@[BYTE(1)0,0(4)X(12)30(18)1000]
        SOJGE   X,LOOP
```

adds the 512K array from location 777 in section 32 down to 1000 in section 30. Note that, even if the array contained fewer than 2^{17} words and did not cross a section boundary, it would still not be possible to use AOBJN for the loop, because global indexing uses the entire index register. The following gets the same result with negative indexing.

```
        MOVE    X,[-2000000+1]
LOOP:   ADD     T,@[BYTE(1)0(4)X(12)32(18)777]
        AOJLE   X,LOOP
```

2.16 Unimplemented Operations

Codes not assigned as specific instructions act as unimplemented operations, wherein the word given as an instruction is trapped, either because it should not be given or because it must be interpreted by a routine included for this purpose by the programmer. Codes that are available for interpretive use are unimplemented user operations, or UUOs (the several mnemonics mentioned in this discussion are for convenience and mean nothing to the assembler). Codes in the range 001–037 are for the local use (LUUOs) of the user or the executive. Various other codes are set aside specifically for user communication with the Monitor or for communication between one level of the Monitor and another; in either case these MUUOs are interpreted by the Monitor. Basic codes (except 000) that are not used for instructions or UUOs and extended codes not used by EXTEND are regarded as the unassigned codes; 000 is regarded as an illegal code. All unassigned or illegal codes are processed as MUUOs.

2.16.1 LUUOs

Let us consider first how an LUUO works.

Local Unimplemented User Operation

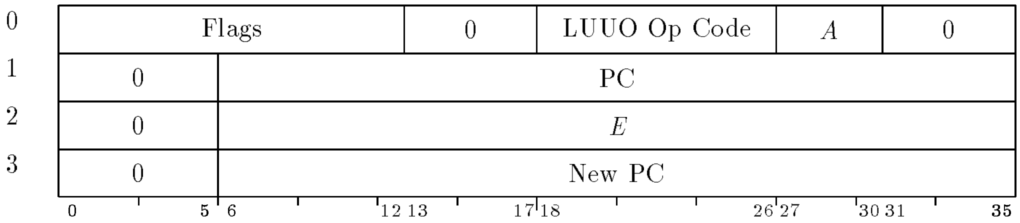
001–037	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the program is running in section zero, store the instruction code, *A*, and the effective–address *E* in bits 0–8, 9–12, and 18–35, respectively, of location 40; clear bits 13–17 of location 40. Execute the instruction contained in location 41. The original contents of location 40 are lost. Every LUUO in section zero uses some pair of locations numbered 40 and 41, but which pair depends upon the circumstances. An LUUO in a user program uses virtual locations 40 and 41 and is thus entirely a part of and under control of the user program. The locations used in executive mode depend on the processor:

XKL-1, KL10, KS10	40 and 41 in executive virtual space
KI10	40 and 41 in the executive process table
KA10	Unrelocated 40 and 41[77]

If the program is running in a non–zero section, perform the operations described below, using a block of four consecutive locations beginning at the address specified by bits 6–35 of location 420 in the executive or user process table (`UP.ULO==:420`). The UPT, which is used when the LUUO is executed from user mode, specifies a user address; the EPT, used for executive LUUOs, specifies an executive address.

In the first two locations, save the program flags and PC in a flag–PC double word; clear bits 13–17 and 31–35 of the flag word; store the instruction opcode and A in bits 18–26 and 27–30 of the flag word, respectively. In the third location, store E in bits 6–35 (clear bits 0–5).



Then load bits 6–35 of the fourth location into PC and continue performing instructions in normal sequence beginning at the location then addressed by PC. If the LUUO is from user mode, the new PC is a user PC; if the LUUO is from exec mode, the new PC is an exec PC. If E is a local AC address, store it in global form (i.e., with a section number of 1).

2.16.2 MUUOs

The actions of MUUOs depend to a considerable degree on the processor and also on which Monitor is in use. These are the MUUO codes.

TOPS–20	104, 040–051, 055–077 in section zero
TOPS–10 except KA10	040–051, 055–077
KA10	040–051, 055–100

MUUOs have considerable flexibility in the way they can alter the operating characteristics of the

---

[77] If a single memory serves as memory number 0 for two KA10 processors, the second processor (with the trap offset) uses unrelocated 140–141 and 160–161 respectively for each instance in which 40–41 and 60–61 are given here. The offset does not apply to user LUUOs because it is assumed that the Monitor would relocate these to different physical blocks.

machine (mode, section). However, the information that governs the alterations is contained in the user process table and is therefore assumed to be under sole control of the kernel program.

The unassigned codes, which are listed in Appendix C, are not MUUOs, but the processor reacts to them in the same way in order to turn control over to the Monitor. (In the KA10 there are minor differences, explained below.) The processor also takes the same action if the program gives a JRST with an undefined function, an instruction that is illegal because of the context in which it is given, an extended instruction with incorrectly formatted accumulators, or code 000. The last is so that control returns to the Monitor should a user program wipe itself out or inadvertently attempt to execute a location that has been cleared.

The rest of this section is devoted to the different ways in which MUUOs are performed. Except in the KA10, all types use locations in the user process table to store similar information. Figure 2.4 shows what information is stored in which locations for each processor type.

2.16.2.1 XKL-1 MUUOs

If the processor is in executive mode, use the executive MUUO block in locations 430–437 of the user process table (`UP.EMO==:430`); otherwise, use the user MUUO block in locations 440–447 of the UPT (`UP.UMO==:440`). Store an image of the MUUO (the Op Code and A) in the left half of location 2 of the MUUO block (offset `UP.UOP`); set bit 35 in this word if the effective–address specified in the operation is global. Store E in location 3 of the block (`UP.UEA`). Store the PC flags, previous context (CAC, PAC, and PCS), and PC in locations 4 and 5 of the MUUO block (offsets `UP.OFL` and `UP.OPC`, respectively). Complete the specification of the MUUO context by setting up the new PCS (previous context section) with the PC section from which the MUUO was executed. Set the processor flags and the Current and Previous AC blocks from the left and right halves of location 6 (`UP.NFL`) of the MUUO block; take the new PC from location 7 (`UP.NPC` of the MUUO block. Processing continues in normal sequence beginning at the location now addressed by PC. The MUUO can change PC from any section to any other.

2.16.2.2 Extended KL10 MUUOs

In locations 424–426 of the user process table, store the same information (as specified above) that is stored in the first three locations of an LUUO block by an LUUO given in a non–zero section, except that, when the MUUO is given in executive mode, also save the previous–context section in bits 31–35 of location 424. Store the process–context word in location 427; this word saves information that partially defines the context in which the MUUO is given and is exactly the information read by a DATAI PAG, (§4.1.5). Complete the specification of the MUUO context by setting up the previous–context flags and clear the rest of the flags to place the processor in kernel mode. Then load PC from bits 6–35 of the appropriate location in a PC list and continue performing instructions in normal sequence beginning at the location then addressed by PC. (Note that the MUUO can change PC from any section to any other.) The new PC is taken from one of the eight locations in the user process table listed here depending upon the mode at the time the MUUO is given and whether or not it is executed as the result of an overflow trap.

Figure 2.4: User Process Table MUUO Configuration

	0–5	6–8	9–12	13–17	18–20	21–23	24–34	35	
4x0									
4x1									
4x2	*MUUO Op Code*		*A*	0				*G*	`UP.UOP==:2`
4x3	0	*E*							`UP.UEA==:3`
4x4	*Old Flags*			0	*CAC*	*PAC*	*Previous Context Section*		`UP.OFL==:4`
4x5	0	*PC of MUUO*							`UP.OPC==:5`
4x6	*New Flags*			0	*CAC*	*PAC*	0		`UP.NFL==:6`
4x7	0	*New PC*							`UP.NPC==:7`

XKL-1 — Executive MUUO Block at 430 (`UP.EMO`); User MUUO block at 440 (`UP.UMO`)

	0–5	6–12	13–17	18–26	27–30	31–35
424	*Flags*		00	*MUUO Op Code*	*A*	00/*PCS*
425	00	*PC*				
426	00	*E*				
427	*Process Context Word (from* `DATAI PAG,`*)*					

Extended KL10 or TOPS–20 KS10

	0–8	9–12	13–17	18–35
B+0	*MUUO Op Code*	*A*	00	*E*
B+1	*Flags*		00	*PC*
B+2	*Process Context Word*			

Single Section KL10 with TOPS–20 Release 3. B = 425

KS10 with TOPS–10 or KL10 with TOPS–20 Release 1 or 2. B = 424

	0–8	9–12	13–17	18–35
424	*MUUO Op Code*	*A*	00	*E*
425	*Flags*		00	*PC*

KI10

HRMF-UPTMUUO.TEX

Mode	*Execution*	*Location*
Kernel	No trap	430
Kernel	Trap	431
Supervisor	No trap	432
Supervisor	Trap	433
Concealed	No trap	434
Concealed	Trap	435
Public	No trap	436
Public	Trap	437

2.16.2.3 Single–section KL10 MUUOs

With either the TOPS–20 or TOPS–10 Monitor, MUUOs store the same information and take the same action, but they use a different set of three locations in the user process table. In the first location store the instruction code, A, and the effective–address E in bits 0–8, 9–12, and 18–35, respectively, and clear bits 13–17 (this is the same information as that stored by an LUUO given in section zero); save the flags and PC in a PC word in the second location; and save the process–context word in the third location. Then set up the flags and PC according to the contents of the appropriate location in a PC word list and continue performing instructions in normal sequence beginning at the location then addressed by PC. The PC word list occupies the same area as the PC list for an extended KL10, and it is organized and used (with respect to mode and trap) in the same way.

There are no restrictions on the manner in which the new PC word of an MUUO can set up the flags. It can switch the processor from any mode to any other.

2.16.2.4 KS10 MUUOs

The PC or PC–word list contains only four entries for executive and user modes, in the locations corresponding to the kernel and concealed modes as given above—the supervisor and public locations are not used. The process–context word for the KS10 is that read by an RDUBR (§4.2.5). Otherwise, with TOPS–20 an MUUO is performed in the same way as in an extended KL10, and with TOPS–10 it is performed in the same way as in a single–section KL10 running under TOPS–10.

2.16.2.5 KI10 MUUOs

An MUUO is performed in exactly the same way as on a single–section KL10 with the TOPS–10 Monitor, except that it does not store a process–context word (only two words of information are stored, in locations 424 and 425). Note that the trap locations in the PC–word table are used for either overflow or a page failure.

2.16.2.6 KA10 MUUOs

MUUOs and unassigned codes,[78] regardless of mode, perform exactly the operations given above for an LUUO, with the exception that MUUOs use unrelocated 40–41 and unassigned codes use unrelocated 60–61 (140–141 and 160–161 for a second processor). Note that, in executive mode, LUUOs and MUUOs act identically.

The important point is that an MUUO or unassigned code results in executing an instruction in an unrelocated location; i.e., an instruction under the control of the Monitor. This would most likely be a jump that leaves user mode, saves the PC word, and enters a routine to interpret the MUUO configuration. In the instruction descriptions, any reference to events resulting from execution by an MUUO should be taken to also include the unassigned and illegal codes.

2.17 KS10 Input–Output Instructions

Unlike earlier processors, the KS10 has no special format for IO instructions. Instead, instructions are simply those that handle the peripheral equipment, the console, and memory status—although, for consistency with earlier processors, they have 1s in the left three bits. KS10 IO instructions are oriented toward Unibus–type devices, because all peripheral equipment in a DECSYSTEM–2020 is handled through Unibus adapters. There are twelve of these instructions, six each for manipulating full words and bytes, described here in terms of their general behavior in handling external devices. Information about external devices—individual instruction descriptions, IO addresses, etc.—is given in the device documentation (however, memory status is defined in §4.2.8).

NOTE

> Ordinarily, the user has no use whatever for the instructions described in this section. In almost all cases, input and output are handled by the Monitor in response to user requests employing MUUOs and various software formats. For information on user procedures to invoke Monitor handling of user IO requirements, the reader should refer to the appropriate Monitor Calls manual.
>
> Programmers who do handle their own input–output should note that, unless otherwise specified, all instructions described in the remainder of this manual are In–Out instructions, which are affected by the timeshare instruction restrictions. Namely, an instruction of this type cannot be performed by a user program unless User In–Out is set. Any In–Out instruction that violates this restriction does not perform the functions given for it in the instruction description. Instead it executes as an MUUO.
>
> The system instructions discussed in chapters 3 and 4 for the various processors are subject to the same restriction as In–Out instructions. This restriction will not be mentioned in the instruction descriptions, because it applies to *all* instructions in this section.

---

[78] Codes 247 and 257, although not assigned as specific instructions, are nonetheless not regarded as "unassigned codes". They execute as no–ops unless implemented by special hardware.

As in all instructions, the processor does an effective–address calculation; but, for the In–Out instructions, it ignores the result and recomputes an effective IO–address beginning with the I, X, and Y parts of the instruction word. The IO–address specifies an IO register in some Unibus device or in the console or memory controller. For notational convenience, this manual will refer to this effective IO–address also as E. An IO–address is analogous to an extended virtual address in that it has a fundamental length of thirty bits, but not all of its bits are implemented in a given processor. In a KS10 IO–address, the right eighteen bits are the register address and the left twelve are the controller number, of which only four bits are implemented. An IO–address, thus, has this format:

0	C	Register Address
0 13	14 17	18 35

where C is the controller number and bits 0–13 must be zero. Of the sixteen possible controller numbers, only three are used at present: 0 addresses the console and the memory controller, 1 addresses Unibus adapter 1, and 3 addresses Unibus adapter 3. These are the presently allowed IO addresses; no others can be used.

Controller	*Register Address*	*Specifies*
0	100000	Memory status
0	200000	Console (microcode only)
1	400000–777777	Adapter 1 Unibus registers
3	400000–777777	Adapter 3 Unibus registers

The IO address calculation is like an effective–address calculation in which the result can be global; i.e., can have more than eighteen bits. When the result is an 18–bit local register address, it is automatically interpreted as specifying controller 0. The calculation is limited to one level of indirection or indexing or both, and any intermediate result that is used as a memory address must be local (since the KS10 is confined to section zero).

> If there is no indexing or indirection, the IO address is simply Y.
>
> If there is indexing only and the left half of XR is negative, the IO address is the local sum of Y and XR right.
>
> If there is indexing only and XR is positive, the IO address is the global sum of Y and XR (but remember that bits 0–13 must be zero).
>
> If there is indirection only, the IO address is the contents of location Y.
>
> If there is both indexing and indirection, the IO address is the contents of the location specified by the sum of Y and XR right.

Note that an index register can supply the entire IO address, but it can also be used to supply only the controller number when Y is the register address. This latter technique is useful for employing common code for both adapters.

BSIO Bit Set IO

714	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the word read from IO register E, set bits corresponding to 1s in AC and write the result back in register E.

BCIO Bit Clear IO

715	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the word read from IO register E, clear bits corresponding to 1s in AC and write the result back in register E.

RDIO Read IO

712	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of IO register E into AC.

WRIO Write IO

713	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Write the contents of AC into IO register E.

TIOE Test IO Equal

710	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If all bits of IO register E corresponding to 1s in AC are zero, skip the next instruction in sequence.

TION Test IO Not Equal

711	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If not all bits of IO register E corresponding to 1s in AC are zero, skip the next instruction in sequence.

BSIOB Bit Set IO Byte

724	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the byte read from IO register E, clear bits corresponding to 1s in AC bits 28–35 and write the result back in register E.

BCIOB Bit Clear IO Byte

725	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In the byte read from IO register E, clear bits corresponding to 1s in AC bits 28–35 and write the result back in register E.

RDIOB Read IO Byte

722	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of IO register E into AC bits 28–35. Clear AC bits 0–27.

WRIOB Write IO Byte

723	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Write the contents of AC bits 28–35 into IO register E.

TIOEB Test IO Equal, Byte

720	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If all bits of IO register E corresponding to 1s in AC bits 28–35 are zero, skip the next instruction in sequence.

TIONB Test IO Not Equal, Byte

721	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If not all bits of IO register E corresponding to 1s in AC bits 28–35 are zero, skip the next instruction in sequence.

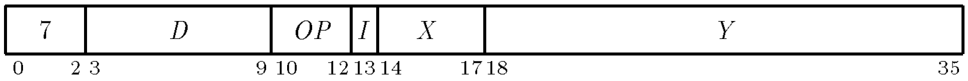
Unibus devices generally have data registers and control/status registers. Frequently, a single IO address specifies two registers, one for reading and one for writing. A control register and a status register in a device usually have the same address and also have bits in common; i.e., information loaded into some of the control bits can be read as status. Ordinarily a device is set up by loading or adjusting individual bits of its control register. Data can then be read or written, and the state of the device can be determined by reading status or testing individual status bits. Complete information about the characteristics of each device is given in the device documentation.

Giving an IO address for a register that does not exist produces a page–fail trap (§4.2.3, §4.2.4).

2.18 Pre–KS10 Input–Output Instructions

In the KL10 and earlier processors, the input–output instructions control the movement of information to and from the peripheral equipment and perform many system–oriented operations within the processor; i.e., management of the internal devices, which in the KL10 are connected to the E bus.

An instruction in the In–Out class is designated by 111 in bits 0–2; i.e., its leftmost octal digit is 7. In this section these instructions are shown like this:



The *OP* field, bits 10–12, selects one of eight IO instructions, which are described here in terms of their general behavior in handling external devices. In the descriptions that follow, the *OP* field is expressed as a two–digit octal number, even though there are only three bits. The octal values are all multiples of 4 in the range 00 to 34. This notation is used so that the value expressed in *OP* can be related to the octal representation of the entire instruction. For example, the CONSO instruction is described as having “34” in the *OP* field; the octal for CONSO is 700340,,0.

The *D* field addresses the device that is to respond to the instruction. The format allows for 128 device codes. The *D* field is expressed in three octal digits, but always as a multiple of 4. Thus, the octal for the device code can be read from the octal digits. For example, the octal for BLKI 124,400 would be encoded as 712400,,400; the *OP* field for BLKI is 0.

The KL10 uses the first six device codes (000–024) for internal devices (the KI10 uses the first three, the KA10 the first two). In instruction descriptions for individual devices, the instruction and device codes are combined into a single, 5–digit code for bits 0–12. Codes for the internal devices are included in the tables in Appendix A.1, but all information about external devices—device codes, individual instruction descriptions, etc.—is given in the device documentation.[79] Bits 13–35 are the same as in all other instructions: they are the *I*, *X*, and *Y* parts, which are used to calculate an effective–address, set of conditions, or mask to be used in the execution of the instruction.

---

[79] Electrical and logical specifications of the IO bus are given in the interface manual.

NOTE

Ordinarily, the user has no use whatever for the instructions described in this section. In almost all cases, input and output are handled by the Monitor in response to user requests employing MUUOs and various software formats. For information on user procedures to invoke Monitor handling of user IO requirements, the reader should refer to the appropriate Monitor Calls manual.

Programmers who do handle their own input–output should note that, unless otherwise specified, the instructions described in this section are affected by the timeshare instruction restrictions. Namely, an instruction of this type cannot be performed by a user program unless User In–Out is set. Any In–Out instruction that violates this restriction does not perform the functions given for it in the instruction description. Instead it executes as an MUUO.

In the KI10 and KL10, In–Out instructions using device codes 740 and above can be performed by user–mode programs without restriction. Also, In–Out instructions are restricted in supervisor mode, because In–Out is normally handled in kernel mode.

The system instructions discussed in chapters 3 and 4 are subject to the same restrictions as IO instructions. This restriction will not be mentioned in the instruction descriptions, because it applies to *all* instructions in this section.

CONO Conditions Out

7	D	20	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

Set up device D with the effective initial–conditions E.[80] The number of condition bits in E that are actually used depends on the device.

CONI Conditions In

7	D	24	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

Read the input conditions from device D and store them in location E. The number of condition bits stored depends on the device; the remaining bits in location E are cleared.

DATAO Data Out

7	D	14	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

Send the contents of location E to the data buffer in device D and perform whatever control

---

[80] E will always be regarded as being bits 18–35, even though it is actually placed on both halves of the bus and many devices receive the information from the left half.

operations are appropriate to the device.

The amount of data actually accepted by the device depends on the size of its buffer, its mode of operation, etc. The original contents of location E are unaffected.

DATAI Data In

7	D	04	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

Move the contents of the data buffer in device D to location E and perform whatever control operations are appropriate to the device.

The number of data bits stored depends on the size of the device buffer, its mode of operation, etc. Bits in location E that do not receive data are cleared.

CONSZ Conditions In and Skip if Zero

7	D	30	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

Test the input conditions from device D against the effective–mask E. If all condition bits selected by 1s in E are 0s, skip the next instruction in sequence.

If the device supplies more than 18 condition bits, only bits 18–35 are tested.[81]

CONSO Conditions In and Skip if One

7	D	34	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

Test the input conditions from device D against the effective–mask E. If any condition bit selected by a 1 in E is 1, skip the next instruction in sequence.

If the device supplies more than 18 condition bits, only bits 18–35 are tested.[81]

BLKO Block Out

7	D	10	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

BLKI Block In

7	D	00	I	X	Y
0 2	3 9	10 12	13	14 17	18 35

---

[81] Conditions reported in bits 0–17 can be tested by reading them with a CONI and then using a test instruction (§2.7).

Add 1 to each half of a pointer[82] in location E, and place the result back in E. Then perform a data IO instruction in the same direction as the block IO instruction, using the right half of the incremented pointer as the effective–address. If the given instruction is a BLKO, perform a DATAO; if it is a BLKI, perform a DATAI.

The remaining actions taken by this instruction depend on whether it is executed as a priority interrupt instruction.

> *Not as an Interrupt Instruction*: If the addition has caused the count in the left half of the pointer to reach zero, go on to the next instruction in sequence. Otherwise skip the next instruction.
>
> *As an Interrupt Instruction*: If the addition has caused the count in the left half of the pointer to reach zero, execute the instruction in the second interrupt location for the level. Otherwise dismiss the interrupt and return to the interrupted program.

It is not expected that block instructions will be of any use in a DECSYSTEM–20. For compatibility, however, the address supplied by the pointer is taken to be in the local section.

Notes: A block IO instruction is effectively a whole In–Out data handling subroutine. It keeps track of the block location, transfers each data word, and determines when the block is finished.

Initially the left half of the pointer contains the negative of the number of words in the block and the right half contains an address 1 less than that of the first word in the block.

The above eight instructions differ from one another in their total effect, but they are not all different with respect to any given device. A BLKO acts on a device in exactly the same way as a DATAO—the two differ only in counting and other operations carried out within the processor and memory. Similarly, no device can distinguish between a BLKI and a DATAI; and a device always supplies the same input conditions during a CONI, CONSZ, or CONSO whether the program tests them or simply stores them.

Hence, the eight instructions may be categorized as being of four types, represented by the first four instructions described above. Moreover, a complete treatment of the programming of any external device can be given in terms of these four instructions, two of which are for input and two for output.[83]

Every device requires initial conditions; these are sent by a CONO, which can supply up to eighteen bits of control information to the device control register. The program can determine the status of the device from up to thirty–six bits of input conditions that can be read by a CONI (but only the rightmost eighteen bits can be tested by a CONSZ or CONSO). Some input bits simply reflect initial conditions sent by a previous CONO, others are set up by output conditions but are subject to subsequent adjustment by the device, and still others may have no direct connection with output conditions.

---

[82] In the KA10 incrementing both halves of the pointer is effected by adding 1000001_8 to the entire register (and a carry can therefore go from the right half into the left).

[83] The word "input" used without qualification always refers to the transfer of data from the peripheral equipment into the processor; "output" refers to the transfer in the opposite direction.

Data is moved in and out in bytes of various sizes or in full 36–bit words. Each program transfer between memory and a device data buffer requires a single DATAI or DATAO. Every device has a CONO and CONI, but it has only one data instruction unless it is capable of both input and output. A DATAI that addresses an output–only device simply clears location E. On the other hand, a DATAO that addresses an input–only device is a no–op. When the device code is undefined or the addressed device is not in the system, a DATAO, CONO, or CONSO is a no–op; a CONSZ is an unconditional skip; and a DATAI or CONI clears location E.

The general effects of the IO instructions are as given above, but a single instruction varies in its individual effects from one external device to another. For KI10 and KA10 internal devices, the instructions still have the same general effects and have the same relation to one another; but, again, they vary in individual effects that are documented in the descriptions in Chapter 4.

The situation is quite different, however, with respect to KL10 internal devices. For example, a DATAI PAG, is really a DATAI—it reads information from the pager; but a DATAI CCA, is not a DATAI—it sweeps through the cache invalidating all pages, and it has its own mnemonic, SWPIA. The instruction BLKI PI, has no connection whatever with DATAI PI, because it is not a block instruction at all—it is actually the instruction RDERA, which reads the error address register. In other words, although some of the IO instructions for KL10 internal devices are equivalent in general terms to the same instructions for external peripherals, many of them are uniquely defined operations that bear none of the standard relationships to the typical case or to other instructions using the same device code (in some cases even when for the same device). When a unique mnemonic is assigned for an instruction, the form using a device mnemonic is given at the right end of the top line in the description.

2.19 User Programming

The preceding sections define the machine–language characteristics of the system from a user point of view. However, efficient and effective use of the system is affected greatly by the software; the user should therefore consult the appropriate Monitor manual, especially for the employment of the Monitor for input–output. For convenience, those rules that the user must observe and that are the result of XKL-1, KL10, and KS10 hardware characteristics are listed here.

- If an area of memory is write–protected (e.g., for a reentrant program shared by several users), do not attempt to store anything in it. In particular, do not execute a JSR or JSA into a write–protected page.
- Use the MUUO codes only in the manner prescribed in the Monitor manual. Unless they are prescribed for special circumstances, the unassigned codes should not be used. Code 000 is illegal in any any circumstance.
- Do not use HALT (JRST 4,) unless you want your program stopped.
- Always be aware of the context in which the program is running, and make sure to use only operations appropriate to that context. In particular, be familiar with which forms of the JRST instruction are legal in which circumstances, as explained in §2.9.4. JRST functions for handling interrupts are legal when IO is legal.

- Unless User In–Out is set, do not give any IO instruction with device code less than 740 (any at all in the XKL-1, KS10 or KA10). The program can determine if User In–Out is set by examining bit 6 of the saved flags.
- If your public program has the use of concealed programs, do not reference a location in a concealed page for any purpose except to fetch an instruction from a valid entry point; i.e., a location containing a PORTAL (JRST 1,).
- In an extended processor, do not use JEN in a non–zero section. Also be aware of the differences between running in section zero and in other sections. Differences appear both in the execution of instructions, such as JSR and JSP, and in the format and handling of such quantities as index registers, indirect address words, and stack and byte pointers.
- Make sure to format the accumulators correctly in string instructions (§2.12)

The user can give a JRSTF or XJRSTF, but a 0 in bit 5 of the PC word or flag word does not clear User (a program cannot leave user mode this way); and a 1 in bit 6 does not set User In–Out, so the user cannot void any of the instruction restrictions himself. Note that a 0 in bit 6 will clear User In–Out, so a user can discard his own special privileges. Similarly a 1 in bit 7 sets Public, but a 0 does not clear it, so a public program cannot enter concealed mode this way.

Many hardware characteristics, however, are actually transparent to the user; in particular, the whole paging setup is invisible. Although the hardware allows for user virtual address spaces that are scattered or very large (even larger than available physical memory), the actual constraints will be dictated by the particular Monitor and the system manager. Most TOPS–10 Monitors enforce a two–segment virtual address space that mimics the one imposed by the KA10 hardware. In any case, the user must write a sensible program which can be handled easily and cheaply by the system; if he uses addresses a few to a page all over memory, his program can be run but will require a much larger amount of space than necessary or cause excessive page swapping.

The basic idea is to localize everything as much as possible. Do not spread parts of the program out through the address space, leaving gaps. Put together whatever will be used together: divide a large program into smaller segments and with each group of instructions put whatever pointers, data locations, and the like that will be used with it. Group together subroutines that are called by the same programs. If a package is to be used at all frequently, take advantage of the various features (e.g., a core map) provided by the Digital software to determine just how the package was assembled and, if necessary, revise it to reduce the working set of pages.

The rules given above apply generally to all systems, but there are minor differences from one to another; a user who wishes to write programs to run on more than one type of processor must be aware of whatever incompatibilities exist. For example, the interrupt–handling JRST functions are legal in user IO mode except on the KI10, where they are restricted to kernel mode. Because of the more restricting JRST decoding in the earlier processors, the XKL-1, KL10, and KS10 have more functions, and they produce quite different effects when given in a KI10 or KA10 program. The matter of unassigned codes works both ways with respect to different processor models: instructions added in a later machine use codes unassigned in earlier machines, but the codes for the software double–precision floating–point instructions are unassigned in later machines. Unassigned codes that correspond to implemented instructions in other machines should be used only if the software includes interpretive routines for them, but wherever possible they should be avoided because of the severe time penalty.

Chapter 3

TOAD-1 System and XKL-1 Processor Operations

Note

Most of the material in this chapter has settled down. The areas still changing are those concerning multi–processor configurations, clustered systems, and, to a lesser degree, TDBoot.

Ralph Gorin controls the content; he would appreciate your comments and suggestions.

The information presented in this chapter is primarily for XKL's own hardware designers and systems programmers. This information documents an important component of the TOAD-1 System design and provides the information necessary to the authors of the operating system, diagnostics, and other software. This information is also germane to anyone who wishes to write his own operating system; it may be needed by users who wish to handle their own I/O or by programmers in a situation where all the facilities of a system are dedicated to a single large task.

Warning!

> XKL-1 processor functions are implemented in microcode (which can be revised much more easily than hardware). Although the user operations described in Chapter 2 are deliberately kept as compatible as practicable with other PDP–10 processors, XKL LLC will change XKL-1 processor microcode whenever such a change will result in greater speed, efficiency, or effectiveness. Therefore, anyone writing system software should be sure to obtain the most recent version of this documentation. Before embarking on any project as enormous and critical as an operating system, be sure to check with XKL LLC for any changes not yet documented and for such system development tools as might be provided to customers undertaking such a project.
>
> This warning applies also to the various subsystems connected to the TOAD-1 System's backplane bus. They too are heavily dependent on microcode, which code may change for any of the reasons stated above.
>
> Locations in MemA and in NVRAM identified in this manual are **not** part of the architectural specification of the TOAD-1 System. They are subject to change without notice.

Programming for the system as a whole is programming in executive mode. Only the executive program is without instruction restrictions. All other programs labor under instruction restrictions.

The amount of useful work done by the system depends on how efficiently and effectively the executive manages the physical resources of the system. These resources include the processor, memory, input–output devices, the file system, and the bandwidth of the paths between various components. The executive selects which process to run next. It manages the working sets of the various processes, responding to their changing needs. The executive reacts to error situations and even to unacceptable behavior on the part of a user. The executive accomplishes these objectives by handling all in–out for the system and setting up user page maps, trap locations, interrupt locations, etc. for itself and for the users. The executive handles user accounts, passwords, and level of privileges. It controls access to all system resources.

The activities of an operating system, particularly as they are implemented in the XKL-1 processor, are the topics of this chapter. Of course the system programmer must be fully familiar with the material presented in the earlier chapters. The programmer must understand the TOAD-1 System architecture as presented in Chapter 1 and must be totally conversant with the instruction set, including the various modes of JRST, MUUOs, and I/O instructions. Executive–mode extensions of the instruction set (e.g., PXCT and others) are discussed here.

3.1 TOAD-1 System Backplane Bus

The high–speed backplane bus carries information between the various components of the system. A **bus transaction** involves, first, a device that requests the bus; second, a grant, in which the requesting device is allowed control of the bus for a time; and, third, one or more bus cycles directed by the requestor (also known as the source) to another device—the target (or destination)—after which the requestor removes its request for the bus. A bus transaction is strictly one–way: information moves from the requestor to the target. The bus cycles from which bus transactions are composed are of several different types, identified by name.

A **semantic transaction**, that is, a meaningful exchange of information, is completed in either one or two bus transactions. For example, a write (to e.g., a memory) takes only one bus transaction, because the requestor supplies the address and the data. However, a read (from e.g., a memory) requires two bus transactions: the original requestor supplies the read command and address to the target device in the first bus transaction; subsequently, the former target device now acts as the requestor and supplies data to its target (the original requestor).

Of course, any semantic transaction that can be completed in one bus transaction is performed that way. To maximize system throughput, semantic transactions that involve a variable delay (e.g., the time while a memory finds the data that was requested) are performed as two bus transactions: a request and a return. This organization of semantic transactions into one or two bus transactions helps to speed bus throughput by eliminating the delays inherent in turning the bus around: in the TOAD-1 System backplane bus the initiator is always the source of commands, addresses, and data; the target receives them. The source and target of the request portion of a two–part transaction interchange roles for the return bus transaction.

A semantic transaction that involves two bus transactions allows any number of unrelated bus transactions to occur after the request and before the corresponding return. A device may initiate a transaction (i.e., make a request) even while it has incomplete transactions pending. Requests directed by one source to different targets are answered independently, so the responses may come in some order other than that of the requests. Multiple requests by one source to the same target will (if accepted) produce responses by the target in the same order as the requests were received. As the target of a transaction, each device is required to honor at least one (appropriate) request at a time, though some may honor more. When unable to handle an appropriate request, a target will respond "busy". When presented with a request of an inappropriate type (e.g., an Interrupt_Request directed to a memory module), the target device may ignore the request. To avoid deadlock, designers of devices are strongly urged to accept immediately (without being busy) the Word_Return and Line_Return responses whenever they appear, because these are responses to requests initiated by the target device. During normal operation, the XKL-1 processor is never busy to a request.

A source that receives a busy response is expected to retry the request some number of times before abandoning the request.

Any device that initiates two–part transactions will time–out a transaction that is not completed by an appropriate return within a specified time period. Such time–outs signify errors.

The backplane bus is quite short; the likelihood of a field becoming corrupted is quite low.[1] Therefore, no error checking is included with the bus transactions. Individual modules, notably memory, will do their own error checking before placing data on the bus; the bus provides a method by which error conditions may be signalled.

Bus arbitration is centralized. To gain access to the bus, a module must assert BUS_RQ[n] (where n is the physical slot number occupied by the module) and keep it asserted until it begins the last cycle of its transaction. Central bus arbitration will grant service by raising BUS_AK[n] to a module late in the cycle preceding the one at which the module may begin its transaction.[2] Central

---

[1] Many people would like some assurance regarding this point.

[2] In the implementation of the backplane the n in BUS_RQ[n] and BUS_AK[n] is implicit. BUS_RQ and BUS_AK are dedicated pins on each bus connector; the slot number is determined by the backplane connection from each slot

bus arbitration will present the number of the selected module on the SRC[0:3] lines visible to all modules. In this way, the target device knows which module initiated a request and thus knows to whom to address the return, if needed.

3.1.1 Request Transactions

These are the semantic transactions from which no response is expected; thus, they are accomplished in one bus transaction. However, a device may respond busy at the end of the first bus cycle that selects it as the target, in which case the requestor must be prepared to try again.

Notes: When a request transaction is directed to an empty slot, there will be no indication that an error has occurred.

Word_Write

> Write a word to a target device (e.g., a memory). The source device will place the target device slot number and the in–module address on the bus and perform a cycle of type Word_Write. On the following cycle, the source will provide the contents of the word, in a cycle of type DataW1 (write one data word). If the memory is unable to receive the data, it responds Busy at the end of the cycle that sends Word_Write; the Word_Write must be repeated later. Otherwise, no response is required: the memory is assumed to have accepted the address and data.

Line_Write

> Write a line (eight consecutive 36–bit words on an 8–word boundary) to a target device. (N.b, not all devices support Line_Write; specifically, a memory device will support this cycle type.) The source device will place the target device slot number and the in–module address on the bus and perform a cycle of type Line_Write. On the following four cycles, the source will provide a pair of words on each cycle, along with the cycle type DataW2 (write two data words). If the target device (memory) is unable to receive the data, it responds Busy at the end of the Line_Write cycle; in this case, the source will have to retry the Line_Write. Otherwise, no response is expected. The address sent in the Line_Write cycle must be a multiple of 8. Data words with low–order addresses 0 and 1 are sent on the first DataW2 cycle; these are followed by addresses 2 and 3, 4 and 5, and 6 and 7 in the next three DataW2 cycles.

LongWord_Write

> Write one longword (a pair of words at an even address). The source device will place the target device slot number and an even in–module address on the bus and perform a cycle of type LongWord_Write. On the following cycle, the source device will provide the contents of a double word, in a cycle of type DataW2 (write two data words). If the target is unable to receive the data, it responds Busy at the end of the cycle that sends LongWord_Write; in this case, the LongWord_Write must be repeated later. Otherwise, no response is required: the target is assumed to have accepted the address and data.

Device_Control

> Send control information to a device. The source device places the target device slot number and a control register address (i.e., the target's in–module address) on the bus

---

to the central arbiter.

and performs a cycle of type Device_Control. If the target device is unable to accept a control command at this time, it responds Busy, in which case the source device will have to repeat the Device_Control function. Otherwise, the source performs a second cycle of type DataW1, in which the desired control information is placed on the bus.

Interrupt

Make or withdraw a request for an interrupt. The source device places the target device address, the desired priority level, and an indicator for "Make" or "Withdraw" on the bus and performs a cycle of type Interrupt. If the target device responds Busy at the end of the Interrupt cycle, the source device must repeat the Interrupt cycle. Otherwise, if this is a "Make" request, the target device is obliged to attend to the interrupting device (at some future time); if this is a "Withdraw" request, the target device will understand that a previously requested interrupt has now been satisfied.

3.1.2 Request–and–Return Transactions

These are the semantic transactions that require two bus transactions. They begin when one device makes a request to another; they end when the second device responds (i.e., returns information) to the first. After a request transaction and before the corresponding return transaction, the bus is available for other transactions. In the explanation that follows, the request–and–return transactions are presented in pairs. That is, Word_Read_Request is matched by Word_Read_Return, etc. The return transaction is accomplished by a cycle of type DataR1 or by cycles of the DataR2 type. The name of the bus cycle specifies particular signals on the backplane bus; the name of the semantic transaction denotes the meaning of the particular bus cycle or cycles.

Word_Read_Request

Request that a word be read from the target device. The source device will put the target device and its in–module address on the bus and perform a Word_Read_Request cycle. If the target device responds Busy at the end of the Word_Read_Request cycle, the source must repeat the Word_Read_Request. Otherwise, the source device may expect, in due course, to be the target of a Word_Read_Return transaction.

Word_Read_Return

Transmit a word from a device (memory) in response to a previous Word_Read_Request. The device will target the original source device, place the data word on the bus and perform a cycle of type DataR1 (read one data word). If the requested word has a parity error, the error will be signalled by putting a 1 on the MISC[7] line. Although it is generally bad form for the target to respond Busy at the end of the DataR1 cycle, it may do so, in which case the source must repeat the DataR1 cycle.

Line_Read_Request

Request that a line (eight consecutive 36–bit words on an 8–word boundary) be read from the target device. The source device will put the target device slot number and its in–module address on the bus and perform a Line_Read_Request cycle. If the target device responds Busy at the end of the Line_Read_Request cycle, the source must repeat the cylce. Otherwise, the source device may expect a Line_Read_Return transaction. Bits 33–34 of the given address specify which pair of words to return first.

Line_Read_Return

Transmit a line (eight consecutive 36–bit words on an 8–word boundary) from a device (memory) in response to a previous Line_Read_Request. The device will target the original source device, place the first two data words on the bus, place the line index (corresponding to bits 33–34 of the address) on the bus in MISC[1–2], and perform a cycle of type DataR2 (read two data words). If either word has a parity error, that will be signalled by putting a 1 on the MISC[7] line.[3] Although it is generally bad form for the target to respond Busy, it may do so at the end of the first DataR2 cycle, in which case the source must repeat the DataR2 cycle. Otherwise, the source device will use the next three cycles (also of type "DataR2") to complete the transmission of the line of data. The line index is incremented (modulo 4) in each of the cycles to effectively count through the four double word addresses in the line.

LongWord_Read_Request

Initiate the read of a longword (two words). The initiating device (the source device for this cycle) will place the target device slot number and the (even) in–module address on the bus and perform a cycle of type LongWord_Read_Request. After a variable delay to retreive the requested data, the target device, now acting as a source, will respond with a cycle of type DataR2 (return doubleword), as explained in LongWord_Read_Return.

LongWord_Read_Return

Transmit a longword in one bus cycle from a device (memory) in response to a previous LongWord_Read_Request. The transmitting device will target the original source device, place the data longword on the bus and perform a cycle of type DataR2 (return two data words). If either of the requested words has a parity error, the error will be signalled by putting a 1 on the MISC[7] line. Although it is generally bad form for the target to respond Busy at the end of the DataR2 cycle, it may do so, in which case the source must repeat the LongWord_Read_Return transaction.

Device_Status_Request

Request that a word of status be read from a device. The source device places the target device and the address of its status register (i.e., in–module address) on the bus and performs a cycle of type Device_Status_Request. If the target responds Busy at the end of the Device_Status_Request cycle, the source must repeat the cycle. Otherwise, the target device is expected to respond (eventually) with a Device_Status_Return transaction.

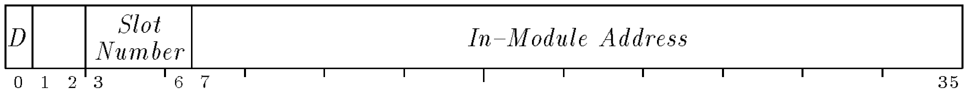
Device_Status_Return

The previously requested status is returned. The source device (the target of a previous Device_Status_Request command) names the original requestor (the source of the previous Device_Status_Request) as the target of this cycle. The requested data is put on the bus and a cycle of type DataR1 is performed. The MISC[7] line (parity error) is driven to logic 0 by the source device (no error). Although it is generally bad form for the target to respond Busy at the end of the DataR1 cycle, it may do so, in which case the Device Status Return transaction must be repeated.

---

[3] Further information about parity errors can be obtained by a Device Status Request to the memory (§3.11).

Figure 3.1: Bus Address Word



3.1.3 Special Bus Functions

The bus carries the “Reset” signal to all modules. The assertion of Reset will stop or prevent all manner of activity on every module. The removal of Reset will force every module to its power–up state. The TOAD-1 System power system is arranged to assert Reset for several seconds following power turn–on so that the power supplies can reach stable levels before any module commences its power–up sequencing.

The bus carries the “PFail” signal to all modules. This signal is asserted when the AC power to the system is insufficient for continued operation. The PFail signal is also asserted if the temperature in the card cage exceeds the Thermal Warning limit. (At the slightly higher Thermal Shutdown temperature, the DC power is removed.) Software/Firmware in all modules will begin their power fail sequences in response to PFail.

The “Need DC” signal on the bus may be asserted by any device on the backplane. The presence of “Need DC” informs the power controller that one or more devices still needs power to complete its power fail sequence. When no device is asserting “Need DC”, the power controller will turn off all system DC power, thus conserving the battery life. “Need DC” can be asserted by the CPU by using the WCTRLF instruction.

The “System Active” signal on the bus may be asserted by any device on the backplane, though it is customary for the CPU to control it. When asserted briefly, the “System Active” signal sets a one–shot, which resets itself after a 15 ms delay. While set, the one–shot turns on the yellow “System Active” light on the front panel. The WCTRLF instruction is provided to trigger the one–shot. During normal system operation, the operating system will attempt to keep the System Activity light on by periodically executing this instruction.

3.1.4 XKL-1 Bus Operation Instructions

The XKL-1 processor does not use traditional input–output instructions. To affect devices on the backplane bus, the processor has two instructions that are described below. These instructions all begin by computing E in the usual way and then reading the *bus address word* (BAW) contained in E. A bus address word has the format depicted in Figure 3.1.

The significance of the fields in the bus address word is as follows:

D `BA%DEV==:1B0` The “device” bit. When set to 1, this bit signifies that the “Device Status Request” and “Device Control” bus cycles will be used by PMOVE and PMOVEM, respectively; in other words, the device will be treated as an IO device. When D is 0, the bus cycle types “Word Read Request” and “Word Write” will be used by PMOVE and PMOVEM, respectively; that is, the device will be treated as memory.

Slot Number `BA%SLT==:17B6` The physical slot number of the device being addressed. Slot numbers in the range 1–17 (octal) are allowed. Slot 0 does not exist. (When the CPU references slot 0, it gets the CPU's private ROM.) The initial configuration of the TOAD-1 System provides just seven backplane slots, 1–7.

In–Module Address `BA%IMA==:3777777777` The 29–bit in–slot physical address of the location to be affected.

In addition to the two instructions presented here, ordinary instructions may be used to affect devices by creating an appropriate pager mapping.

The instructions to affect devices are these two:

PMOVE Physical Move from Memory or Device

704	A	I	X	Y
0 8	9 12	13	14 17	18 35

Read the contents of a memory location or its cached representation, or read a device register, to AC, bypassing the pager.

Using the cache and pager as usual, compute E and read a bus address word at location E.

The given bus address is cacheable if all of the following are true:

- D is zero in the bus address word, and
- the pager is on, and
- the CST base address is not zero, and
- the slot number in the bus address word corresponds to a memory, and
- the CST entry for the page containing this address specifies that the page is cacheable.

Perform one of the actions described below, depending on whether or not the given bus address corresponds to a cacheable memory page:

- If the given bus address is cacheable, read the cached representation of the given address from the cache and store the data in AC. If the cache does not already contain a representation of this address, load the cache with the memory line that includes this address and copy the selected word to AC.
- If the given bus address is not cacheable, transmit the bus address word on the backplane. If D is zero, use a Word_Read_Request cycle; if D is one, use a Device_Status_Request cycle. A normal response is a backplane transaction of the "DataR1" type; the data transmitted with the return will be copied to AC. If the given bus address does not match any device or a device's in–module address, there will be no response. After a timeout, the CPU will perform a page trap in which the page–failure word will indicate a bus timeout.

This data movement portion of this instruction bypasses the pager. No update to the CST will be performed; if the Monitor requires a CST update, it must update the CST itself.

This operation guarantees that the data retrieved is current. If the target page is not cacheable, this instruction will not cause data to appear in the cache.

PMOVEM Physical Move to Memory or Device

705	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Write the contents of AC to memory, to the cached representation of memory, or to a device register, bypassing the pager.

Using the cache and pager in the usual way, compute E and read a bus address word from location E.

The given bus address is cacheable if all of the following are true:

- D is zero in the bus address word, and
- the pager is on, and
- the CST base address is not zero, and
- the slot number in the bus address word corresponds to a memory, and
- the CST entry for the page containing this address specifies that the page is cacheable.

Perform one of the actions described below, depending on whether or not the given bus address corresponds to a cacheable memory page:

- If the given bus address word specifies a cacheable memory page, copy the data in AC to the cached representation of the specified memory address, marking the corresponding cache line as “modified”. If the cache does not already contain a representation of this address, read the memory line that includes this address into the cache and then copy the AC data into the cached representation of the specified address.
- If the given bus address does not specify a cacheable memory page, transmit the bus address word on the backplane. If D is zero, use a Word_Write cycle; if D is one, use a Device_Control. In the following cycle, transmit a copy of the data in AC on the bus with cycle type DataW1. No response is expected from the bus. If the given bus address does not match any device or a device’s in–module address, there will be no indication of error.

The data movement component of this instruction bypasses the pager. No update to the CST will be performed; if the Monitor requires a CST update, it must perform the CST update itself.

This operation guarantees that the cache will contain a current representation of the intended contents of memory. If the target page is not cacheable, this instruction will not cause data to appear in the cache.

3.1.5 Communication Between the Processor and Devices

Communication between the XKL-1 central processor and a generic peripheral device is effected by two one–way "connections". Each connection is represented by one word in (uncached) storage. The connection word is similar to the flag on a rural mailbox: set by the sender to indicate that the box contains a message, cleared by the receiver to indicate that the message has been removed. As the mailbox, the message may be in many parts (i.e., a list of messages). In contrast to the mailbox analogy, once the sender has set the flag, the sender is prohibited from adding more messages to the box until the first batch has been taken.

More specifically, the CPU and device share a word called "ToDev". When the CPU has work for the device, it creates a list of messages for the device and stores the bus address of the list in ToDev, but only if the present contents of ToDev are zero. After storing in ToDev, the CPU "rings the doorbell" by sending an appropriate device control signal to the device. The device acknowledges the doorbell by reading the contents of ToDev and storing a zero in ToDev. (Then, if the CPU has so requested, the device can acknowledge receipt of the message by sending an interrupt back to the CPU.) At this point, the messages formerly in ToDev are now the responsibility of the device. If the CPU has work for the device and ToDev is non-zero, the CPU appends additional messages in its queue of messages intended for the device, and continues appending such messages until ToDev becomes zero.

In the other direction, the behavior is similar. The device and CPU share a word called FromDev. When the device has information to report to the CPU, it creates a list of messages and stores the bus address of the list in FromDev, but only if FromDev is already zero. After storing in FromDev, the device interrupts the CPU to alert it to the new messages. The CPU responds to the interrupt by reading FromDev and setting it to zero. (If the device so requests, the CPU can send a device control message back to the device indicating that FromDev is now zero.) At this point, the messages formerly in FromDev are now the responsibility of the CPU. If the device has messages for the CPU but FromDev is not zero, the device will append these messages to its queue of pending messages for the CPU, where they will be held until FromDev is zero.

The locations of communication cells are assigned by the CPU; the CPU informs the device of the assigned locations by means of device control messages. If shared communication regions are needed, they are assigned by the CPU, which will inform the device. The authors of monitor device drivers must be aware of the general needs of the device and arrange communication regions appropriately.

To support multi–processor systems, a device must be able to accommodate multiple "ToDev" and "FromDev" communication locations.

A device with specialized queues may have independent "ToDev" and "FromDev" locations associated with each queue.

3.1.6 Identification of Backplane Devices

Every processor, memory, or I/O interface intended for use on the TOAD-1 System backplane bus must adhere to the electrical conventions of the bus, the signalling conventions of the bus, and the module identification convention described here.

Every device shall respond to a Device_Status_Request function at address 0 by returning a main status word. The main status word is intended to identify the general nature of the addressed unit and to report its major status (i.e., ready or not). The identification is by means of bits 0–7 (`DS%TST==:377B7`), the Device Type and Subtype fields, of the main status word. This field is divided into bits 0–2 (`DS%TYP==:7B2`), the Device Type field, and bits 3–7 (`DS%STY==:37B7`), Device Subtype field. Each device description will describe the assigned values for these fields.

3.2 Console

This section describes the XKL-1 processor console operation, the interface between the micro–console and the macro–console, the console command set, and macro–instructions for programming input and output to the console terminal.

> **Notice**
>
> This description is highly specific to the present implementation of the XKL-1 processor and is **not** part of the architectural description of the TOAD-1 System.

The XKL-1 processor has two RS–232 serial communication ports, labeled "Console" and "Auxiliary". These are used to connect a console terminal and an optional diagnostic terminal or modem to the system. As explained in more detail below, the XKL-1 processor attempts to treat both communications ports and their connected terminals identically, so the term "console port" refers to either or both communications ports and "console terminal" to whatever device is connected to either or both ports.

The console terminal and diagnostic terminal are handled at all times by the XKL-1 processor's microcode. The interpretation of characters typed on the console terminal varies according to the state in which the processor microcode has placed the console ports. The major division of states is between "Console" and "Program" states. The transitions of the state of the console port are described in more detail below; they are depicted in Figure 3.2.

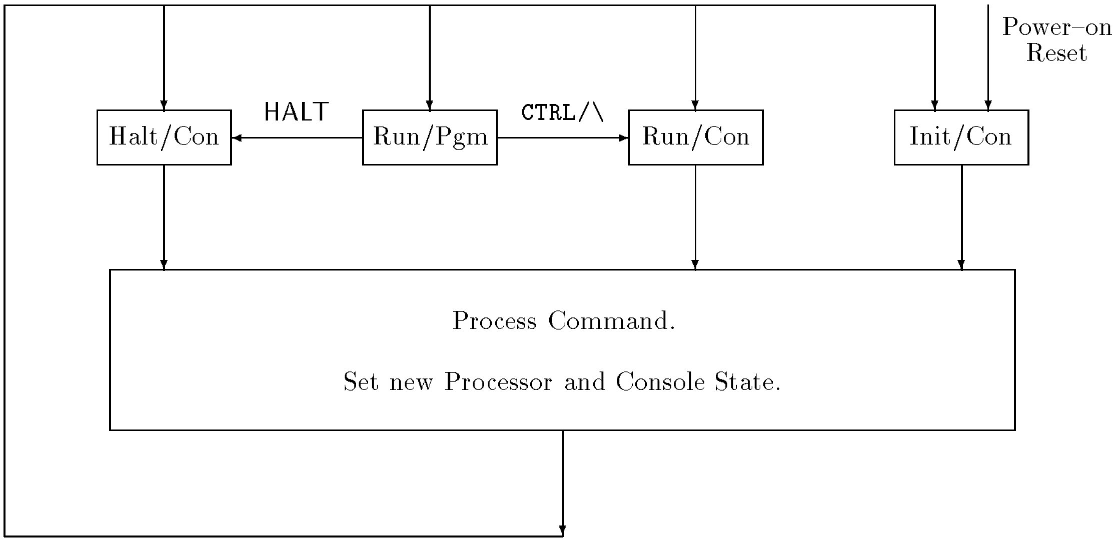
Figure 3.2 depicts the transitions between various states.

In "Program" state, the console port is generally under control of the macro–program executing in the XKL-1.

In "Console" state, one–line commands are received by the micro–console program, which is implemented in the XKL-1 microcode. The micro–console either performs the command itself or it passes the command to the macro–console where it is performed. (The macro–console is implemented by macro–code instructions in the Boot ROM.) The Console state has substates depending on the condition of the macro–program being run by the XKL-1.

The micro–console is activated either by receipt of a "`CTRL/\`" (control backslash) character on the console terminal port or by execution of a HALT instruction by the running program. When activated, the micro–console will collect a command line from characters typed on the console terminal port.

Figure 3.2: Console State Transitions



Commands that begin with a "`.`" (period) are parsed and processed by the micro–console, without resorting to macro–code. The micro–console provides a small set of commands for XKL-1 initialization and control. These commands are available even in the unfortunate circumstance of not being able to execute macro instructions. In micro–console commands, lowercase letters are interpreted as uppercase.

To handle all other commands, the micro–console activates the macro–console (unless it is disabled). Since the XKL-1 may be "running a program" (i.e., executing PDP–10 macro–code instructions) when the micro–console receives a command, the micro–console is responsible for preserving the state of the running program before activating the macro–console. State is preserved by simulating an interrupt of the current program. The current PC, flags, relevant machine state, and a pointer to the command string are saved in a predetermined area of MemA; part of the stored state indicates whether a program was running when the macro–console was activated. Then the pager is turned off and the processor priority interrupt level is raised to 0 (which is higher than any normal program interrupt). Finally, the PC is set to a predetermined location in the Boot ROM.

The macro–console code then parses the command stored in MemA and takes the appropriate action. Generally, the macro–console does not perform commands that alter any machine state unless it can restore the state at the end of the command. If no program was running when the macro–console was started, commands that intentionally alter the machine state are permitted.

Upon completion of a command, the macro–console executes a HALT instruction to return control to the micro–console. The micro–console restores the machine state and PC from where it was saved in MemA. (The macro–console command may have altered the saved state.) If appropriate, the micro–console resumes the execution of the interrupted program.

In detail, the micro–console will enter the macro–console by performing the following steps:

1. Save the state of the current macro–code in MemA:

`AM%MFG`	Flags and context
`AM%MPC`	PC
`AM%MEB`	EBR
`AM%MUB`	UBR
`AM%MCS`	CSB
`AM%MPI`	Highest PI level in progress

2. Set or clear the following flags in `AM%MBT` in MemA:

`MS%RUN`	A program was running if set. (Otherwise, there was no program, the program had never run, or it was halted.)
`MS%VAL`	Program PC is valid if set.
`MS%MCA`	Macro–console was active if set. (This flag is used for debugging when running the macro–console as a program.)
`MS%MCE`	Macro–console is enabled, if set. (This flag is always cleared to zero at entry to the macro–console, thus disabling reentry of the macro–console unless the macro–console is able to progress far enough to set this bit again.)

3. Make the machine enter PI priority level 0. This disables any interrupts at all lower priority levels (all normal levels).
4. Set the EBR and UBR to 10,,0. This is the address of TDBoot's vestigial EBR and UBR. Then turn paging off.
5. Set the PC to a predetermined offset in the entry vector and continue the macro–code, thus starting the macro–console. The following offsets are currently defined:

+3	Initialize the macro–console.
+4	Respond to a program halt. The address at which the HALT occurs will be in `AM%OPC`. The macro–console will examine the state of the Dump, Diagnose, and Boot flags to determine its next actions.
+5	Process the command to which `AM%MCM` points.

If the macro–console executes a PI reset, PI level 0 is cleared (as are all other PI levels). Thereafter, the macro–console is no longer bound to preserve the machine state and update the MemA variables described above. A subsequent HALT will enter the micro–console as if the XKL-1 were running an ordinary program.

Upon a program's execution of a HALT instruction, the XKL-1's microcode will perform one of the following:

- If `MS%NCA` is clear, clear `MS%RUN` in `AM%MBT`. If the macro–console is enabled, start it as described above, using offset +4. If the macro–console is disabled, enter the micro–console to collect CTY input as a micro–console command.

- If `MS%NCA` is set, this is the macro–console returning control to the micro–console. If `MS%MCE` is clear, disable the macro–console. (`MS%MCE` being clear indicates that the macro–console has declared itself to be sick; the macro–console remains disabled until a `.M` command is issued.) Otherwise, restore the machine state from the locations in MemA at which it was saved.

3.2.1 Console State Transitions

The console state is defined by two things: first, whether input is going to the console ("Con", indicated by `CT%CON` set in `AM%CTS`) or to the macro–code ("Pgm", indicated by `CT%CON` clear in `AM%CTS`) and, second, by the condition of the PDP–10 program: either running (`MS%RUN` set), halted (`MS%RUN` clear, `MS%VAL` set), or initialized (`MS%VAL` clear).

3.2.2 Micro–Console Messages

The following are messages from the micro–console.

`System Processor (XKL-1, 1995) Ver - 000000000123` This denotes XKL-1 processor microcode version "123".

`XKL-1%` This micro–console prompt signifies either the "Halt/Con" state or the "Init/Con" state: the macro–code is not running.

`XKL-1>` This micro–console prompt signifies "Run/Con" state: the macro–code is running.

`?CMD` An invalid command letter was entered or a required command letter was missing.

`?ARG` An incorrect number of numeric arguments or a non–numeric argument was entered.

`?PC` No valid macro–code PC exists. (The PC is invalid after a hardware reset, after either a `.I` or `.M` command, or after a variety of macro–console commands.)

`?RUN` The given command is not legal while macro–code is running. (The macro–code can be stopped by the `.H` command.)

`?MCON` A command was entered which would normally be passed to the macro–console, but the macro–console is not enabled. (The `.M` command will enable the macro–console.)

`HALT at` *nnn* The macro–code (the running program) has halted at the indicated PC, *nnn*. The console terminal port is now in "Halt/Con" mode. The micro–console prints this message only if the macro–console is not enabled; otherwise, the macro–console prints its own message.

`?MCON HALT at` *nnn* The macro–console has halted at the location indicated by *nnn* because of an error. (The error condition is signified by `MS%MCA` set and `MS%MCE` clear.) Possible causes include a failure in the basic instruction test. The macro–console is now disabled (until a `.M` command).

`?MCHK` *nnn* `at` *mmm* Micro–code or hardware failure *nnn* at the PC *mmm*. The interpretation of the code *nnn* can be found in the corresponding micro–code `FIELDS.MIC` definitions.

`?IOPF` I/O page fail which was not handled by the normal macro–code or TDBoot. The latest page–fail information is stored in MemA at `AM%PFN` (locations 500–517). The previous page fail information is stored in MemA at `AM%PFC` (locations 560–577). (The console terminal port will enter "Halt/Con" mode.)

`?XBRO` An operation was attempted that accessed the EBR or UBR, and the corresponding address was zero. (The console terminal port enters "Halt/Con" mode.)

`?INITERR` *nnn* An error was detected during initialization tests. The micro–code will attempt to continue, but proper operation is doubtful. The number printed, *nnn*, is a bit mask indicating the failed tests. The following values indicate particular failures:

1	The data did not compare during the FIFO test.
2	When the FIFO test ended the FIFO was not empty.
4	The FIFO became empty before the test was complete.
10	The XIlinx test failed.

`?PF` *jjj kkk lll* A page fail occurred as a result of a micro–console command. (The console terminal port will enter "Halt/Con" mode.) The values typed are suggestive of the faulting address and the nature of the fault.

jjj	The value of `AM%PFB` (EPT+500)
kkk	The value of `AM%PF0` (EPT+502)
lll	The value of `AM%PF1` (EPT+503)

See "Hard Page Failure" Section 3.7.1.8, page 257 for a description of these data words.

3.2.3 Console Terminal Programming

When the console port is in "Program" state, the console terminal is at the disposal of the running program. In Program state, each character that arrives at the console port is available by using the RDCTY instruction. If the program has enabled the console port to interrupt, the arrival of a character is signalled with an interrupt. Similarly, when the console port is ready, a character can be sent by the WRCTY instruction; if the program has enabled the console to interrupt, the Output Ready condition is signalled by an interrupt.

While running PDP–10 instructions, the processor microcode examines the condition of the console port (Input Ready, Output Ready) at every point where the XKL-1 processor can accept an interrupt. If the console requires service, it is served (at the microcode level) without regard for the status of Priority Interrupt system; of course, further processing of input or output characters depends on the responsiveness of the running program. When the console is in Program state, the UART Input Ready causes the microcode to read the character into its memory (where it is held for a RDCTY operation); set the Input Ready device status; and, if Console Input Enable is true, request an interrupt at the assigned level. UART Output Ready causes the microcode to set the Output Ready device status and, if Console Output Enable is true, request an interrupt at the assigned level.

The following instructions are available for the program to use in communicating with the console terminal:

RDCTY Read Console (APR3 1,)

703	1	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If a console port input character is available (Input Ready true), deposit it, right justified, in the location addressed by E; clear Input Ready (and clear the input interrupt request, if any). If no character is available, store zero in the location addressed by E.

WRCTY Write Console (APR3 3,)

703	3	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the console port output UART is available (Output Ready true), send the 8–bit character found right justified in the location addressed by E; clear Output Ready (and clear the output interrupt request, if any). If the console port output UART is busy (Output Ready false), this operation does nothing.

WRCTYS Write Console Status (APR3 4,)

703	4	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This is an immediate mode instruction. The status sent to the device in E consists of the priority interrupt level assignment in bits 33–35 and individual interrupt enables for output and input in bits 29 and 30, as shown below in the description of RDCTYS. When the priority level is non–zero, the console will generate an interrupt when Console Output Ready and Output Interrupt Enable are true or when Console Input Ready and Input Interrupt Enable are true.

RDCTYS Read Console Status (APR3 5,)

703	5	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the console status and store it in the location addressed by E. The console status consists of:

CON	*COI*	*CII*	*OIE*	*IIE*	*COR*	*CIR*	*Priority Level*
26	27	28	29	30	31	32	33 35

The bits and fields returned are

CON `CT%CON==:1B26` Console Mode. This flag, when set, indicates that the console port is communicating with the microcode console; i.e., "Console" state. When clear, the terminal is under control of the macro program: "Program" state.

COI	`CT%COI==:1B27` Console Output Interrupt. This flag is the logical AND of *OIE* (output interrupt enable) and *COR* (console output ready) and a non–zero priority level. This flag is read–only.
CII	`CT%CII==:1B28` Console Input Interrupt. This flag is the logical AND of *IIE* (input interrupt enable) and *CIR* (console input ready) and a non–zero priority level. This flag is read–only.
OIE	`CT%OIE==:1B29` Output Interrupt Enable. If set when the assigned priority level is not zero and when Console Output Ready is true, an interrupt will be requested. This flag is set by WRCTYS.
IIE	`CT%IIE==:1B30` Input Interrupt Enable. If set when the assigned priority level is not zero and when Console Input Ready is true, an interrupt will be requested. This flag is set by WRCTYS.
COR	`CT%COR==:1B31` Console Output Ready. When set, this flag indicates that the console port UART is free; another output character can be accepted. This flag requests an interrupt when set, if a non–zero priority level is assigned and *OIE* is set. This flag is read–only.
CIR	`CT%CIR==:1B32` Console Input Ready. When set, this flag indicates that the console port UART has a character available to be read. This flag requests an interrupt when set, if a non–zero priority level is assigned and *IIE* is set. This flag is read–only.
Priority Level	`CT%PIA==:7` Console Interrupt Assigned Priority Level. If zero, console activity will not interrupt the program. Otherwise, this is the level on which interrupts will be requested for input or output. This field is set by WRCTYS.

SZCTYS Skip if Zero, Console Status (APR3 6,)

703	6	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction tests bits 18–35 of the console status (as indicated by RDCTYS) against the immediate mask supplied by bits 18–35 of E. If all status bits selected by 1s in E are zero, the next instruction in sequence is skipped.

SNCTYS Skip if Non–zero, Console Status (APR3 7,)

703	7	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction tests bits 18–35 of the console status (as indicated by RDCTYS) against the immediate mask supplied by bits 18–35 of E. If not all status bits selected by 1s in E are zero, the next instruction in sequence is skipped.

3.2.4 Auxiliary Port

As mentioned above, the XKL-1 processor has an auxiliary console port to which a modem may be connected to allow a person at a remote location to watch and interact with the system as though using the local console terminal. The second port may be used to provide a facility for remote diagnosis.

The auxiliary port is equipped for RS232 modem control. When the port is disabled, the microcode will not answer incoming calls. When enabled, the microcode will assert DTR (Data Terminal Ready) on the port so that it can answer an incoming call. The microcode will "connect" the diagnostic terminal "in parallel" with the local console terminal port: all information sent to the console port will be copied to the diagnostic port; keystrokes from either port will be accepted as if they came from the local console port, and they will be echoed to both.

Both serial communication ports are responsive to `<CTRL/S>` for stopping type–out and to `<CTRL/Q>` for resuming it. If the line speeds of the two ports are different, throughput will be limited by the rate of the slower port.

3.2.5 Console Commands

Both the micro–console and macro–console commands are documented in Appendix E.

3.2.6 Console Communication Characteristics

Both console ports appear as Data Terminal Equipment (DTE) on standard, 25–pin D-series plugs. The main console port is wired for Signal Ground, Transmit Data, and Receive Data. The auxiliary port is wired for those three signal and for Data Terminal Ready, Ring Indicate, Request to Send, and Carrier Detect. The modem control signals are controlled and monitored by means of the WCTRLF and RCTRLF instructions, respectively.

Both ports operate at 9600 baud, with 8 data bits, 1 stop bit, and no parity.

3.3 Processor Initialization

This section is specific to the XKL-1 processor implementation. Future models may differ in detail.

The XKL-1 processor contains read–only memory from which it loads its operating microcode when power is applied.

Each XKL-1 in a system will perform the following initialization steps:

- Micro–code initialization:
  - Set the error light (red LED showing through the module cover panel) so that any failures during self–test will leave the light on.

  - Perform microcode level self–test. This tests the internal CPU datapaths, registers, and control logic. If any failures are detected during these tests, the microcode loops leaving the error light illuminated but without printing anything on the console. (The console is not used because it has not been tested yet.)
  - Perform secondary testing of remaining internal CPU hardware which is not vital to the micro–console operation.
  - Check NVRAM for power failure. Set bit `AP%NVB` in RDAPR if a power failure is detected.
  - When these tests are complete, extinguish the error light and the four green lights below it. Print one line identifying the processor type and microcode version. If any errors were encountered during secondary testing, the microcode will print the following message:

    `?INITERR` *nnn*

    The micro–code will attempt to continue, but proper operation of the system is doubtful. The number printed, *nnn*, is a bit mask indicating the failed tests. The following values indicate particular failures:

1	The data did not compare during the FIFO test.
2	When the FIFO test ended the FIFO was not empty.
4	The FIFO became empty before the test was complete.
10	The XIlinx test failed.

  - Test for the presence of the option jumper J2–0. (With the processor board held in its usual orientation, J2–0 is at the top position of the four jumpers.) This jumper appears as bit 0 in the APRID data at offset +1. If the jumper is present, do not run any macro code; just prompt for a command with the macro–console disabled (the micro–console `.M` command can be used to enable the macro–console manually). If the jumper is not present (this is the usual case), initialize the macro–console by starting it at offset +3, with the pager disabled. The macro–console fetches instructions from the CPU's Boot ROM.

- Macro–console initialization:
  - Perform some basic instruction tests to further verify proper operation of the hardware and microcode. If there are any failures during this process, the macro–console will halt and the microcode will print:

    ?MCON HALT at xxx

    The PC of the HALT instruction can be used to identify the problem with the XKL-1 processor's instruction set.
  - Print macro–console version, system ID, options, and microcode version. (See the `SHOW VERSION` command in Appendix E.)
  - Checksum the contents of the Boot ROM and report any error.
  - Validate the contents of non–volatile RAM (NVRAM). If the contents of NVRAM are not valid, then the entire contents are set to zero, except that all system start–up parameters are set to disabled. (See `DISABLE ALL` in Appendix E.)
  - Set the state of the auxiliary console from data saved in NVRAM. (The auxiliary console is disabled if the NVRAM was not valid in the previous step.)

- If `AP%NVB` is set in RDAPR data, issue a message warning of an NVRAM low battery condition
- If `AP%PWR` is set in RDAPR data, issue a mesage warning of a system power problem.
- Report any system start–up parameters that are disabled.
- If Cache–Test is enabled (see `ENABLE/DISABLE` in Appendix E), test the XKL-1 cache.
- If Pager–Test is enabled, test the XKL-1 pager (the translation buffer).
- If Bus–Poll is enabled, poll each slot (1–15) to determine which controllers are installed (including this CPU's slot number). If a previous configuration is stored in NVRAM, print any discrepancies between that and the current configuration.
- If there are multiple CPUs in the system, synchronize with each and choose a master processor. If this CPU is not the master, wait for instructions from the master CPU. If this CPU is the master, proceed.
- If Configure–Memory is enabled, configure all memory controllers. During configuration, test and/or clear according to the following settings:
  1. If Test–Memory is enabled, perform all memory tests, clear memory, and configure memory.
  2. If Test–Memory is disabled but Clear–Memory is enabled, clear and configure memory.
  3. If neither is enabled, configure memory.
- If Device–Configuration is enabled, poll for direct–access devices (disks) on each bus of each XRH-1 Mass–Storage Interface Processor and issue a start command if necessary.
- If Auto–Boot is enabled, attempt to boot the operating system using defaults (see `BOOT` command in Appendix E).
- If an Auto–Boot is successfully performed, the macro–console enters "Pgm" mode with the macro–code running. Otherwise, the macro–console enters "Con" mode with the macro–code halted.

3.3.1 Boot ROM

The Boot ROM consists of five 2M–bit EPROMs which are organized as one 256K–word memory (`BR%SIZ==:1000000`). The Boot ROM responds to addresses in Slot 0, section 10 (`BTSECT==:10`).

The Boot ROM contains the TDBoot program; a copy of Ddt;diagnostics for the cache, pager, and main memory; and a CPU verification test.

The Boot ROM is accessed when the pager is off by CPU references to addresses in section 10 (unpaged accesses imply slot 0). The Boot ROM may also be accessed while the pager is on by using an immediate page pointer[4] that specifies slot 0 and page numbers in the range 10000–10777; such references should be uncached. (References through the pager are uncached when there is no CST.)

The program that implements the macro–console is in the Boot ROM.

---

[4] See §3.7.1.

3.3.2 Initial Program Environment

For the macro–console commands that load programs (i.e., `BOOT` (`RUN`), `LOAD` (`GET`), and `MERGE`), the TDBoot program will create a virtual–memory environment for the program it loads, as described here. If the `.EXE` file specifies that anything is to be loaded into section zero, TDBoot creates a complete section zero in linear pages numbered 0 through 777. For programs that specify virtual addresses above section zero, TDBoot allocates memory starting at linear page number 1000 and consecutive linear pages as directed by the `.EXE` file being loaded. Pages explicitly described by the `.EXE` file as containing zero will be created with zeros in them. Pages outside of section zero that are not explicitly mentioned by the `.EXE` file are not created.

TDBoot creates an Executive Process Table, a User Process Table, and supersection tables and page tables as needed; these are allocated in memory starting at the highest linear page number and working downward. The page tables will use immediate, writable page pointers to describe where the loaded pages reside in memory. The page tables do not reference themselves; i.e., they are not part of the virtual–address space that they describe. TDBoot will not create a CST; therefore, no references are cacheable. No SPT is created. See also §3.7. TDBoot will map its ROM code into section 36. In the initial UPT and EPT, TDBoot will set up page trap and pushdown trap entries that point to its own trap handlers in section 36.

3.4 Priority Interrupt

The TOAD-1 System contains various processors that share the backplane bus. The various subsystems are subsidiary to the XKL-1 processor, but they maintain a degree of autonomy from it. Each subsystem processor generally operates from its own in–memory command queue and signals the completion of a task by moving an item from the command queue to the completed list. A subsystem processor interrupts the XKL-1 when errors occur or when it completes a task and causes the completed list to transition from empty to non–empty.

The priority–interrupt (PI) system allows the various subsystem processors to interrupt the XKL-1 at assigned levels of priority so that all can operate simultaneously. The hardware also allows conditions internal to the central processor, e.g., interval expiration, to request interrupts.

3.4.1 Sources of Interrupts

Any subsystem processor on the TOAD-1 System backplane bus may signal an interrupt, provided the program has enabled it to do so. Such interrupts are identified below by the physical slot number from which they originate.

Additionally, the XKL-1 processor has several appurtenances that may interrupt, when enabled by the program to do so; these are called “internal devices”. The internal devices include the console terminal port, the interval timer, the error logic, and the program request facility.

3.4.2 Priority Levels

Interrupts are handled on eight "levels" arranged in priority sequence, with level 0 being the highest priority and level 7 being the lowest.

Level 0 is totally unlike the other levels, in that the interruption of the normal flow of program execution is totally invisible at the level of the PDP–10 instruction set processor: activity on interrupt level 0 occurs between macro instructions. Level 0 is not under the control of the program: the WRPI instruction (§3.4.8) does not affect level 0. Level 0 is enabled whether the PI system is on or off.

Level 0 is used by the processor's operating microcode for the following purposes:

- Handling the internal devices.
- Responding to "Device Status" requests from other devices.
- Managing the pending–interrupts list: adding new interrupt requests, deleting requests that have been withdrawn, and determining whether to accept an interrupt request.

Note a semantic inconsistency: making an assignment of priority level 0 to an internal device or subsystem effectively disables that device from making interrupts.

Assignment of priority levels 1–7 to peripheral subsystems and internal devices is entirely at the discretion of the programmer. To direct a subsystem to use a particular level, the program sends the level to the subsystem in a device control message (or in a command list). Special instructions, unique to each internal device, direct an internal device to use a particular level. A subsystem or internal device that has been assigned a non–zero priority level is said to be "enabled" to interrupt at that level. Directing a subsystem or an internal device to use level zero tells that subsystem or device to avoid using interrupts altogether.

Any number of subsystems and devices may use the same priority level.

3.4.3 Interrupt Requests

When a peripheral subsystem that has a non–zero priority level requires the attention of the central processor, it sends an interrupt request message to the processor. In the message, the subsystem specifies the priority level that was assigned to it by the program. The bus interface portion of the processor receives the message (asynchronously relative to instruction execution) and places it in the hardware Request FIFO (first–in, first–out) queue, where it remains until the processor microcode reaches a point where it is receptive to interrupts. An interrupt is said to be "pending" at this point: the processor has the request, but it has not yet committed to "accept" the interrupt request by starting to process it.

On the backplane bus, the interrupt request is encoded as follows: Priority on bits 69–71 in binary format; bit 68 is Request/Withdraw interrupt (1 = Request).

At places where the operating microcode is receptive to interrupts, a non–empty Request FIFO queue or any of a variety of internal conditions will cause the microcode to depart from its usual path so that it might decide whether or not to accept an interrupt. Internal conditions are serviced.

For example, a newly available character on the console UART will be placed in microcode memory (MemA) and, if the console internal device is enabled to produce input interrupts, a console input interrupt request will be placed in the pending–interrupts list (a data structure which is maintained by the processor microcode). The contents of the Request FIFO are emptied into the pending–interrupts list.[5]

In the following circumstances, the processor will examine the pending–interrupts list to determine whether any pending request should be accepted:

- After the processor has made a change to the pending–interrupts list. The pending–interrupts list may change as the Request FIFO is emptied or when an instruction is executed that changes the interrupt status of an internal device.
- After the program has dismissed an interrupt.
- After a WRPI instruction that may have changed the status of the PI system.

3.4.4 Interrupt Acceptance

A pending request will be accepted as soon as the necessary conditions are satisfied: it must be at a priority level that is currently enabled and higher in priority than the level held by the processor; there must not be any request of higher priority corresponding to an enabled level; and, of all requests of equal priority, the accepted request is the oldest.[6]

The processor accepts a request by "holding[7]" the priority level of the request and performing the equivalent of an XPCW instruction directed at the subsystem's or device's "Interrupt Control Block". The Interrupt Control Block is four consecutive words in the Executive Process Table (EPT). The location of the Interrupt Control Block is selected according to which subsystem or internal device is the source of the interrupt.

For a device external to the processor, the EPT locations are determined by the physical slot number of the device. The device in slot number S uses four consecutive words in the EPT starting at location $100+4\times S$. Each of the processor internal devices is handled by the processor's operating microcode and has an assigned set of locations. The EPT locations of subsystem and internal device Interrupt Control Blocks are given in Table 3.1.

The processor accepts an interrupt by performing the equivalent of XPCW directed at the Interrupt Control Block corresponding to the interrupting subsystem or internal device. Thus, the flags and PC of the interrupted program are stored in the first pair of locations and new flags and PC are loaded from the second pair of locations. The new PC is usually the address of the Monitor service routine for the interrupting device; the new flags must clear User mode.

The most important point of which the programmer should be aware is that, even while User is set, the "equivalent of XPCW" executed to accept the interrupt is not part of the user program. It is executed in executive mode and is subject to executive mode restrictions only. The "equivalent of

---

[5] The Request FIFO is also used to hold "Device Status" and "Device Control" requests to which the processor will respond as it empties the FIFO. Further, the FIFO may contain requests to withdraw a previous interrupt request; to these the processor responds by removing the previous request from the pending–interrupts list.

[6] Although the XKL-1 processor will accept requests at a given priority level in the order that they are presented, no program should depend on this ordering.

[7] The word "hold" is used here in the sense of "possess", not "delay".

Table 3.1: EPT Locations for Interrupt Control Blocks

Device	EPT Location	Symbol
APR Error Conditions (§3.9.1)	000–003	`EP.APR`
Program Request (§3.4.8) on Level L, where $1 \leq L \leq 7$	$000 + 4 \times L$ $–003 + 4 \times L$	`EP.LV1` –`EP.LV7`
Console Input (§3.2.3)	040–043	`EP.CTI`
Console Output (§3.2.3)	044–047	`EP.CTO`
Keep–Alive Interrupt (§3.8.3)	050–053	`EP.KPA`
Interval Timer (§3.8.1)	100–103	`EP.ITM`
Subsystem in Slot S, where $1 \leq S \leq 15$	$100 + 4 \times S$ $–103 + 4 \times S$	`EP.D01` –`EP.D15`

XPCW" is implemented by processor microcode performing the same sequence of micro instructions as it does for an actual XPCW instruction, even though no actual XPCW instruction is fetched from memory.

3.4.5 Interrupt Processing

Upon accepting an interrupt request, the processor holds the priority level corresponding to the request. While holding that level, the processor will accept no request of equal or lower priority. The processor holds an interrupt level until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus, the processor can hold a number of different levels simultaneously. The condition of holding a priority level is sometimes called "interrupt in progress" on the level held.

In accepting an interrupt request, the processor executes the equivalent of an XPCW instruction. The program run subsequent to the XPCW is called an "interrupt service routine." This program is expected to attend to whatever the interrupting subsystem or device requires, thus satisfying the request. The requesting subsystem (or device) is expected to send a message (or change its status) withdrawing its interrupt request.

3.4.6 Interrupt Dismissal

Upon completion of the interrupt service routine, the program dismisses the interrupt by using the XJEN (JRST 7,) instruction. This instruction directs the processor to restore the priority level currently held to a state in which it—and lower priority levels—is receptive to further interrupts. XJEN also restores the PC of the process that was interrupted.

The XJEN should restore the interrupted program's flags and PC from the double word stored at the subsystem's or device's interrupt location in the Executive Process Table.[8]

---

[8] Note the effective–address of the XJEN is normally the virtual address of the EPT in executive space, plus the offset corresponding to Interrupt Control Block of the source of the interrupt.

Caution

An interrupt routine must dismiss the interrupt when it returns to the interrupted program. Otherwise, its level and all levels of lower priority will be disabled and the processor will treat the interrupted program as a continuation of the interrupt service routine.

A single interrupt level will shut out all levels of lower priority if, every time its service routine dismisses the interrupt, a subsystem or device at this priority level is already waiting with another interrupt request. In particular, if a subsystem or device fails to withdraw its interrupt request, the unwithdrawn request will shut out all levels of lower priority and all other requests at this level.

3.4.7 Interrupt Register

Interrupts and other unusual conditions are gathered in the interrupt register. The logical OR of the various conditions reported in the interrupt register is visible to the microcode, which can dispatch to an alternate address when the OR of these conditions is true. Reading the interrupt register resets the register, so the microprogram must store the result for perusal. The action of reading the register also advances the FIFO to the next data item, if any.

The microcode stores the latest data read from the interrupt register in MemA; it can be examined by using the console command *.EI*.

The format of the interrupt register is depicted below.

Interrupt Register

FNE	*FFul*	*BPE0*	*CLS*	*BPE1*	*CDP*	*CTP*	*PDP*	*PTP*	*BBC*	*TBO*	*ITI*	*UI*	*UIP*	*PFI*	*PFP*		*FIFO Data*		
																	FC	*Software and PIA*	*Slot*
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		18 19	20 31	32 35

The contents of the interrupt register are interpreted as shown below. The items marked with "*" are included in the logical OR of terms that cause the alternate microcode dispatch.

FNE	Interrupt FIFO is Not Empty. When this bit is set, the contents of bits 18–35 are valid.
FFul	FIFO Full. The interrupt FIFO is full: all 64 locations of the FIFO have been filled, and an interrupt request has been discarded.
*BPE0**	Bus Parity Error 0. A word read from memory has bad parity. The bad data has been provided to the CPU. (This condition gives rise to a Memory Parity Error 0 page failure.)
*CLS**	Cache Line Order Scrambled. During a "Line Read" operation, the memory presented data in an order other than what the CPU requested. The CPU has been given the wrong data; incorrect data is also present in the cache.

*BPE1**	Bus Parity Error 1. In a "Line Read" operation, the memory has provided a data word that has bad parity. The erroneous data has not yet been seen by the CPU; however, the bad data has been written, with good parity, in the cache. For further diagnosis, the CPU should flush the cache. (This condition gives rise to a Memory Parity Error 1 page failure.)
*CDP**	Cache Data Parity Error.
*CTP**	Cache Tag Parity Error.
*PDP**	Pager Data Parity Error.
*PTP**	Pager Tag Parity Error.
*BBC**	Bad Bus Cycle. An unexpected response has been received from some device on the backplane. For example, a cycle that timed out has completed, or a "Line Read" request was met by a "Word Read Return". Due to scarcity of time and data path in the processor, no further information is available.
*TBO**	Time-base Overflow. A carry out of the time–base counter has occurred. The microcode should increment the in–memory copy of the time–base accordingly.
*ITI**	Interval Timer Interrupt. The interrupt counter has reached its assigned value.
*UI**	UART Interrupt. A rising edge of the "UART Interrupt Pending" lead (see below) has been detected.
UIP	UART Interrupt Pending. This bit monitors the condition of the UART status output. The UART status output lead is programmable (by the microcode) on which conditions to report as interrupts.
*PFI**	Power Fail Interrupt. The PFAIL– backplane signal is being asserted while the interrupt is enabled. The PFAIL– signal denotes either the failure of AC power or the presence of the thermal warning condition. (The data returned by `RCTRLF` allows the program to determine which environmental conditions prevail.)
PFP	Power Fail Pending. This flag reflects the state of the PFAIL– backplane signal.
FIFO Data	Data is valid if bit 0 contains one. Bits 18–19 encode *FC*—function type, which is one of "Control Write", "Status Read", or "Interrupt Request". The interpretation of the other FIFO bits depends on the function selected by bits 18–19, as follows:

FIFO Bit	Bus Bit	Control Write	Status Read	Interrupt Request
18	Type	1	0	0
19	Type	0	1	0
20–28	A60–A68	Reserved	Reserved	Reserved
29	A69	Reserved	Reg<0>	PI<0>
30	A70	Reserved	Reg<1>	PI<1>
31	A71	Reserved	Reg<2>	PI<2>
32–35	SRC0–SRC3	Source Slot Number of Request		

3.4.8 Program Control of the Priority Interrupt System

The program can control the PI system by means of the following instructions:

WRPI **Write Priority Interrupt (APR0 14,)**

700	14	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the function(s) specified by the effective–conditions *E* (an immediate quantity) as shown; a 1 in a bit produces the indicated function, a 0 has no effect.

	Drop Prgm Req On Lvls	Clear PI Sys–tem	Make Prgm Req On	Turn On	Turn Off	Turn Off	Turn On	*Select Levels for Bits 22,24,25,26*						
			Selected Levels			PI System		*1*	*2*	*3*	*4*	*5*	*6*	*7*
18	22	23	24	25	26	27	28	29	30	31	32	33	34	35

22 `PICPIR==:1B22` On levels selected by 1s in bits 29–35, turn off any "program requests" (see bit 24, below) made previously. This is the proper way to clear program requests. If this bit is set when bit 24 is also set, the effect of the instruction is not defined.

23 `PICLPI==:1B23` Clear the PI system: turn off the PI system, turn off all levels, drop all program requests, restore all levels that are currently being held, and clear the processor's pending–interrupts list. If this bit is set with any other bit, the effect of this instruction is not defined.

24 `PISPIR==:1B24` Make a "program request" for interrupts on levels selected by 1s in bits 29–35. If this bit is set when bit 22 is also set, the effect of this instruction is undefined.

A program request persists indefinitely. Therefore, as soon as an interrupt is completed on a given level, another is started until the request is turned off by a WRPI that selects the same level and has a 1 in bit 22.

The processor may allow the program to continue while it decides whether to accept a request. Thus, when this bit creates a pending request, some additional program instructions may be performed before the interrupt. If the program forces an interrupt on the lowest–priority level when all levels are active, there can be a very long time interval between the WRPI and its interrupt.

25 `PICHON==:1B25` Turn on (enable) the levels selected by 1s in bits 29–35 so that interrupt requests can be accepted on them. If this bit is set when bit 26 is also set, the effect of this instruction is not defined.

26 `PICHOF==:1B26` Turn off (disable) the levels selected by 1s in bits 29–35 so that interrupt requests cannot be accepted on them. Pending requests for this level accumulate while the level is disabled. If this bit is set when bit 25 is also set, the effect of this instruction is not defined.

27 `PIPIOF==:1B27` Turn off the interrupt system so that no requests can be accepted. Pending requests accumulate while the interrupt system is off. If this bit is set when bit 28 is also set, the effect of this instruction is not defined.

28 `PIPION==:1B28` Turn on (enable) the interrupt system so that the hardware can accept requests. Pending requests (and any subsequent new requests) are accepted in order of their priority. If this bit is set when bit 27 is also set, the effect of this instruction is not defined.

29–35 `PICHNM==:177B35` Individual bits to select levels for bits 22, 23, 24, and 25.

RDPI Read Priority Interrupt (APR0 15,)

700	15	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the status of the PI system into location E as shown:

	Program Requests on Levels								*Interrupt Holding on Levels*							*PI On*	*Levels On*						
	1	2	3	4	5	6	7		1	2	3	4	5	6	7		1	2	3	4	5	6	7
0	11	12	13	14	15	16	17		21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Levels that are on (enabled) are indicated by 1s in bits 29–35; 1s in bits 21–27 (`PIPIIP==:177B27`) indicate levels on which interrupts are currently held (i.e., levels on which interrupts have been accepted and not yet dismissed); and 1s in bits 11–17 (`PIPIRM==:177B17`) indicate levels on which program–initiated requests have been made by using WRPI with a 1 in bit 24. A 1 in bit 28 indicates that the PI system is turned on.

SZPI Skip if Zero, Priority Interrupt (APR0 16,)

700	16	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction tests bits 18–35 of the PI system status (as indicated by RDPI) against the immediate mask supplied by bits 18–35 of E. If all status bits selected by 1s in E are zero, the next instruction in sequence is skipped.

SNPI Skip if Non–zero, Priority Interrupt (APR0 17,)

700	17	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction tests bits 18–35 of the PI system status (as indicated by RDPI) against the immediate mask supplied by bits 18–35 of E. If not all status bits selected by 1s in E are zero, the next instruction in sequence is skipped.

SIMIRD Simulate Interrupt Register Data (APR0 12,)

700	12	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction reads the word at location E and uses the data found there as though it had come from the interrupt register and FIFO.

Software uses this instruction to force the processor hardware to simulate the effect of certain types of interrupts and bus cycles. For example, to drop an interrupt from a particular slot, the program may issue this instruction with the data portion set to “FIFO Not Empty”, FIFO Function Code set to 00, and bits 29–31 set to the priority level and 32–35 set to the slot number.

3.4.9 Special Considerations

Generally, interrupts are processed between instructions; in such cases PC points to the next instruction to execute. After taking care of the interrupt, the processor simply returns to the interrupted process by restoring the PC, flags, and PI system status by using an XJEN instruction. Some instructions have been designed to be interruptible at particular points within their execution; for these, the instruction (i.e., the microcode) has been arranged to save the state of its partial computation so that the instruction can be resumed if interrupted: the PC will point to the instruction that was interrupted. Either:

- the interrupt was between the first and second parts of a two–part instruction, where First Part Done being set prevents the processor from repeating any unwanted operations in the first part or
- the interrupt occurred at some point in a multipart instruction where the microcode rigged the various pointers and other quantities so the processor actually restarts the instruction at the point where it stopped, rather than at the beginning.

Note that, in multipart instructions such as BLT, the byte manipulation instructions, or the string instructions, the very mechanism that facilitates the resumption of an interrupted instruction results in special properties of which the programmer must be aware.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then, when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that, if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT; the programmer cannot have the BLT load AC, except by the final transfer; moreover, AC will not be the same after the instruction as it was before.

An interrupt can also start in the second effective–address calculation in a two–part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a flag word; if it is restored by the interrupt routine when the interrupt is dismissed, this flag prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same byte pointer, because it now points to the next byte. Giving ILDB or

IDPB would skip a byte, and if the routine restored the flag, the interrupted ILDB or IDPB would process the same byte as did the routine.[9]

3.4.10 Programming Suggestions

The Monitor handles all interrupts for user programs. Even if the User In–Out flag is set, a user generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program PI routines, there are several rules to remember:

- To prevent a device from hanging up a level, the programmer must be aware of—and satisfy—whatever requirements the device has for dropping its request.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. Computations and any other time–consuming activities that could be performed elsewhere should be excluded from the interrupt service routine.
- Never turn off the interrupt system in a routine unless it is absolutely necessary to do so, and then always turn it on again as soon as possible. If one or more priority levels can be turned off instead, that is preferable to turning off the whole PI system.
- If the interrupt service routine uses a UUO, it must first save the contents of the locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type (§2.16).
- The routine must dismiss the interrupt with an XJEN when returning to the interrupted program. Flags and UUO locations must be restored.

3.5 Cache Operations

For the user, the cache is transparent: any program simply gets information from memory and stores information in memory. However, the use of a cache as part of the memory subsystem reduces program time, because the cache is faster than the storage modules. The cache also reduces the program's need to access storage, making a larger fraction of the storage bandwidth available to other parts of the system. As explained in §1.1.2, transfers between cache and storage are in eight–word lines; storage references are to eight locations at a time.

The cache contains representations of a selection of such memory lines. One may view the cache as 131,072 registers organized in lines of eight, which temporarily substitute for the most frequently referenced storage locations. The cache serves this function not only for the program but for all applicable microcode references, including those for handling interrupts, traps, page refills, and other automatic operations.

The cache is organized as 8,192 pairs of eight–word lines. Address bits 33–35 select a word within a line. Address bits 20–32 select a pair of lines. A cache directory specifies the backplane bus address[10]

---

[9] Generally speaking, it would be better to use different byte pointers at different priority levels.

[10] The 4–bit slot number and bits 7–19 of the in–module address; for cache references to memory, the device bit, D, is implicitly zero.

associated with each of the lines in the pair, the "valid" and "modified" flags for each of the lines, and one bit to indicate which one of the pair of lines was least recently used.

Initially, the cache contains nothing: both valid bits in every pair of lines are clear. The way the hardware handles the cache depends on whether the processor's initial reference to a location in a line is a read or a write.

When the processor's first reference to a line is to read the contents of one of its locations, memory control retrieves from storage the entire eight–word line that contains the referenced location. The single word requested is supplied to the processor (which had been waiting) and all eight words are loaded into the cache line, which is marked as valid and unmodified; i.e., representing the true contents of memory. The actual backplane bus transactions that read a line are arranged so that the requested word in the line is returned in the first bus cycle of the memory's response. Subsequent references, read or write, to the same line are made to the cache, not to storage. A write reference to any word in the line changes the contents of the cache, not storage, and causes the entire line to be marked as "modified". A line that is valid and modified represents what the contents of storage should be; a modified line eventually must be written back to storage.

When the processor's first reference to a line is for writing, the memory control accepts the new contents of the word and holds it while the processor continues. Memory control will cause the entire eight–word line to be read from storage and loaded into the cache. The newly written data item is placed in the cache, obliterating the information read from the corresponding storage location, and the cache line is marked as "valid" and "modified".

As the program executes, the cache fills with valid data. Eventually, a reference is made to a location not already in the cache and for which both lines in the appropriate set are valid. To make room in the cache for this latest reference, the least recently used line of the selected set is removed. If the line being removed is both valid and modified, the line will be written to memory before the line is reassigned for the latest reference. If the line was not both valid and modified, it is reassigned immediately. In either case, the memory control and cache then proceed with the processor's read or write reference, as described in the two paragraphs above.

A memory reference is cacheable (i.e., subject to being put in the cache and/or found in the cache) only when the following are true: the memory reference occurs while the pager is on, the reference uses the pager, the CST Base Register is non–zero, and the CST entry for the page on which the reference occurs permits caching; see also §3.7.

3.5.1 Cache Programming

The cache is interposed between storage and the XKL-1 processor, not between storage and the peripheral subsystems. Therefore, any operation of the peripheral subsystems to write storage may render the cache's representation of storage incorrect. For example, in an input operation, data from a peripheral device is written in memory; this action changes memory without automatically marking as invalid the cache's representation of what is in that memory. Therefore, in the absence of measures taken to prevent this problem, the cache would supply the old, incorrect data to the processor. Similarly, a peripheral's attempt to read storage in an output operation will obtain stale (incorrect) data when the cache contains valid and modified data.

The Monitor is responsible for managing the relationship between storage modules, peripheral devices, and the cache. To prevent problems of cache/storage inconsistency, the XKL-1 provides operations that the program can perform on the cache. Any one of these operations may be applied to all entries in the cache or to all entries on a single page. These operations are

- Invalidate cache. To invalidate a cache line is simply to clear its valid and modified flags so it no longer represents anything.
- Validate storage (writeback). To validate storage is to write to memory a cached line whose modified flag is set and then clear the modified flag; validation of an unmodified cache line is a no–op.
- Validate storage and invalidate the cache (unload). To unload a cache line is to validate storage, if needed, and then to clear the valid flag.

Following power turn–on, the cache directory contains indeterminate data. The cache is properly initialized by invalidating all entries.[11]

Consider the situation in which a program is finished with the data in a particular (modified) page which is now to be swapped via a peripheral subsystem with new data to be brought into the same physical page for later use. Any cached data for this page must be unloaded into storage: first, so that storage will contain validated data prior to outputting that data to disk; second, so that the newly input data from disk will be read from storage rather than from the cache. In contrast, consider a page of data that has just been loaded into memory by LINK. To create an executable file from this memory image, the memory pages must be copied to disk; however, the data remains in memory, ready for execution. Any cached data for the page must be validated in storage prior to the output operation; however, so that the program can execute from the cache, there is compelling reason to leave any valid data for the page in the cache.

3.5.2 Cache Sweeping Instructions

There are six instructions to perform the three sweep operations (i.e., invalidate cache, validate storage, and unload cache), either for one physical page or for the entire cache. When sweeping for one page, only the 64 sets of lines that may contain information for the selected page are examined. The validate–all and unload–all operations require an examination of all 16,384 lines; however, in the XKL-1, the invalidate–all operation is very fast.

---

[11] The cache contents will have indeterminate parity following power turn–on, but this should not create any problem in the normal use of the cache. Invalidating all entries ensures that the cache directory has good parity.

Caution

> When interrupted, a sweep–all instruction stores its intermediate state in locations of MemA specific to the particular form of sweep–all that is in progress. If a sweep–all instruction is interrupted and the interrupt process attempts another sweep–all instruction of the same type (i.e., SWPUA is interrupted and another SWPUA is started), the state of the interrupted instruction will be lost. The instruction in the interrupt routine will perform the entire indicated operation (unless it is interrupted); when the interrupted program resumes, the interrupted sweep–all instruction will terminate immediately.
>
> The same caution applies to processes that run in response to page traps, because a sweep instruction could be interrupted by a Cache Tag Parity Error or a Cache Data Parity Error page trap. Recovery from a Cache Data Parity Error or a Cache Tag Parity Error should avoid the use of the sweep instructions and, instead, pinpoint the error via the cache diagnostic instructions.

SWPIA Sweep Cache, Invalidate All (APR1 11,)

701	11	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Clear the valid bit from all cache lines. This operation is appropriate after power–up.

The cautionary message that precedes this instruction description should generally be heeded. In fact, however, the actual implementation of SWPIA in the XKL-1 does not require storage of intermediate–state information and it cannot generate page traps.

SWPIO Sweep Cache, Invalidate One Page (APR1 15,)

701	15	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Clear the valid bit from the lines holding information for one page. Location E contains a bus address word (§3.1.4) that specifies the page being swept. Bits 27–35 of the bus address word are ignored. Bit 0 of the bus address word must be zero.

This instruction might be given for a page after an operation that inputs data to it and before that data is examined. Equivalently, it may be given after a page is discarded; e.g., when a read–only page is swapped out.

SWPVA Sweep Cache, Validate All (APR1 12,)

701	12	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

All 16,384 cache lines are examined. Copy to storage any line in which the "modified" bit is set and clear the "modified" bit.

The caution on page 229 applies to SWPVA.

SWPVO Sweep Cache, Validate One Page (APR1 16,)

701	16	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Copy to storage any modified lines belonging to the specified page. Location E contains a bus address word (§3.1.4) that specifies the page to be swept. Clear the "modified" bit for any line copied to storage. Bits 27–35 of the bus address word are ignored; bit 0 must be zero. This instruction is appropriate when a page must be written to a peripheral and retained in memory.

SWPUA Sweep Cache, Unload All (APR1 13,)

701	13	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

All 16,384 cache lines are examined. Copy to storage any line in which the "modified" bit is set. Clear the "modified" and "valid" bits in every line.

This instruction is appropriate before turning off the cache.[12] It may be appropriate in situations where a large–scale output operation is contemplated.

The caution on page 229 applies to SWPUA.

SWPUO Sweep Cache, Unload One Page (APR1 17,)

701	17	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Copy to storage any modified lines belonging to the specified page; invalidate all lines belonging to the specified page. Location E contains a bus address word (§3.1.4) that specifies the page being swept. Bits 27–35 of the bus address word are ignored; bit 0 must be zero.

This instruction is appropriate when a page is to be written to a peripheral and then discarded.

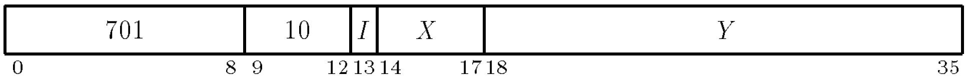
3.5.3 Cache Diagnostic Instructions

There are two instructions by which the program can verify the cache data and tag memories. These instructions are not considered part of the normal operating repertoire; they may, however, be part of system initialization, diagnosis, or error recovery.

---

[12] The cache cannot be turned off directly. The CST must be changed to mark pages as uncacheable; then the cache can be unloaded.

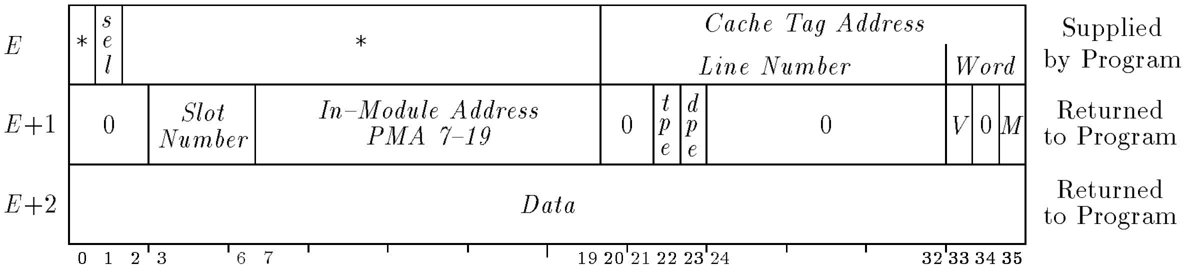
DRDCSH Diagnostic Read Cache Tag and Data (APR1 10,)



Read selected data and directory information from the cache. The effective–address E points to a block of three consecutive words. The word at E contains fields that select which cache word to read. Data is returned in E+1 and E+2, as shown below.

If a Cache Tag Parity Error or a Cache Data Parity Error is detected by this instruction, the error condition is reported in the returned data; these errors do not cause page traps. (Of course, a page trap can occur if the address specified by E is not in memory, etc.)

Cache Tag and Data Tripleword — DRDCSH



The fields have meanings as follows:

*	Fields marked with an asterisk are ignored by hardware.
sel	`CH%SEL==:1B1` The select bit: a zero selects the left–hand line of a pair of cache lines and one selects the right–hand line.
Cache Tag Address	`CH%ADR==:177777` Bits 20–32 select the line number (`CH%LIN==:177770`) and bits 33–35 select the word within the line (`CH%WRD==:7`).
Slot Number	`CH%SLT==:17B6` The 4–bit physical slot number of the memory module being represented by this cache line.
In–Module Address	`CH%IMA==:17777B19` This field contains bits 7–19 of the in–module address of the memory line represented by this cache line. (Bits 20–32 of the in–module address of the memory line are given by the cache line number.)
tpe	`CH%TPE==:1B22` This bit is set to denote a Cache Tag Parity Error in one or both sets for the indicated cache line.
dpe	`CH%DPE==:1B23` This bit is set to denote a Cache Data Parity Error in the specific word.
V	`CH%VLD==:1B33` This is the Valid bit. If set, the other information that is returned in this instruction is a representation of the contents of memory. If V is zero, the data returned by this instruction is not meaningful (except,

perhaps, for diagnostic purposes). The select bit and bits 20–32 of the *Cache Tag Address* (the cache line number) uniquely identify one line in the cache. Each line has just one Valid bit, one Modified bit, one Slot Number field, and one In–Module Address field; these bits and fields serve all eight permutations of bits 33–35 of the *Cache Tag Address* (the word number within the cache line).

M	`CH%MOD==:1B35` This is the Modified bit. If set, some word on the selected line (not necessarily the word addressed by bits 33–35 of the *Cache Tag Address*) differs from memory. To validate memory, this line must be rewritten to memory.
Data	This is the in–cache data corresponding to the memory location addressed by the *Slot Number* and a 29–bit in–module address composed of bits 7–19 of the *In–Module Address* and bits 20–35 of the *Cache Tag Address*. If the modified bit is set, this word may differ from actual memory contents; if the modified bit is zero, this word should be identical to actual memory contents.

DWRCSH Diagnostic Write Cache Tag and Data (APR1 14,)

701	14	*I*	*X*	*Y*
0–8	9–12	13	14–17	18–35

The address of a three-word block of memory is specified by E. The cache address into which to write is specified in the word addressed by E. The data to write into the cache is given in E+1 and E+2, as shown below.

This instruction can be used to correct cache tag parity and/or cache data parity.

Cache Tag and Data Tripleword — DWRCSH

<table>
<tr><td><i>E</i></td><td>*</td><td><i>sel</i></td><td colspan="3">*</td><td colspan="4"><i>Cache Line</i></td><td colspan="3"><i>Word Num</i></td><td>Supplied by Program</td></tr>
<tr><td><i>E</i>+1</td><td>*</td><td colspan="2">0</td><td><i>Slot Number</i></td><td><i>In–Module Address PMA 7–19</i></td><td>*</td><td><i>wbt</i></td><td><i>wbd</i></td><td>*</td><td><i>V</i></td><td>*</td><td><i>M</i></td><td>Supplied by Program</td></tr>
<tr><td><i>E</i>+2</td><td colspan="12"><i>Data</i></td><td>Supplied by Program</td></tr>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3–6</td><td>7–19</td><td>20–21</td><td>22</td><td>23</td><td>24–32</td><td>33</td><td>34</td><td>35</td><td></td></tr>
</table>

The fields are as described in DRDCSH, with the following additional notes:

wbt	`CH%WBT==:1B22`) Write Bad Parity Tag: set this bit to make this operation write bad parity with the cache tag. A subsequent read of this tag is expected to report the bad parity, which normally would cause a page failure. (However, DRDCSH would report the parity error without causing a page failure.)
wbd	(`CH%WBD==:1B23`) Write Bad Parity Data: set this bit to make this operation write bad parity with the data.

V	`CH%VLD` The value supplied by the program will be written into the cache tag selected by the *sel* bit and the cache line number.

If the cache is to be kept consistent with memory (a requirement of operating systems programming, but not necessarily a requirement of a diagnostic routine), this instruction must be used with particular care.

3.5.4 Cache Management

Management of the cache is relatively straightforward. The Monitor must simply be sure always to update storage pages before an output operation and to invalidate the cache representation of pages after an input operation so that processor references to the new data will go to storage.

The same procedures are used for a multiprocessor system, but here a problem arises when different processors are allowed to reference the same page at the same time, if either processor is allowed to modify the page. With read–only pages, the cache copies in both processors will be remain valid. However, if a processor modifies a shared page, the other processor cannot expect to get up–to–date data from its cache. To handle this situation, the pager includes mechanisms for bypassing the cache. A cache bit, associated with each individual memory page (see §3.7), is used by the paging hardware to determine whether or not cache use is allowed for a page.

To the extent that input–output operations are specified via in–storage command lists and completed lists, these lists appropriately belong in pages that are uncached.

To cause a page to be cacheable, the Monitor must set the CST cacheable bit for the page to indicate that it is cacheable, and the Monitor should issue a `CLRPT` to cause a pager refill for the affected page, so the pager can “see” the new status of the page.

To cause a page to be uncached, the Monitor must clear the CST cacheable bit for the page, issue a `CLRPT` to prevent any new entries from coming into the cache, and sweep the cache to unload the affected page. To cause all pages to be uncached, the Monitor must validate the pages containing the EPT and UPT; disable caching of the EPT, UPT, and SPT; disable caching throughout the system (perhaps by setting the CST to zero); and unload the entire cache.

3.6 XKL-1 Processor Internal Memory

The XKL-1 processor board includes two kinds of internal memory, called MemA and NVRAM.

3.6.1 MemA

The XKL-1 processor includes an on–board internal memory called MemA. This memory stores 36–bit data words and has 8,192 locations (`AM.CAP==:20000`). MemA is used for a variety of internal purposes. The first 128 locations are the eight AC blocks.

3.6.1.1 Operations on MemA

The following operations are supported.

AMOVE Move from MemA

714	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of MemA location specified by E and store the data in AC. The in–section value of E must be in the range 0–17777 (octal).

AMOVEM Move to MemA

715	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Store the contents of AC in the MemA location specified by E. The in–section value of E must be in the range 0–17777 (octal).

3.6.1.2 MemA Specific Locations

Notice

Specific locations in MemA identified in this manual are **not** part of the architectural specification of the TOAD-1 System. They are mentioned here for the convenience of the authors of the processor microcode, TDBoot, and diagnostics. They are subject to change.

The present allocation of MemA locations is described in Appendix F.1.

3.6.2 Non–Volatile RAM

The XKL-1 processor includes an internal non–volatile memory, called NVRAM. The NVRAM is organized as an 8K×8–bit memory (`NV%SIZ==:20000`). NVRAM holds processor configuration parameters that are needed by the system before it accesses the disks.

3.6.2.1 Operations on NVRAM

The NVRAM is accessed by the following instructions:

NMOVE Move from NVRAM

706	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of NVRAM location specified by E and store the data in AC bits 28–35; clear AC bits 0–27. The in–section value of E must be in the range 0–17777 (octal)

NMOVEM Move to NVRAM

707	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Store the contents of AC bits 28–35 in the NVRAM location specified by E. The in–section value of E must be in the range 0–17777 (octal). The contents of AC are not affected.

The NVRAM is a low–power RAM that contains a lithium battery to maintain the memory contents while external power is off. When the battery discharges (the estimated life is 10 years) the NVRAM must be replaced. Each time that external power is applied to the NVRAM (after having been absent), if the battery is low, the second access to the device will fail. Therefore, in its initialization sequence, the processor microcode reads NVRAM location 17777 (the first access); complements the data; writes it back (the second access); and then reads the data again. If the second read (the third access) produces the same value as the first, the processor knows that the NVRAM battery is low: it signals that fact via the `AP%NVB` flag in RDAPR. If the data changes, the battery is working properly and the processor restores the original data to location 17777. When the battery charge is low, the NVRAM contents may be unreliable.

3.6.2.2 NVRAM Specific Locations

> **Notice**
>
> Specific locations in NVRAM identified in this manual are **not** part of the architectural specification of the TOAD-1 System. They are mentioned here for the convenience of the authors of the processor microcode, TDBoot, and diagnostics. They are subject to change.

The allocation of NVRAM locations is described in Appendix F.2.

3.7 Paging and Memory Management

General information about machine modes and paging procedures is given in §1.4. This section treats in detail the structure of the process tables and certain hardware procedures—paging, page refills, and page failures—a knowledge of which is necessary for an understanding of operating system programming.

The mapping described here is the XKL-1's extension of what was informally called "TOPS–20 paging" on the KL10 processor. "TOPS–20 paging", which is now, formally, called "Extended KL Paging", was originally developed to support the TOPS–20 operating system. However, with the advent of extended addressing, DEC's other monitor, TOPS–10, was converted to use the newly renamed "Extended KL Paging".

Although the remainder of this section describes the hardware support for the TOPS–20 operating system, the reader should keep in mind that the TOPS–10 operating system makes use of a subset of the same facilities.[13]

At the physical level, the TOAD-1 System deals in bus addresses. As described in §3.1.4, a bus address has three components: a 4–bit physical slot number, which selects a particular memory or peripheral subsystem; a 29–bit in–module address, which specifies a particular location within the selected module; and one bit, called D, that specifies the bus cycle type (either memory or control/status) to be used in dealing with the device addressed by the slot number. Because D, the bus address, and the in–module address, together, are wider than the 30–bit addresses produced by the effective–address calculation, all program references to memory are regarded as being references to virtual addresses; i.e., all addresses are translated by the pager.[14] Special instructions (§3.1.4), intended for input–output operations and for diagnostics, bypass the pager by providing bus addresses directly.

In executive mode, the program considers all of its dealings with memory to be in its virtual–address space; instructions reference executive virtual space except for the special instructions that specifically call for bus–space references. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Bus–space references are made by the pager microcode to carry out the mapping procedure; also, microcode references to peripheral subsystems are performed in bus–space.

Note

> Paging operations are inextricably intertwined with the activities of the TOPS–20 operating system. The reader must become familiar with both in order to understand either fully.

3.7.1 Paging

All of memory is divided into pages of 512 words each. Architecturally, the TOAD-1 System backplane bus specifies addresses by a four–bit physical slot number and a 29–bit in–module address; slot numbers 1–15 are legal, slot 0 does not exist. The pager translates virtual addresses (either user or executive) to bus addresses.

---

[13] XKL LLC has no plans to support TOPS–10 on the XKL-1.

[14] When TDBOOT is started, paging is turned off: virtual addresses are passed unchanged through the pager. These addresses invariably select slot 0, a non–existent slot, but the processor's on–board ROM will respond to these addresses.

The virtual–memory space of a program is 2,097,152 pages, expressed in 30–bit addresses. Of the address bits, the left twenty–one bits (6–26) are the extended page number. The virtual–address space is usually regarded as 4,096 sections, each of 512 pages. With this view, the extended page number has two parts: the left–most twelve bits (6–17) specify the section and the right–most nine (18–26) specify the page.[15]

By paging, the hardware maps each page of the virtual–address space into a part of the backplane bus–address space.[16] In this transformation, the right–most nine bits (27–35) of the virtual address are not altered; in other words, a given offset into a virtual page is the same offset into the corresponding bus–address page. The paging translation maps a 30–bit virtual address to the bus–address space by transforming the 21–bit extended page number to a four-bit bus–slot number and a 20–bit in–module page number. These, together with the nine unchanged in–page address bits, are the bus address. The procedure is carried out automatically by the pager, but the maps that supply the necessary substitutions are constructed by the executive program.

The pager makes use of the Executive Base Register (EBR), by which the executive identifies the bus–page address of the Executive Process Table (EPT). Among other things, the EPT contains, at locations 540–547 (`EP.SS0==:540`), the eight–entry executive supersection table. The EPT also contains information by which the Monitor handles "hard" page traps (§3.7.1.8).

The pager also contains the User Base Register (UBR), by which the executive identifies the bus–page address of the User Process Table (UPT). The UPT is the user–space analog of the EPT; it contains the eight–entry user supersection table (at `UP.SS0==:540`). The User Process Table also contains information by which the Monitor handles "soft" page traps (§3.7.1.8) and MUUOs (§2.16).

The translation of a 30–bit virtual address is performed in the following conceptual steps. To select one supersection, the supersection number, bits 6–8 of the virtual address, is added (right justified) to 540 plus the value found in the EBR (or UBR, if mapping a user address). The result is the bus address of the needed supersection pointer. The supersection pointer provides the bus address of a section table; the section table contains 512 section pointers. To select one section pointer, the section number, bits 9–17 of the virtual address, is added (right justified) to the bus address found in the supersection pointer. The result is the bus address of the needed section pointer. The section pointer provides the bus address of a page map; the page map contains 512 map pointers. To select a map pointer, the in–section page number, bits 18–26 of the virtual address, is added (right justified) to the address contained in the section pointer. The word thus selected is a map pointer that contains the bus address of the desired page.

Every pointer and mapping requires one word. Thus, a page map for one section requires one page; a page map for one supersection table requires one page. The figures on the following pages show the organization of the virtual–address spaces, the process tables, the supersection tables, and the section tables for both user and executive. Figure 3.3 gives the general layout of the process tables and shows the relation between virtual–address spaces and the supersection and section tables. Figure 3.4 shows the process table configuration used in XKL's monitors. Any table locations whose

---

[15] The reasons to hold this view are two. First, although large data structures can arbitrarily cross section boundaries, the program cannot. For the program to get from one section to another requires an explicit transfer of program control. The Program Counter has thirty bits, but it counts only the right–most eighteen: when going beyond the end of a section, PC simply wraps around to the beginning of the same section (from location 777777 to location 0). Second, although the page mapping procedures are actually set up in terms of eight supersections of 512 sections each, this distinction is not visible to user programs.

[16] AC references, which can be made by any program, even when virtual page 0 is inaccessible, are made directly to fast memory and require no mapping. All other references to storage addresses are mapped, whether cached or not, whether to storage modules or to mapped device registers.

use is not defined are reserved for future use of the hardware or Monitor software.

Although the virtual–address space is always eight supersections, each containing 512 sections that each contain 512 pages by virtue of the address capability of the instruction and indirect–word formats, the Monitor usually limits the actual address space for a given program by defining only certain supersections, sections, and pages as being accessible. There is no requirement that the virtual–address space be continuous—it can be scattered pages. The monitor also specifies whether each page is writable or not and cacheable or not. To determine the mapping for a given virtual page, the microcode carries out a pointer evaluation procedure that starts at the appropriate entry in the supersection table. If nothing is amiss, the procedure is carried out entirely in microcode—without resorting to monitor software—and it generates the mapping for the specified virtual page. However, if it is discovered during this process that the supersection, section, or page is inaccessible; if the page or any of the map pages required to translate the virtual address is not in memory; or if the program is attempting to write in a write–protected page, the microcode traps to the monitor which must handle the situation; a trap to the Monitor for any of these reasons is is called a "soft page failure". The mapping procedure requires access to the EPT or the UPT, to one supersection table, to one section table, to one page table, and then to the actual data page. Further, the software provides a memory status table in which the microcode keeps track of the use made of these map pages; there may be access to other data as well.

If this complete procedure were carried out for every memory reference, the system would be very slow. To speed up memory references, a cache of recently used address translations is kept in the Pager Translation Buffer (PTB), which the pager consults first in its process of translating a virtual address to a bus address. Hence, it is necessary to go through the whole evaluation process only to translate addresses that do not yet have translations in the PTB. This entire process, called "page refill", is undertaken only when the translation is not present.

3.7.1.1 Pager Translation Buffer

The Pager Translation Buffer (PTB) is a cache of recently used translations. It is organized as 8,192 lines of two–way associative mapping entries. A particular PTB line is selected by 13 virtual–address bits (14–26). Each line contains a pair of mapping entries; each is in this format:

Page Tag			Page Mapping Data					
U	*V*	*Page ID* *VMA bits 6–13*	*W*	*A*	*C*	*D*	*Slot* *Number*	*In–Module Page Address* *Bits 7–26*

The meaning of the fields in the PTB entry is as follows

U	User. A 1 in this bit signifies that this entry is a mapping for a user virtual address; a 0 in this bit means the entry maps an executive virtual address.
V	Valid. A 1 in this bit means this entry is valid. A zero here means that the entry is not valid.
Page ID	This field contains bits 6–13 of the virtual address that is mapped by this entry. The three fields, *V*, *U*, and *Page ID* together form the Page Tag. The Page Tag

Figure 3.3: Virtual–Address Space and Process Table Layout

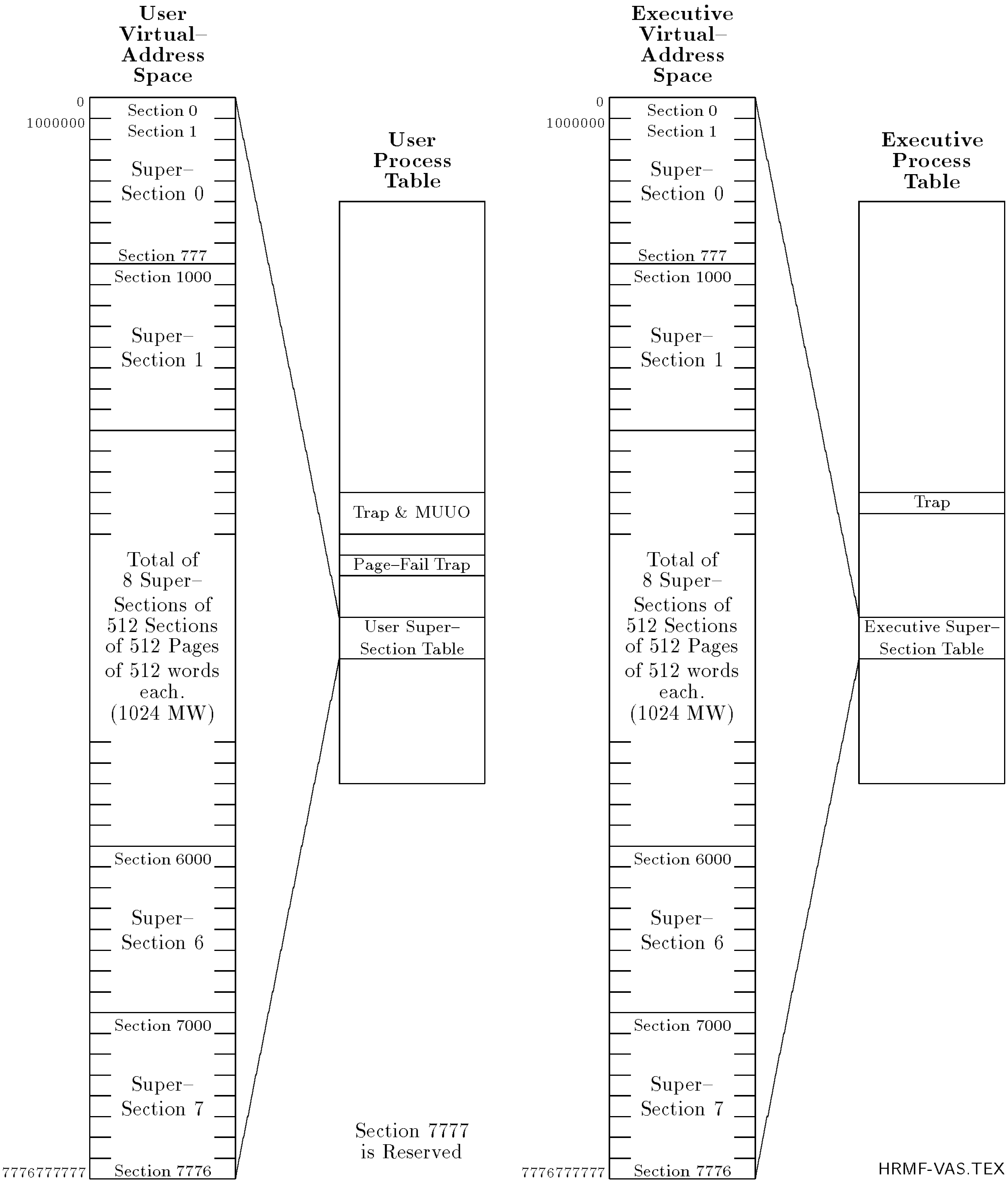
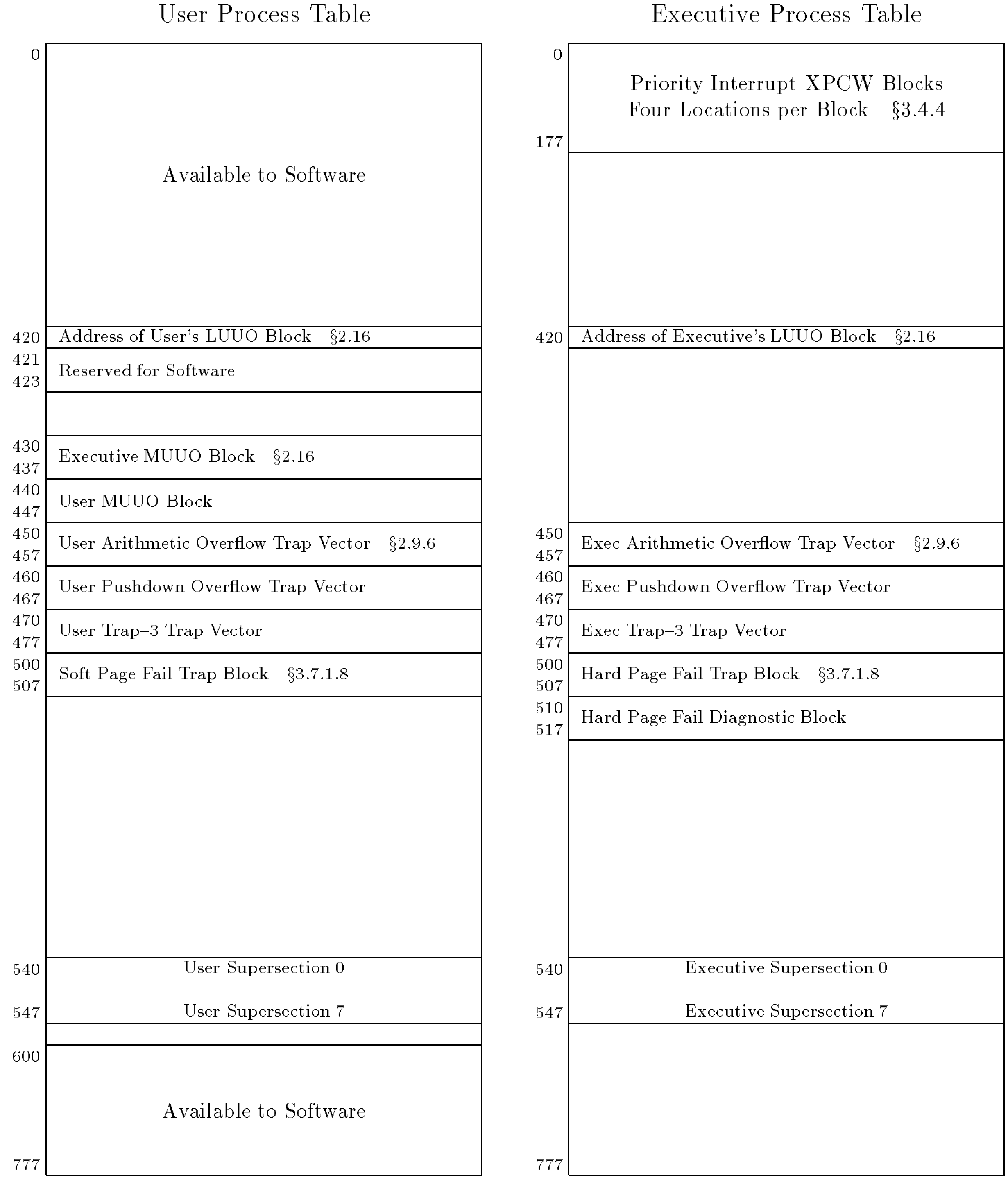


Figure 3.4: Process Table Configuration



identifies which, if either, side of the two–way associative cache contains the correct, valid mapping. The pager uses bits 14–26 of a virtual address to select one line (two entries) of the Pager Translation Buffer. The Page ID (address bits 6–13), and the user/executive bit are sent to the Tag RAM: if it contains a valid match, then data in the Pager Translation Buffer is used to supply the mapping; otherwise, a page refill (see below) is performed.

W	Writable. A 1 in this bit means the page has been written on, so additional writes are allowed without further ado. If a write is attempted when this bit is 0, a page refill will be performed to see if the status of the PTB entry can be changed to permit the write.
A	Address Break. This bit is 1 to signal that the address–break system is active and that references to an address on this page are sought. See §3.7.5. The pager refill microcode will set this bit for a mapping whose mode (executive or user) and virtual–address bits 6–26 match the corresponding bits in the address–break register.
C	Cacheable. This bit is 1 to allow the data on this page to be loaded into the cache. The cacheable bit is loaded by the page refill microcode based on data found in the CST entry for the page.
D	Device. This bit is a 1 to signify that the page mapping represents a device control/status address; 0 for a memory. The difference determines the type of backplane bus cycle to use when reading or writing. Device control/status information may be accessed this way or via the instructions described in §3.1.4. There is no CST entry corresponding to a page in which *D* is 1. An entry in which *D* is 1 should not be cacheable; the pager refill procedure makes no provision to set the *C* bit when *D* is set.
Slot Number	This is the four–bit physical slot number on the backplane to which storage requests will be directed for this mapped address.
In-Module Page Address	This field supplies twenty bits of the in–module address; the remaining in–module address bits (27–35) are supplied directly by the corresponding virtual–address bits.

3.7.1.2 Pager Translation Buffer Diagnostic Instructions

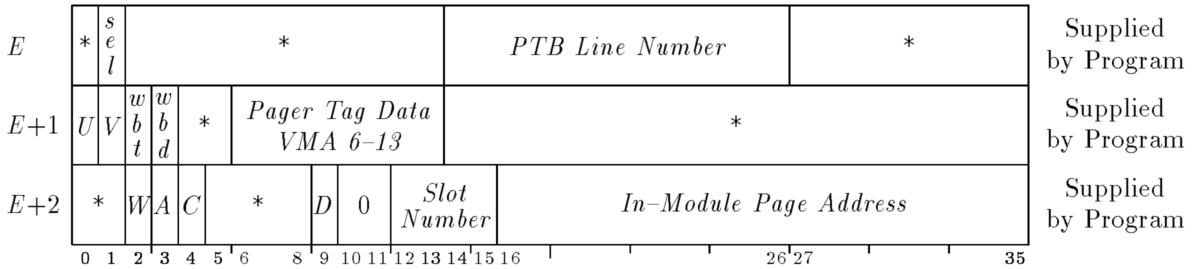
Two instructions are provided to allow testing of the PTB data storage and retrieval facilities. These instructions are diagnostics; they are not expected to be used by the timesharing Monitor.

DWRPTB Diagnostic Write Pager Tag and Data (APR2 16,)

702	16	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

The pager tag and data tripleword at location E, $E{+}1$, and $E{+}2$ are sent to the pager. The data format is shown below.

Page Translation Buffer Tag and Data Tripleword — DWRPTB



Fields marked with asterisk are ignored by the hardware. At the location addressed by E, the fields *sel* and *PTB Line Number* specify the "address" in the PTB into which the data is written. All other fields are the data to be stored in the PTB: the word at E+1 supplies data to be written into the PTB's page tag field; the word at E+2 supplies data to be written into the PTB's page mapping data field. The meaning of the fields is as described here:

sel	Set selection (`PF%SEL==:1B1`): 0 or 1 to select the left–hand or right–hand side of the PTB into which to write.
PTB Line Number	This field (`PF%LIN==:17777B26`) selects one of the 8,192 PTB line pairs into which to write. (In normal use of the PTB, this field is composed of bits 14–26 of the virtual memory address.)
U	User (`PF%USR`): this mapping is for a user–mode virtual–address if U is set; otherwise, this maps an executive–mode address.
V	Valid (`PF%VLD`): this bit is 1 to indicate that a mapping is valid. Otherwise, in the normal use of the PTB, the data in the mapping is ignored.
wbt	Write Bad–Parity Tag (`PF%WBT==:1B2`): set this bit to cause this operation to write bad–parity data in the page tag field. A subsequent read of this tag is expected to reveal the bad parity, which normally would cause a page failure. (However, a DRDPTB instruction would report the parity error without causing a page failure.)
wbd	Write Bad–Parity Data (`PF%WBD==:1B3`): set this bit to cause this operation to write bad–parity data in the page mapping data field. The integrity of the page mapping data is guarded by four parity bits. Each parity bit, when taken with its corresponding data bits, normally provides even parity. When this instruction writes bad parity page mapping data, each parity bit is written as the complement of its normal form. A subsequent read of this data is expected to reveal the bad parity, which normally would cause a page failure. A DRDPTB instruction can read the bad parity data without causing a page failure; it has a field in which it would report the parity error.
Pager Tag Data	This field (`PF%TAG==:377B13`) is the associative key on which PTB cache hits are based. In the normal operation of the PTB cache, this field contains bits 6–13 of the virtual–address which this PTB entry represents.
W	If set, the Writable bit (`PF%WRB`) means that this mapping represents a page that

	has been written on.
A	Address Break Active (`PF%ABA`) is set if this page has address–break trap on it somewhere.
C	Cacheable (`PF%CHB`) is set if the cache is used for references to this page. (In normal operation of the system, this bit must be zero if D is set to 1.)
D	Device (`PA%DEV`) is set if the address is to be presented to the bus as a device control/status address; it is 0 if the address is to be presented as a memory address.
Slot Number	This field (`PA%SLT`) contains the physical slot number of the memory or device to respond at this virtual–address.
In–Module Page Address	This field (`PA%MPA`), when shifted left nine bits, contains the base in–module address for this mapping.

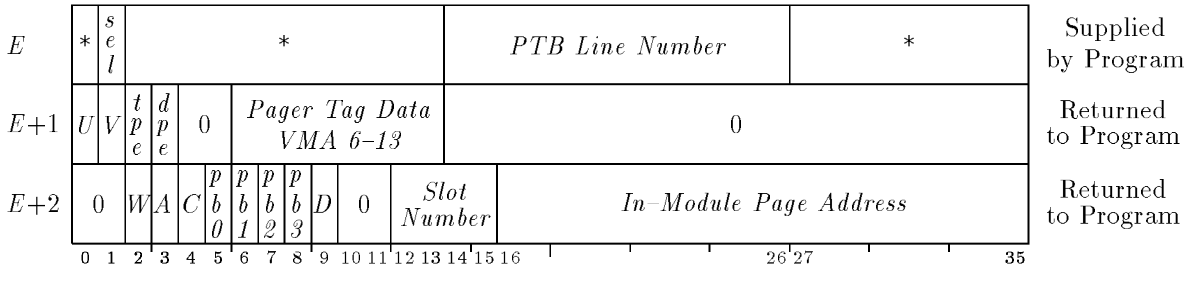
All indicated fields are writable. Note that setting C and D simultaneously is permitted only for diagnostic purpose.

DRDPTB Diagnostic Read Pager Tag and Data (APR2 6,)

702	6	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Select the Pager Translation Buffer address using the contents of E. Return the indicated data in E+1 and E+2.

Page Translation Buffer Tag and Data Tripleword — DRDPTB



Fields marked with an asterisk are ignored by the hardware. The various fields that correspond to those in DWRPTB have the same significance as explained above. The additional fields are interpreted as follows:

tpe	A 1 in this bit (`PF%TPE==:1B2`) signifies a parity error in PTB's page tag field.
dpe	A 1 in this bit (`PF%DPE==:1B3`) signifies a parity error in the PTB's page mapping data field. One or more of the byte parity bits, *pb0*—*pb3*, when taken with its corresponding data, will show odd parity.
pb0	(`PF%PB0==:1B5`) This is the parity bit for page mapping data bits 12–18. Good parity is

even.

pb1 (`PF%PB1==:1B6`) This is the parity bit for page mapping data bits 19–25.

pb2 (`PF%PB2==:1B7`) This is the parity bit for page mapping data bits 26–32.

pb3 (`PF%PB3==:1B8`) This is the parity bit for page mapping data bits 2–4, 9, and 33–35.

3.7.1.3 Use of the PTB

Provided all necessary conditions are satisfied, a virtual–address is translated to a BAW by the paging hardware, using the data in the PTB. Bits 14–26 of the virtual–address are presented to the PTB as a "line number"; bits 6–13 and a bit denoting either executive–mode or user–mode mapping are presented to the PTB as the desired "tag". The portion of the PTB called the Tag RAM will check to see if it has a valid match for the desired tag at the given line number. If it has one valid match, the Tag RAM will report the set selection (one bit, to select either the left–hand side or the right–hand side of the PTB). The data from selected side of the PTB provides the four–bit slot number and twenty–bit in–module page address, which are used together with the nine–bit in–page address (bits 27–35 of the virtual–address) as the backplane bus address for the reference. If the C bit is set, this address will be presented to the cache to see if it contains a representation of the data being addressed. The D bit, the C bit, and the type of reference (read or write) determine the type of backplane cycle to use:

	$D = 0$		$D = 1$	
Type of Reference	$C = 0$	$C = 1$	$C = 0$	$C = 1$
Read	Word Read Request	Cache⋆	Status Request	Illegal
Write	Word Write Request	Cache⋆	Device Control	Illegal

⋆ Cache use may involve a Line Write and/or a Line Read Request

When the pager tag indicates that the appropriate line contains no valid mapping for the requested virtual page and address space, the pager forces a microcode page refill to replace one of the mappings with an appropriate entry. If there is already an appropriate entry but a write access is requested and W is zero, the microcode does a page refill to check whether the PTB entry can be revised to permit the write reference.[17]

The entire Pager Translation Buffer is invalidated when the Monitor selects a new process to run. Specifically, the instructions that load new values for the EBR or UBR invalidate all entries in the Pager Translation Buffer, see §3.7.2.

In addition to setting a new value in the UBR, the context switch to run a different user may involve setting a new AC block context and setting the address break system on behalf of the new process. See §3.7.4, §3.7.2, and §3.7.5.

---

[17] When a read reference causes a page refill, the pager will be loaded with W cleared to zero (unless the page is writable and the CST indicates that the page has already been modified). With W clear, a subsequent write reference will force another page refill. Then, if writes to the page are permitted, W will be set and the memory status (CST entry) of the affected page will be set to "modified". Thus, the Monitor can know which of all the writable pages are actually different than their representation on the backup medium.

3.7.1.4 Page Refill

The refill of a mapping into the Pager Translation Buffer is performed by evaluating various types of pointers found in several kinds of tables. At some point in the procedure, the microcode must encounter

- a "page address" that identifies the section map for the supersection,
- a second page address that identifies the page map for the section, and
- a page address that identifies the on–bus location of the page corresponding to the referenced virtual page.

To identify locations, a Page–Address Word (PAW) is used. Although this data item is called a "word", it actually consists of only bits 5–35. A PAW may be contained in a pointer, in which case bits 0–4 have defined uses. However, a PAW may exist in contexts other than page pointers; in such a case, the software has free use of bits 0–4. A PAW has either of the following two formats:

0	*X*	*D*	0	*Slot Number*	*In–Module Address of Page for Mapping (Bits 7–26)*
5 7	8	9	10 11	12 15	16 35

Non–Zero	*Available to Software (Backup Address)*
5 7	8 35

Bits 5–7 (`PA%NIM==:7B7`) If this field contains all zero bits, the storage medium is "memory":

Bit 8 This bit (marked *X* in the diagram) is not interpreted by the hardware; it is reserved for software use.

Bit 9 (`PA%DEV==:1B9`) If *D* is zero, the page is in memory: the CPU will access this page by bus cycles of the type "Word_Write" and "Word_Read_Request", or "Line_Write" and "Line_Read_Request". If *D* is one, the given page is in the registers of a peripheral device: the device will be accessed by means of the bus cycle types "Device_Control" and "Device_Status_Request".

Bits 10–11 These bits are reserved and must be zero.

Bits 12–15 (`PA%SLT==:17B15`) This field is the bus slot number of the device in which the memory (or registers) reside.

Bits 16–35 (`PA%MPA==:3777777`) This field is the in–module page address, i.e., the page number, within the selected device, that this PAW refers to.

If `PA%NIM` contains a value other than zero, this PAW refers to a page that exists elsewhere than in memory. The microcode will create a not–in–memory page failure and trap to the Monitor. Bits 8–35 are reserved for use by software. In TOPS–20, this field identifies the backup medium and location where the page is stored.[18]

3.7.1.5 Special Tables

In addition to the supersection table (contained in the process table), the section table, and the page map, the page refill makes use of two predefined tables: the Special Page–Address Table (SPT) and the memory status table (known as CST, for "core status table"). These are software–determined tables in memory, but their base addresses are held in registers, the SPB and CSB respectively, known to the page refill microcode.[19]

The Special Page–Address Table contains page addresses that specify shared pages or special pages (for example, those used as page maps or other software–defined tables). The microcode accesses specific entries in the SPT by indexing on a base address (a bus address: 4–bit slot number and 29–bit in–module address) contained in the SPT Base register (SPB). The pointer format provides for an index of twenty–two bits, so the SPT can actually be as large as sixteen sections. The SPT must occupy consecutive physical addresses; it need not be page–aligned. Entries in the SPT are the format of a Page Address Word, as described above. Bits 0–4 of the SPT entry are reserved for software.

Information about the use made by programs of the various memory pages is kept in the memory status table. In every refill, the microcode updates CST entries for the section table, the page map, and the actual page referenced by the program. The CST entry for a page is a word; it is referenced by adding the "linear page number" (LPN) of the page being referenced to a base address (which is a BAW) contained in the CST Base register (CSB). Each page of physical memory requires one word of CST. Note that the microcode does not manipulate CST entries for the process tables, the SPT, or the CST itself unless they are actually referenced by the program; that is, unless the refill is being performed for a program reference to one of the tables. The CST must occupy consecutive physical addresses in one physical memory module (slot); it must be page–aligned.

The calculation of the linear page number, the index to the CST, is performed by the microcode based on the configuration of physical memory. When memory is configured at system initialization, a base linear page number (and validity flag) is associated with each slot that contains memory. The LPN for a given address is calculated by adding the base linear page number of the slot to the in–module page number (bits 7–26 of the in–module address).

CST references are omitted in several instances:

- When the CST Base Register contains zero, as during system initialization.
- When a page address in which D is 1 is encountered. This is a reference to the I/O registers of a peripheral device, not to main memory.

---

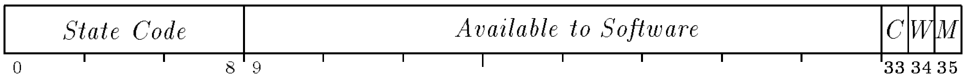
[18] Application note: In TOPS–20, the backup medium and location may be either in swapping space, the "drum", or it may be the disk address of shareable file page. Hence, the format of a backup address is similar to the data found in index and superindex pages of disk files.

[19] All memory tables defined by the pager are in the bus address space; i.e., they have base addresses that include a slot number and a 29–bit in–module address. Of course, to load or access such a table, the Monitor customarily uses paged virtual–addresses. When the base address is limited to a page number (in–module address bits 7–26), the table must be aligned on a page boundary.

- When a page address gives a slot number that does not correspond to a main memory module. This is a reference to memory space within a peripheral device. The CST update is omitted because the base linear page number of the peripheral's slot will be marked as invalid.

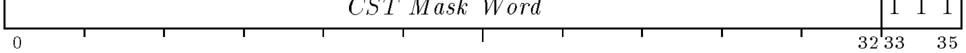
When a CST reference is omitted, the memory reference will not be cacheable.

The status of a page in physical memory is indicated by a CST entry in this format:



State Code	The Monitor keeps a state code in bits 0–8 (`CST%SC==:777B9`) of the entry. Within the state code field, bits 0–5 (`CST%AG==:77B5`) represent the page age, which must be non–zero for the page to be usable, whether it is a program–referenced page, the page map, or the section map. If bits 0–5 are zero, the microcode refill procedure will make an age trap to the Monitor.[20]
Available to Software	The Monitor can use this part of a CST entry for any purpose. In TOPS–20, the Monitor records which processes use the page.
C	`CST%CB==:1B33` A 1 in the *C* bit indicates that the page is cacheable. When a pager refill occurs, the microcode copies this bit into the *C* cacheable bit of the page mapping data held in the PTB.
W	`CST%WB==:1B34` A 1 in the *W* bit indicates write permission; if all other write permissions relating to this page are 1, then the page is writable.[21]
M	`CST%MB==:1B35` A 1 in the *M* bit indicates that the page has been modified since being brought into memory.[22] The microcode sets this bit in the entry for the referenced page—not that for the page map, section table, or supersection table—if the reference is a write and the page is writable. When the microcode sets this bit, it also sets the *W* (writable) bit in the page mapping data held in the PTB.

The microcode updates the CST entry by ANDing the CST mask–word into it and ORing the CST process–use register word into that result. These two words are held respectively in the CST Mask–Word Register (CSTM) and the CST Process–Use Register (PUR). Bits 33–35 in them must be all 1s or all 0s as illustrated, in order to preserve the information that the microcode records in the CST (the *C*, *W*, and *M* bits). Typically, bits 0–5 of the CST mask word are zero so that the data in the PUR will supply a new page age field for the CST entry.



---

[20] The TOPS–20 Monitor uses zero age to denote a page that is idle. Other values encoded in bits 6–8 (while bits 0–5 are zero) denote other transient states in which the page is not yet available for reference.

[21] The write permission for a page is determined by the pager refill procedure, Section 3.7.1.7.

[22] At various times, the Monitor checks the CST to determine which pages have been modified, so that they can be rewritten on the disk. After writing the page to the disk, the Monitor will clear this bit.

CST Process–Use Register	0 0 0
0 … 32	33 35

Indirect pointers make use of tables whose locations are defined entirely by the Monitor. In a single refill, these may include one or more secondary supersection tables, section tables, or page maps. Each such table or map is determined by a bus–page address and a 9–bit index and is therefore a single page. Memory status is kept for the section tables and page maps referenced in the evaluation of indirect pointers. However, memory status is not kept for any secondary supersection tables (process tables) encountered in the evaluation of indirect pointers.

3.7.1.6 Paging Pointers

The microcode evaluates three kinds of pointers: supersection pointers, section pointers, and map pointers. Members of these three classes are identical in form but differ enough in function so that they must be treated separately.

There are four types of each of the three kinds of pointers. Each type of pointer is distinguished by a type code in bits 0–1. Three types are access pointers; i.e., they allow access to the given supersection, section, or page; the other type specifies that there is no access to the given object.

An access pointer has this format in its leftmost five bits:

Type	*W*	*Soft–ware*
0 1	2	3 4

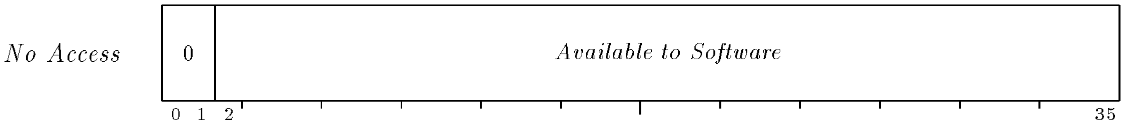
Type `PPW%TY==:3B1` This field determines which type of pointer this is, either no access, immediate, shared, or indirect. (In a no–access pointer, bits 2–35 are available to sofware.)

W `PPW%WB==:1B2` This is the write bit. It indicates whether or not any of the pages in the supersection or section, or the page itself, is writable. Throughout the evaluation procedure, the microcode effectively ANDs this bit from one pointer to the next; if the page is to be writable, then this bit must be must be set to 1 at every step. In other words, if *W* is 1 in the final pointer for the mapping, the page is writable provided that the entire section was also specified as writable by the original section pointer, as well as by the supersection pointer and by every other pointer encountered during the evaluation procedure. (This allows the same page to appear writable to one process and unwritable to another: not all users of the same page need have the same access to the page.)

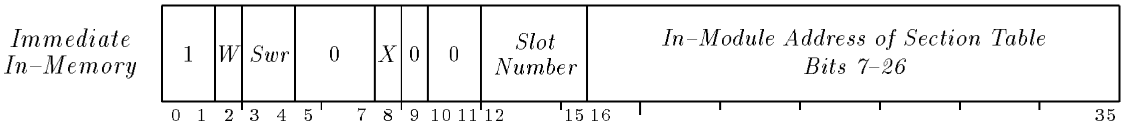
Sofware Bits 3–4 of the access pointers are reserved for software use.

Every access pointer must also contain either a bus–page address or a pointer to an SPT location that contains a bus–page address. A pointer that contains information other than bus–page address (e.g., a no–access pointer or an immediate pointer that specifies a backup address) will stop the page refill and cause a page–fail trap. Such traps do not necessarily result from errors, but they do represent conditions that are beyond the ability of the microcode to handle and so are handed to the Monitor.

Supersection Pointers. Entries in a supersection table are of these types:

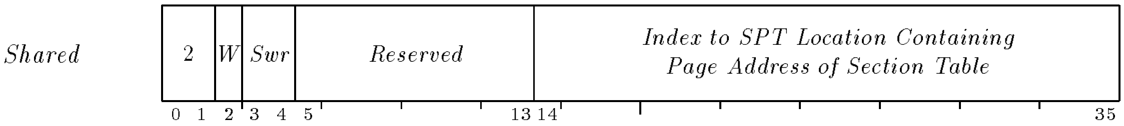


This supersection is inaccessible. `PPW.NO==:0`

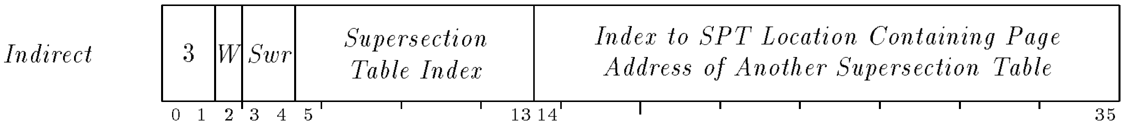


An immediate supersection pointer (`PPW.IM==:1`) contains the page address of the section table. If bits 5–7 (`PA%NIM`) are zero, the section table is in the page specified by bits 12–15 and 16–35 (a bus–page address). Bit 8 (X) is not interpreted by the hardware; it is available for software use.

If bits 5–7 (`PA%NIM`) contain a non–zero value, the section table is not in memory and bits 8–35 are available to software.



The bus–page address of the section table is in the SPT at the offset specified by bits 14–35 (`PPW%SI==:17777777`). This pointer is used for a section table shared by a number of processes. Switching to another map requires changing only the common SPT entry. `PPW.SH==:2`

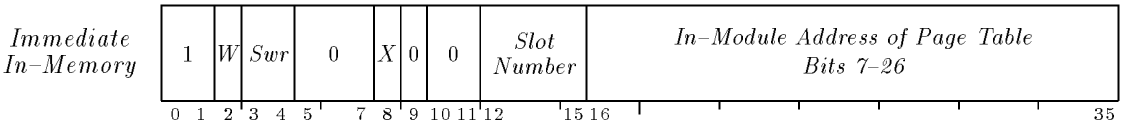


In the SPT location specified by bits 14–35 (`PPW%SI`) is the page address of a secondary supersection table. The next supersection pointer to be evaluated is in that table at the location specified by bits 5–13 (`PPW%PI==:777B13`). A nine–bit field for the Supersection Table Index is provided for compatibility with the section pointer indirect and map pointer indirect types; however, the legal values for the Supersection Table Index are confined to the range 540–547, corresponding to the location of the supersection pointers in a process table. `PPW.IN==:3`

Indirect pointers are used for Monitor reference to per–job and per–process areas. The pointers remain while the secondary supersection table is swapped with the job or process, or the SPT entry is changed.

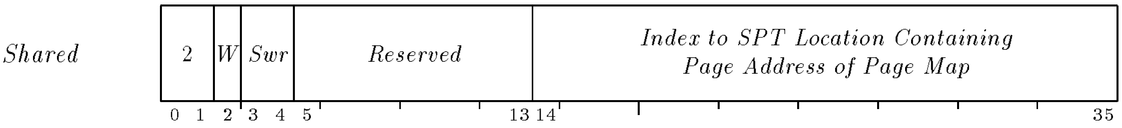
Section Pointers. Entries in a section table are of these types:

This section is inaccessible.



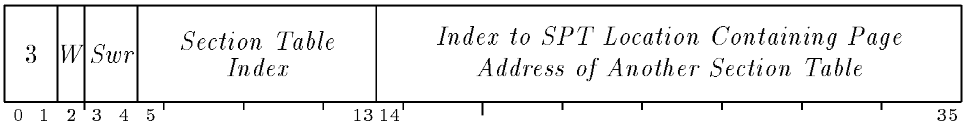
An immediate section pointer contains the page address of the page map. If bits 5–7 (`PA%NIM`) contain zero, the page map is in the page specified by bits 12–15 and 16–35. Bit 8 (X) is not interpreted by the hardware; it is available to software.

If bits 5–7 (`PA%NIM`) contain a non–zero value, the page table is not in memory and bits 8–35 are available to software.



The page address of the page map is in the SPT at the offset specified by bits 14–35 (`PPW%SI`). This pointer is used for a page map shared by a number of processes. Switching to another map requires changing only the common SPT entry.

Indirect

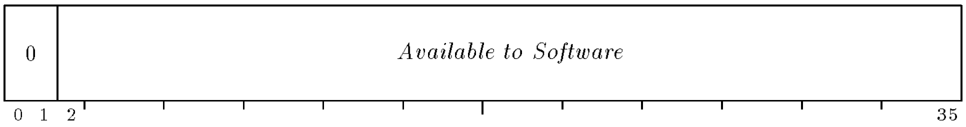


In the SPT location specified by bits 14–35 (`PPW%SI`) is the page address of a secondary section table. The next section pointer to be evaluated is in that table at the location specified by bits 5–13 (`PPW%pI`).

Indirect pointers are used for Monitor reference to per–job and per–process areas. The pointers remain while the secondary section table is swapped with the job or process, or the SPT entry is changed.

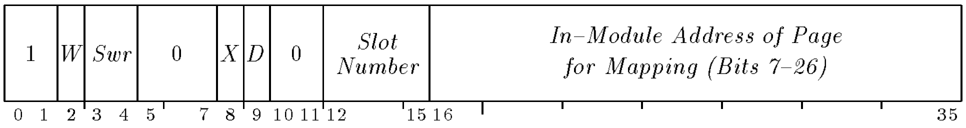
Map Pointers. Entries in a page table are of these types:

No Access



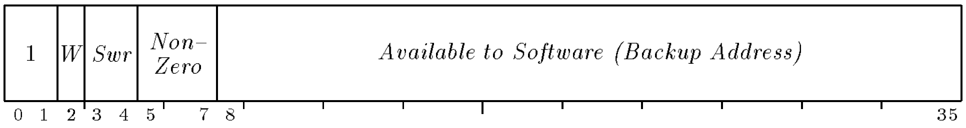
The page is inaccessible.

Immediate In–Memory



An immediate map pointer contains the page address for the mapping. If bits 5–7 (`PA%NIM`) contain zero, the page address for the mapping is specified by bits 9, 12–15, and 16–35. If D is 1, the data is accessed via Device_Status_Request and Device_Control bus cycles that are suitable for accessing device registers; otherwise the data is accessed via Word_Read_Request and Word_Write (or, if cacheable, via Line_Read_Request and Line_Write) bus cycles that are appropriate for accessing main memory. In this format, bit 8 (X) is not used by hardware and is available for software purposes.

Immediate Not In–Memory



If bits 5–7 (`PA%NIM`) contain a non–zero value, the page is not in memory and bits 8–35 are available to software.

Shared

2	*W*	*Swr*	*Reserved*	*Index to SPT Location Containing Page Address for Mapping*
0 1	2	3 4	5 13	14 35

The page address for the mapping of the referenced virtual–address is in the SPT at the offset specified by bits 14–35 (`PPW%SI`). This pointer is used for a page referenced as different virtual pages by different processes. The monitor can move the page for all processes simply by changing the SPT entry.

Indirect

3	*W*	*Swr*	*Page Map Index*	*Index to SPT Location Containing Page Address of Another Page Map*
0 1	2	3 4	5 13	14 35

In the SPT location specified by bits 14–35 (`PPW%SI`) is the page address of a secondary page map. The next page pointer to be evaluated is in that map at the location specified by bits 5–13 (`PPW%PI`).

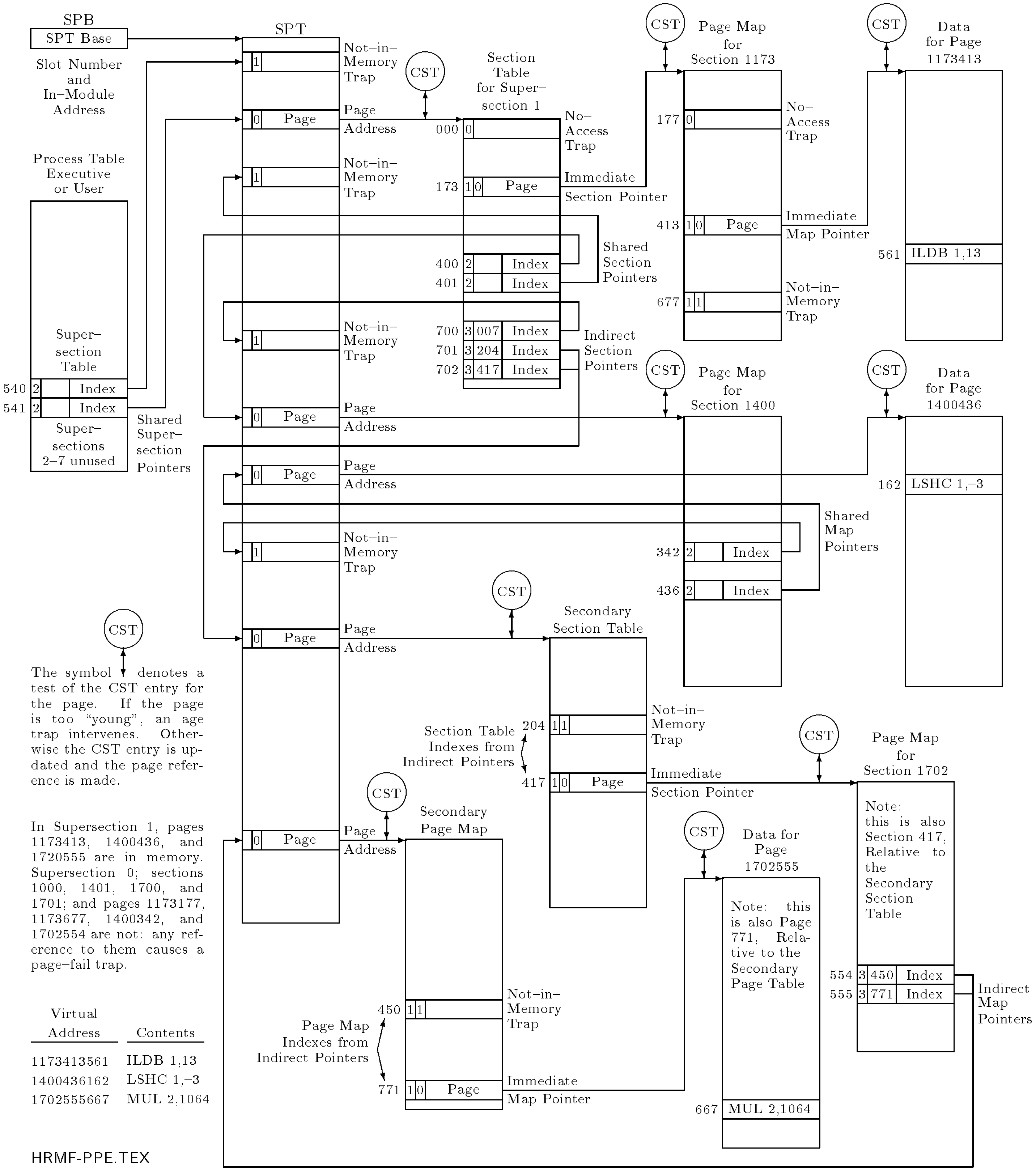
Indirect pointers are used for Monitor reference to per–job and per–process areas. The pointers remain while the second page map is changed when the job or process changes, or the SPT entry is changed.

3.7.1.7 Refill Procedure

When the Pager Translation Buffer lacks a valid mapping for a reference, the microcode that supports the pager must evaluate supersection, section, and map pointers to get the desired mapping. The procedure begins with the pointer for the supersection from the process table, and the pager microcode follows the trail laid by the various pointers, as illustrated in Figure 3.5. At any step, the microcode traps to the Monitor if it encounters a no–access pointer, a page address that indicates the page is not in memory, an age trap, or a write reference to a page that is not writable.

The first part of the procedure, which may go to the SPT or indirectly through it to other supersection tables, begins at location 540 in the process table, indexed by the supersection number (bits 6–8 of the virtual–address, right justified), and evaluates supersection pointers to arrive at the page address of the section table; access to the section table is checked in the CST. Using the page address of the section table indexed by the section number (bits 9–17 of the virtual–address, right justified), the second part of the procedure, which may go to the SPT or indirectly through it to other section tables, evaluates section pointers to arrive at the page address of the page map; access to the page map is checked in the CST. Now, using the page address of the page map, and the number of the referenced virtual page (bits 18–27 of the virtual–address, right justified) as the index, the third part of this procedure retrieves a map pointer and evaluates it. This part may also go to the SPT or indirectly through it to other page maps to arrive at a page address for the mapping; access to this page is checked in the CST. Unless an age trap intervenes, memory status is updated along the way for any section tables and page maps used. If the reference can be made and there is no age trap for the referenced page, its status is updated, including setting the *M* bit in the memory status table if the program is writing. The microcode then constructs the desired mapping, places it in the

Figure 3.5: Page Pointer Evaluation



The symbol ↕ denotes a test of the CST entry for the page. If the page is too "young", an age trap intervenes. Otherwise the CST entry is updated and the page reference is made.

In Supersection 1, pages 1173413, 1400436, and 1720555 are in memory. Supersection 0; sections 1000, 1401, 1700, and 1701; and pages 1173177, 1173677, 1400342, and 1702554 are not: any reference to them causes a page–fail trap.

Virtual Address	Contents
1173413561	ILDB 1,13
1400436162	LSHC 1,–3
1702555667	MUL 2,1064

PTB, and returns to the pending reference.

The mapping data (i.e., the PTB entry) is constructed from the result of the pointer evaluation. The *W* bit is the AND of all the *W* bits seen in the pointer evaluation, AND either the program is making a write reference OR the page has already been modified. The *D* bit, slot number, and in–module address come directly from the corresponding fields of the page address for the mapping. The *C* bit comes from the CST entry for the page being mapped. The *A* bit is set by the page–refill microcode when the address–break system (§3.7.5) is active and the internal address–break register matches the given virtual–address in bits 6–26.

3.7.1.8 Page Failure

When, for any reason, the pager (and its supporting microcode) is unable to make a desired memory reference or when an extended effective–address calculation encounters an incorrectly formatted indirect word, an event known as a "page failure" occurs. For a page failure, the microcode stops the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page–failure trap.[23] The trap operation makes use of certain locations in either the User Process Table or the Executive Process Table. As further explained below, the locations in the User Process Table are used for "soft" page failures and the corresponding locations in the Executive Process Table are used for "hard" page failures.

Any page–failure trap identifies the failed state of the processor by placing a page–failure double word in UPT (or EPT) locations 502 and 503 (`UP.PF0==:502`, `UP.PF1==:503`) and by storing the flag–PC double word in locations 504 and 505 (`UP.POF==:504`, `UP.POP==:505`). The trap completes its operation by setting up new context from the flag–PC double word found in locations 506 and 507 (`UP.PNF==:506`, `UP.PNP==:507`). (The new Previous–Context Section will be set from the section of the trapped–from PC.)

The page–failure double word, stored in locations 502 and 503, will appear in one of several forms, depending on the circumstances of the page failure. Some of the forms of the page–failure double word closely resemble the MAP double word (§3.7.3). All causes of page failure and all formats of page–failure double words report a failure code in bits 12–17 (`PF%FLC==:77B17`) of the first word. The page–failure code specifies the interpretation of the second page–failure word (as a virtual–address, a bus address, or indeterminate). Table 3.2 contains details of each type of failure and specifies how the second page–failure word should be interpreted.

Soft Page Failure

"Soft" page–failures (marked by the trap through the UPT locations) result from conditions generated by the activity of the software. Some are detected during pager refill (e.g., page not in memory, age trap); these the Monitor may be able to correct. Some represent programming errors which, if occurring in a user process, will cause the Monitor to stop a user process and which, if occurring in the Monitor, will cause the Monitor to halt itself.

The contents of the Page–Failure Data Block for soft page–failures, found in the UPT, are shown below:

---

[23] In the case of various multipart instructions (e.g., BLT, IDPB, EDIT), the microcode will update accumulators, memory, and/or machine status appropriately so the instruction can be restarted with correct effect.

Table 3.2: Page–Failure Codes

Soft failures trap through the UPT. Hard failures trap through the EPT, in which they may store additional information.

`PF.AOK==:0`	Soft. No failure (MAP double word): the MAP is successful. In page–failure words, this code is reserved for software.
`PF.AFT==:1`	Soft. Address failure trap: a reference has satisfied the address–break condition (§3.7.5).
`PF.IIN==:2`	Soft. Illegal indirect: an extended effective–address calculation has found an indirect word in which both bits 0 and 1 are one.
`PF.IAD==:3`	Soft. Illegal address: a memory reference has been made to the reserved section 7777.
`PF.MCK==:4`	Hard. Microcode has detected an error condition.*
`PF.OFF==:5`	Soft. Pager is off (MAP double word): MAP occurred while the pager is disabled.
`PF.NLP==:6`	Soft. No logical page: the data in C(E) for a LDLPN instruction does not correspond to a legal page–address word.
`PF.ZPC==:7`	Hard. Zero PC: the data in a trap or an interrupt new–PC word is zero. The second word is the bus address from which the zero was fetched.
`PF.MBT==:10`	Hard. Memory (Device) busy.*
`PF.MTO==:11`	Hard. Memory (Device) timeout.*
`PF.SRF==:12`	Hard. Memory (Device) self–reference.*
`PF.BPE==:13`	Hard. Although this is called a "bus parity error," it actually means that a source device (e.g., a memory) has sent data to the CPU with an indication that the data was bad at the source.*
`PF.HRD==:14`	Hard. Other failure(s).*
`PF.HMC==:15`	Hard. Hard failure delivered subsequent to processing by the macro–console.*
`PF.NA0==:40`	Soft. No access. Supersection.
`PF.NM0==:41`	Soft. Not in memory. Supersection.
`PF.NM1==:42`	Soft. Not in memory. Supersection, share pointer.
`PF.NM2==:43`	Soft. Not in memory. Supersection, indirect pointer.
`PF.NW0==:44`	Soft. Write not allowed. Supersection.
`PF.AT0==:45`	Soft. Age trap. Section Table.
`PF.NA1==:50`	Soft. No access. Section.
`PF.NM3==:51`	Soft. Not in memory. Section.
`PF.NM4==:52`	Soft. Not in memory. Section, share pointer.
`PF.NM5==:53`	Soft. Not in memory. Section, indirect pointer.
`PF.NW1==:54`	Soft. Write not allowed. Section.
`PF.AT1==:55`	Soft. Age trap. Page table.
`PF.NA2==:60`	Soft. No access, Page.
`PF.NM6==:61`	Soft. Not in memory. Page.
`PF.NM7==:62`	Soft. Not in memory. Page, share pointer.
`PF.NM8==:63`	Soft. Not in memory. Page, indirect pointer.
`PF.NW2==:64`	Soft. Write not allowed. Page.
`PF.AT2==:65`	Soft. Age trap. Page.

<table>
<tr><td>500</td><td colspan="6"><i>Reserved</i></td><td><code>UP.PFB==:500</code></td></tr>
<tr><td>501</td><td colspan="6"><i>Reserved</i></td><td><code>UP.PFD==:501</code></td></tr>
<tr><td>502</td><td colspan="6"><i>Page–Failure Word 0 (MAP Word 0)</i></td><td><code>UP.PF0==:502</code></td></tr>
<tr><td>503</td><td colspan="6"><i>Failed Address (MAP Word 1)</i></td><td><code>UP.PF1==:503</code></td></tr>
<tr><td>504</td><td colspan="2"><i>Old Flags</i></td><td><i>0</i></td><td><i>CAC</i></td><td><i>PAC</i></td><td><i>Previous Context Section</i></td><td><code>UP.POF==:504</code></td></tr>
<tr><td>505</td><td><i>0</i></td><td colspan="5"><i>PC of Failed Reference</i></td><td><code>UP.POP==:505</code></td></tr>
<tr><td>506</td><td colspan="2"><i>New Flags</i></td><td><i>0</i></td><td><i>CAC</i></td><td><i>PAC</i></td><td><i>0</i></td><td><code>UP.PNF==:506</code></td></tr>
<tr><td>507</td><td><i>0</i></td><td colspan="5"><i>New PC</i></td><td><code>UP.PNP==:507</code></td></tr>
<tr><td></td><td>0 5</td><td>6 12</td><td>13 17</td><td>18 20</td><td>21 23</td><td>24 35</td><td></td></tr>
</table>

Page–Failure Block at UPT 500

The page–failure double word will appear as shown here:

Page–Failure Double Word in UPT (Soft Failure)

<table>
<tr><td>502</td><td><i>U</i></td><td><i>Reserved</i></td><td>0</td><td>0</td><td><i>T</i></td><td>0</td><td><i>M</i></td><td><i>P</i></td><td></td><td><i>Failure Code</i></td><td><i>Reserved</i></td></tr>
<tr><td>503</td><td colspan="11"><i>Virtual Address (bits 0–5 zero), if M=1, or</i>
<i>Bus Address Word, if P=1, or Zero</i></td></tr>
<tr><td></td><td>0</td><td>1 4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td></td><td>12 17</td><td>18 35</td></tr>
</table>

This format is similar to MAP double word (§3.7.3). In the first word, the microcode provides a specific failure indication in the failure code field (`PF%FLC`), and other useful information is provided in the flag bits. The individual flag bits are decoded as described below. Among the flag bits are `PF%VRT` and `PF%PHY`; these tell how the second word should be interpreted: as a virtual–address, a physical address, or not relevant to the failure.

U User (`PF%USR`): if set, this failure occurred during a reference to user virtual space; otherwise, it occurred during a reference to executive virtual space. Note that, by means of PXCT, the executive can make references to user virtual space; hence, the *U* bit does not indicate that the processor was in User mode at the time of the reference. (The saved PC flags reflect the processor mode.)

T Operation Type (`PF%TYP==:1B7`): *T* is 0 for a read operation and 1 for any operation involving a write. (This flag is not valid for page–failure codes `PF.AOK`, `PF.IIN`, `PF.MCK`, `PF.OFF`, `PF.NLP`, and `PF.ZPC`.)

M Virtual Memory (`PF%VRT==:1B9`): This bit is 1 when the second word contains a virtual–address. If this bit and `PF%PHY` are both zero, the second word is zero: no address is associated with this page failure. No page failures set both this bit and `PF%PHY` to one.

P Physical Memory (`PF%PHY==:1B10`): This bit is 1 when the second word contains a physical address (i.e., a bus address word). If this bit and `PF%VRT` are both zero, the second word is zero: no address is associated with this page failure. No page failures set both this bit and `PF%VRT` to 1. This bit will be set in a Zero PC trap, in a MAP double word that found a valid mapping, and in some hard failures.

Failure Code The failure–code field (`PF%FLC`) contains a unique code that describes the nature of the particular failure. The codes are unique: in all cases the code alone can provide all the information needed to describe the nature of the page failure. The complete set of page failure codes is depicted in Table 3.2.

Hard Page Failure

"Hard" page–failures trap through the page–failure data block in the EPT. A hard failure indicates that one of a collection of miscellaneous hardware–related conditions has occurred. Some of these may be software–induced failures that are expected (e.g., bus timeouts or self–reference while configuring the system), some might be corrected by repeating the operation that failed (e.g., busy timeout), and others represent failures of components or unreliable data transmission that make further operation of the system doubtful (e.g., any of the parity–error indications).

There are seven classes of "hard" failures: bus timeout, bus busy, self–reference, bus parity error, machine check, zero PC, and "other".

- A bus timeout indicates that an expected response (for example, the response to a read request) has not arrived. Unless there is a hardware fault, this usually means that the program has addressed a memory location (or device control location) that does not exist. The bus timeout interval for the XKL-1 processor is approximately 7.5 microseconds.
- The busy condition results from a device returning "busy" in 64 consecutive attempts by the processor to access it. Possibly, the condition will clear up after waiting (devices are "busy" during their power–on initialization sequence), or possibly the device somehow has become "jammed" and it may need to be reset.
- A self–reference error occurs when the CPU makes a reference to a slot and detects that it has addressed itself.
- The backplane bus does not have parity, hence "bus parity error" does not denote an error in the operation of the backplane bus. Instead, "bus parity error" signifies that a device, such as a memory (which keeps track of data parity internally), has sent data on the bus with an indication that the data is suspect, having failed the device's internal parity check.
- Machine check indicates that the microcode has detected a hardware error.
- Zero PC indicates that the data in a trap or an interrupt new PC word is zero.
- The "other" errors include bad bus cycles, scrambled data order (during a cache refill), other bus failures, and other hardware conditions.

The contents of the Page–Failure Data Block for hard page failures, found in the EPT, are shown in Table 3.3. The words at locations 504–507 are the same as those described for a soft page failure. The page–failure double word in locations 502–503 and the other words in locations 500–517 are detailed below.

Table 3.3: Page–Failure Block at EPT 500

<table>
<tr><td>500</td><td colspan="3">0</td><td>BP</td><td>BC</td><td>PE</td><td>WW</td><td>UC</td><td>CT</td><td>CD</td><td>CW</td><td>BR</td><td colspan="6">0</td><td>RT</td><td>TO</td><td colspan="3">TT</td><td>W</td><td>U</td><td>V</td><td>S</td><td colspan="9">0</td><td></td></tr>
<tr><td>501</td><td colspan="36">Processor Data</td><td></td></tr>
<tr><td>502</td><td colspan="36">Page–Failure Word 0 (MAP Word 0)</td><td></td></tr>
<tr><td>503</td><td colspan="36">Failed Address (MAP Word 1)</td><td></td></tr>
<tr><td>504</td><td colspan="13">Old Flags</td><td colspan="5">0</td><td colspan="3">CAC</td><td colspan="3">PAC</td><td colspan="12">Previous Context Section</td><td></td></tr>
<tr><td>505</td><td colspan="6">0</td><td colspan="30">PC of Failed Reference</td><td></td></tr>
<tr><td>506</td><td colspan="13">New Flags</td><td colspan="5">0</td><td colspan="3">CAC</td><td colspan="3">PAC</td><td colspan="12">0</td><td></td></tr>
<tr><td>507</td><td colspan="6">0</td><td colspan="30">New PC</td><td></td></tr>
<tr><td>510</td><td>U</td><td>V</td><td>TP</td><td>DP</td><td colspan="2">0</td><td colspan="8">Tag</td><td colspan="22">0</td><td rowspan="2">Pager Set 0</td></tr>
<tr><td>511</td><td colspan="2">0</td><td>W</td><td>A</td><td>C</td><td colspan="4">PB</td><td>D</td><td colspan="2">0</td><td colspan="4">Slot</td><td colspan="20">Page Number</td></tr>
<tr><td>512</td><td>U</td><td>V</td><td>TP</td><td>DP</td><td colspan="2">0</td><td colspan="8">Tag</td><td colspan="22">0</td><td rowspan="2">Pager Set 1</td></tr>
<tr><td>513</td><td colspan="2">0</td><td>W</td><td>A</td><td>C</td><td colspan="4">PB</td><td>D</td><td colspan="2">0</td><td colspan="4">Slot</td><td colspan="20">Page Number</td></tr>
<tr><td>514</td><td colspan="3">0</td><td colspan="4">Slot</td><td colspan="13">Tag</td><td colspan="2">0</td><td>TP</td><td>DP</td><td colspan="9">0</td><td>V</td><td>0</td><td>M</td><td rowspan="2">Cache Set 0</td></tr>
<tr><td>515</td><td colspan="36">Cache Data</td></tr>
<tr><td>516</td><td colspan="3">0</td><td colspan="4">Slot</td><td colspan="13">Tag</td><td colspan="2">0</td><td>TP</td><td>DP</td><td colspan="9">0</td><td>V</td><td>0</td><td>M</td><td rowspan="2">Cache Set 1</td></tr>
<tr><td>517</td><td colspan="36">Cache Data</td></tr>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td></td><td></td><td></td><td></td><td>32</td><td>33</td><td>34</td><td>35</td><td></td></tr>
</table>

Implementation–specific flags are stored in EPT locations 500–501 and 510–517. (The implementation–specific information is apt to change with revisions to the processor or its microcode.) For the XKL-1, the implementation–specific information is as described here.

For page–failure code `PF.MCK` EPT location 500 (`UP.PFB==:500`) contains data pertaining to the machine check; this data is specific to the XKL-1 microcode.

For page–failure codes `PF.MBT`, `PF.MTO`, `PF.SRF`, `PF.BPE`, and `PF.HRD`, EPT location 500 (`UP.PFB`) contains data specific to the page–failure:

$BP^{\star}$ (`HPF%BP==:1B3`) Both pager sets matched. This error results when the pager tag RAM contains the same, valid data in both halves.

$BC^{\star}$ (`HPF%BC==:1B4`) Both cache sets matched. This error results when the cache tag RAM contains the same, valid data in both halves.

$PE^{\star}$ (`HPF%PE==:1B5`) Memory parity error.

$WW^{\star}$ (`HPF%WW==:1B6`) Wrong word order. The memory returned data in a sequence other than what was requested.

$UC^{\star}$ (`HPF%UC==:1B7`) Unexpected cycle. A device on the backplane bus has addressed the CPU in a cycle that was not expected (e.g., “Read Return” when no read was outstanding).

$CT^{\star}$ (`HPF%CT==:1B8`) Cache tag parity error.

$CD^{\star}$ (`HPF%CD==:1B9`) Cache data parity error. Cached data was discovered to have bad parity as it was being supplied to the CPU.

$CW^{\star}$ (`HPF%CW==:1B10`) Cache writeback parity error. Cached data was found to have bad parity as the line containing the data was being written to memory.

* *BR* (`HPF%BR==:1B11`) The event occurred during a Bus Request. This flag is a modifier to the starred conditions listed above. It signifies that the error was detected during a processor–initiated bus request.

PT (`HPF%PT==:1B15`) The microcode decodes bits 20–22 to set this bit.

PD (`HPF%PD==:1B16`) The microcode decodes bits 20–22 to set this bit.

RT (`HPF%RT==:1B18`) Bus retry exhausted. A cycle was attempted in which the target device responded “busy.” The cycle was thereupon repeated, but the busy condition persisted.

TO (`HPF%TO==:1B19`) Bus timeout. A bus cycle was attempted to a specific device (i.e., to a slot), but there was no response from the device.

TT (`HPF%TT==:7B22`) Trap type. This field is decoded to determine the details of the trap:

1 (`HPF.NW==:1`) Write Not Allowed. (The data in the left half of this word is not meaningful.)

2 (`HPF.AB==:2`) Address Break. (The data in the left half of this word is not mean-

ingful.)

3 (`HPF.AW==:3`) Write Not Allowed and Address Break. (The data in the left half of this word is not meaningful.)

4 (`HPF.HD==:4`) Hard Page Failure. Use the bits in the left half to decode this error.

5 (`HPF.PT==:5`) Pager tag parity error. The bits in the left half are valid.

6 (`HPF.PD==:6`) Pager data parity error. The bits in the left half are valid.

7 (`HPF.PB==:7`) Both pager tag and pager data parity errors. The bits in the left half are valid.

W	(`HPF%W==:1B23`) Write reference.
U	(`HPF%U==:1B24`) User mode reference.
V	(`HPF%V==:1B25`) Virtual–address reference.
S	(`HPF%S==:1B26`) Pager set 1 (or both) matched. If zero, either set 0 matched or neither set matched.

Location 501, (`UP.PFD==:501`) will contain data copied out of the processor's "D to D" latch. This information may be of use to engineers in tracking down the precise nature of the page–failure.

Locations 510–511 and 512–513 contain data describing the state of pager sets 0 and 1, respectively, for the pager entries addressed by bits 14–26 of `UP.PF1`. This data is valid only when `PF%VRT` is set. The data is in the same format as is used by DRDPTB instruction:

U	User.
V	Valid.
TP	Tag parity error.
DP	Data parity error.
Tag	Tag data. (Virtual–address bits 6–13.)
W	Writable.
A	Address break is active on this page.
C	Cacheable.
PB	Data parity bits.
D	Device bit for this map entry.
Slot	The backplane slot number for this map entry.
Page Number	The in–module page number for this map entry.

Locations 514–515 and 516–517 contain data describing the state of cache sets 0 and 1, respectively,

for cache entries addressed by `UP.PF1`. If `PF%PHY` is set, bits 20–32 of `UP.PF1` provide the cache line number. If `PF%VRT` is set, the most significant 7 bits of the cache line number come from bits 29–35 of the *page number* field of the matching set of pager data, and the least–significant six bits come from bits 27–32 of `UP.PF1`. The data is in the same format as provided by DRDCSH.

Slot	The slot number of the memory module represented by this cache entry.
Tag	The cache tag for this entry. (Physical address bits 3–19.)
TP	Tag parity error.
DP	Data parity error.
V	Valid.
M	Modified.

The format of the page–failure double word associated with a hard page failure is

Page–Failure Double Word in EPT (Hard Failure)

502	*U*	*Reserved*	*F*	1	*T*	*R*	*M*	*P*	*I*	*Failure Code*	*Pointer to the Previous Fault*
503	*Virtual Address (bits 0–5 zero), if M=1, or Bus Address Word, if P=1, or Zero*										
	0	1 4	5	6	7	8	9	10	11	12 17	18 35

The *H* bit (`PF%HRD==:1B6`) will be set to 1 in the first page–failure word. The *I* bit (`PF%PI==:1B11`) will be set to 1 if the PI system was on (PIPION) at the time of the hard page–failure. The processor turns off the PI system as it traps the hard page–failure, so the program must examine this flag in order to restore the state of the PI system correctly when returning to the trapped–from process.

If a specific address is associated with the failure, that address will be reported in the second page–failure word. The first page–failure word specifies how the second word is to be interpreted: either as a physical address (i.e., a bus address word), as a virtual–address, or as irrelevant. When the failure is associated with a particular address (virtual or physical), then *T* (`PF%TYP`, bit 7) will be set if the access was for any kind of write.

Failures in virtual–address mode may result from any mapped reference to memory (in which a map entry was found and used to make the reference). When a virtual–address is specified, *M* will be set and *U* and *T* will be as described in the soft failure cases.

Failures in physical–address mode result from instructions such PMOVE and PMOVEM and from the pager refill microcode in physical mode references to the SPT, CST, process tables, section tables, and page tables. When a failure occurs in physical–address mode, *P* will be set to 1 and *T* will reflect the type of access. The BAW of the failing reference will be stored in the second page–failure word. (Note: in the XKL-1 processor, bits 3–5 of the BAW are stored as zero.) *U* is not meaningful.

If *R* (`PF%RTP==:1B8`) is set to one, this is a "recursive" trap: a reference by the refill microcode to a process table, a section table, a page table, the SPT, or the CST failed, or a reference, by trap–processing microcode, to the UPT or the EPT failed. In this case, the field `PF%PFP==:777777` will contain the address in MemA where the previous fault data has been logged.

F is set in a failure condition (one of busy, timeout, or self–reference) which, had it occured in a MAP instruction, would have been reported to that instruction instead of trapping.

The remaining hard failures are those that are detected by the central processor in a way that prevents an address from being associated with the failure. In this situation, *M* and *P* are both zero, as is the second word of the page–failure double word.

No hard page failure sets both *M* and *P* to one.

If the *BR* bit in location 500 is zero, the rest of this block is of doubtful validity.

Hard Page–Failure Fallback

In some circumstances, the processor may determine that a problem beyond the normal scope and complexity of page–failure handling has occurred. In such cases, the processor implements a "fallback" strategy to locate some process capable of reporting the error.

If the processor encounters a page failure while it is attempting a pager refill or a trap, it sets the `RPT` flag. The informal name for `RPT` is "recursive page trap", although this is a misnomer. `RPT` is set when the processor is handling a trap or an interrupt and it encounters a page failure. Fallback strategies will be attempted in this order:

- "I/O Page–Failure" when a page fail occurs during the reference to a page table. This will trap through the current EPT.
- "ROM–Fallback Page–Failure" occurs when a page–failure occurs during an I/O Page–Failure. The processor will attempt to use the ROM copy of the EPT.
- "Console Fallback" Page–Failure occurs when a page–failure occurs during ROM–fallback processing.

General

For a page–failure trap, the Monitor should set up the new flag–PC double word in the trap locations so that the trap switches the processor to executive mode. If able to rectify the situation, the Monitor eventually returns to the interrupted instruction, which starts over again from the beginning or from the save point in a multipart instruction. Provided the Monitor restores the First Part Done flag, even a two–part instruction that has been stopped by a failure in the second part is redone properly. The mechanism for making a correct return and the effects it produces on a BLT or XBLT are the same as for an interrupt; this is described at the end of §3.4. Before returning to the failed instruction, the Monitor must invalidate the Pager Translation Buffer entry for the page and revise the pointers for the new situation. Then, when the instruction is restarted, the pager will do a refill to get the new, correct mapping.

A no–access pointer implies that the section or page does not exist. Otherwise, a soft failure does not necessarily imply that the executing program has done anything "wrong". Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in memory. When the user attempts to gain access to a page that is not there (i.e., a page for which the refill encounters a not–in–memory page address), the Monitor would respond to the failure by bringing in the needed page from the disk, either adding it to the user space or swapping out a page the user no longer needs or has not used recently. Similarly, a process using several sections might not have all of its section tables in memory at the same time; in such a case a legitimate reference to a page might result in a trap because the section table is not in memory. While swapping is in progress, the Monitor runs some other user, returning to the interrupted process when the requested page is available.

A similar situation exists for writability. Keeping track of modified pages is handled by the refill procedure using the memory status table. However, a page may be write protected because it is shared by a number of processes, wherein a change made by one might not be wanted by the others. Thus, in response to a write failure, the Monitor may make a separate writable copy of the page for the sole user of the process that wishes to modify it.

3.7.2 Memory Management

In order to manage memory properly, the executive program must set up process tables and page maps for itself and the various users, oversee the operation of the pager, and select the fast–memory block to be used by each program (usually block 0 for itself). At any given time, accumulator, index register, and fast–memory references are made to that AC block that is assigned as "current". Given a particular processor mode (user or executive) and an appropriate process table and page map, the Monitor effectively defines the address space for a process by specifying the base address for the process table and selecting which AC block is to be "current".

Often, when a user program calls the Monitor, it is to request an activity which requires the executive to gain access to the user address space. To facilitate the crossover from one address space to another, the same instruction through which the Monitor assigns its own current AC block also allows the assignment of an AC block and section for the "previous context," i.e., the context of the process that made the call. These quantities, together with flags that indicate the mode of the caller, allow execution of instructions in the previous context, as will be described more fully in §3.7.4. At any point in time, the previous context is essentially the environment in which the previous program was running. Note that the previous context need not be the user; the same techniques can be exploited following a call from one level of the Monitor to another.

WREBR **Write Executive Base Register (APR1 4,)**

701	4	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Set up the Executive Base Register (EBR) in the pager according to the contents of E.[24] The memory operand, whose format was chosen to be similar to a page address, is of the form:

---

[24] Unlike the former CONO PAG, instruction, which treated E as an immediate quantity, this instruction uses a full–word memory operand.

EP	*LE*	0	*CSH*	0	*Slot Number*	*In–Module Address of Page for EPT (Bits 7–26)*
0	1	2–8	9	10–11	12–15	16–35

LE	`PG%LEB==:1B1` When 1, this flag enables the loading of the EBR from the data in bits 12–35. Loading the EBR invalidates the entire contents of the PTB. Loading the value zero (in bits 12–35) is recognized as marking the EBR as "invalid". If this flag is one, this operation is performed before the (optional) enabling of the pager, as described in the next paragraph.
EP	`PG%ENP==:1B0` When 1, this flag enables the pager for TOPS–20 paging. When the pager is enabled, it operates as described throughout this section. Operation of the XKL-1 processor with the pager disabled is intended only for bootstrapping and some diagnostic purposes; this mode is described in §3.7.2.1. If this flag is set and the EBR is not valid, the processor halts.
CSH	`PG%CSH==:1B9` When 1, this flag enables the PTB refill microcode to look in the cache for EPT data; otherwise, the refill microcode will avoid using the cache for EPT references.
Slot Number	This field provides the slot number, in PAW–format, of the memory that contains the EPT.
In–Module Address	This field provides the in–module page number, in PAW–format, of the page that contains the EPT.

RDEBR Read Executive Base Register (APR1 5,)

701	5	*I*	*X*	*Y*
0–8	9–12	13	14–17	18–35

Read the contents of the EBR and store the result in the word addressed by E. Bit 0 will be returned as 1 if the pager is on. Bit 1 will be returned as 1. Bit 9 will reflect the cacheability of the EPT.

WRUBR Write User Base Register (APR1 3,)

701	3	*I*	*X*	*Y*
0–8	9–12	13	14–17	18–35

Set up the User Base Register in the pager according to the contents of E. The format of the data is as shown below:

0	*C S H*	0	*Slot Number*	*In–Module Address of Page for UPT (Bits 7–26)*
0 8	9	10 11	12 15	16 35

Writing the User Base Register invalidate the entire contents of the PTB. The value zero (in bits 12–35) is recognized as "invalid".

The *CSH* flag (`UB%CSH==:1B9`) when set, enables the pager refill microcode to look in the cache for UPT data; otherwise, the refill microcode will avoid using the cache for UPT references.

RDUBR Read User Base Register (APR1 1,)

701	1	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the contents of the User Base Register in the pager and store the result in the word addressed by *E*. Bit 9 will reflect the cacheability of the UPT.

WRCTX Write Context (APR1 6,)

701	6	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Set new values for the current AC block, the previous–context AC block, and the previous–context section number. The data word is shown below:

S A	*S S*	0	*Current AC Block*	*Previous Context AC Block*	*Previous Context Section*
0	1	2 17	18 20	21 23	24 35

SA `PGLACB==:1B0` When this bit is 1, set the current– and previous–context AC blocks from the fields in bits 18–20, and 21–23, respectively.

SS `PGLPCS==:1B1` When this bit is 1, set the previous–context section from bits 24–35.

Current AC Block

`PGCACB==:7B20` When `PGLACB` is set to 1, this field provides the data to select the current AC block.

Previous–Context AC Block

`PGPACB==:7B23` When `PGLACB` is set to 1, this field provides the data to select the previous–context AC block.

Previous–Context Section

`PGPCSF==:7777` When `PGLPCS` is set to 1, this field provides the data to select the previous–context section number (PCS).

RDCTX Read Context (APR1 7,)

701	7	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the settings of the current– and previous–context AC blocks and the previous–context section into the word addressed by E. The data format is as shown in WRCTX; both SA and SS will be set to 1 in the result.

CLRPT Clear Page Translation Buffer Entry (APR1 2,)

701	2	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Invalidate the PTB mapping entry for the page addressed by E. E is interpreted as a virtual–address. Usually, E is interpreted as an executive–mode address. However, when instruction is executed by PXCT, E represents a user virtual–address.

WRSPB Write SPT Base Address (APR2 10,)

702	10	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Load the contents of location E into the SPT Base register (SPB). The data in bits 1–35 of location E are the same as in a BAW (§3.1.4); however, bit 0 (`SP%CSH==:1B0`), if set, signifies that the SPT is cacheable. If this bit is set, the pager refill microcode will make cached references to the SPT; otherwise, such references will be uncached.

The Special Page Address Table (SPT) is used by TOPS–20 paging when processing indirect and shared page pointers. See §3.7.1.5.

RDSPB Read SPT Base Address (APR2 0,)

702	0	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the SPB and store the result in location E.

WRCSB Write CST Base Address (APR2 11,)

702	11	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Load the contents of location E into the CST base register (CSB). The data in bits 1–35 of location

E are the same as in a BAW (§3.1.4); however, bit 0 (`CS%CSH==:1B0`), if set, signifies that the CST is cacheable. If this bit is set, the pager refill microcode will make cached references to the CST; otherwise, the references will be uncached.[25]

When the CSB contains zero, references to the CST are omitted during PTB refills.

The Memory Status Table (formerly known as the Core Status Table, from which comes the acronym "CST") is used in TOPS–20 paging to determine whether or not a page is cacheable, whether or not it has been modified, and for various other purposes. See §3.7.1.5.

The CST must occupy consecutive physical addresses in one physical memory module (slot); it must be page–aligned.

RDCSB Read CST Base Address (APR2 1,)

702	1	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the CST base register and store the result in location *E*.

WRCSTM Write CST Mask (APR2 13,)

702	13	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Load the contents of location *E* into the CST Mask register.

After each successful test of a CST entry during a pager refill, the refill microcode updates the CST entry by ANDing it with the contents of the CST Mask register and ORing that result with contents of the Process Use Register. See §3.7.1.5 and Figure 3.5.

RDCSTM Read CST Mask (APR2 3,)

702	3	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the CST Mask register into location *E*.

WRPUR Write CST Process Use Register (APR2 12,)

702	12	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Load the contents of location *E* into the CST Process Use Register.

After each successful test of a CST entry during a pager refill, the refill microcode updates the CST entry by ANDing it with the contents of the CST Mask register and ORing that result with contents

---

[25] In a shared–memory multi–processor system, it may be advisable to keep the CST uncached.

of the Process Use Register. See §3.7.1.5 and Figure 3.5.

RDPUR Read CST Process Use Register (APR2 2,)

702	2	I	X	Y
0 8	9 12	13	14 17	18 35

Read the CST Process Use Register into location E.

LDLPN Load Logical Page Number

710	A	I	X	Y
0 8	9 12	13	14 17	18 35

Read the contents of the memory location addressed by E. Interpret bits 9–35 as a page–address word (PAW). Convert the PAW to a linear page number (LPN) and store it in AC.

If the conversion is not successful (i.e., D=1, the given slot does not address a memory, or the translation table is not yet established in MemA at **`AM%LPN`**), the failure will be signalled by a page–failure trap, with page–failure code **`PF.NLP`**.

RDCFG Read Configuration

711	A	I	X	Y
0 8	9 12	13	14 17	18 35

Read two words of configuration information into AC,AC+1. The configuration information is organized on a per–slot basis; E, an immediate quantity, specifies the slot for which to supply information.

The word returned in AC corresponds to the device status word for the device at the given slot, as of the latest time that TDBoot recorded that information. That is, this information is not the current device status but, rather, the status as of the time that TDBoot was run. By convention, a zero word signifies that no device is present in the specified slot.

If the slot contains a memory device that is online, the word returned in AC+1 provides a valid bit (bit 0) and the first linear page number contained in this memory. This is similar to the information that the processor uses to implement the LDLPN instruction.

If the slot contains an XRH-1 device, the word returned in AC+1 contains four 8–bit bytes, right–justified in the word, corresponding to the SCSI Bus ID for each of the four buses implemented in the XRH-1. The Bus ID (and flags) for bus 0 are returned in bits 4–11.

If the slot contains some other device, AC+1 contains device–specific information.

Although slot number 0 does not exist, this instruction is defined when E is zero: AC will contain the slot number of this CPU; AC+1 will contain the total system memory capacity, in pages.

The effect of this instruction is undefined (and reserved) when E is larger than a legal slot number.

The intention of this instruction is to provide an implementation–independent way to access configuration information.

3.7.2.1 Pager Programming

Operation of the XKL-1 processor with the pager disabled is intended only for bootstrapping and some diagnostic purposes. The Boot ROM is the only memory directly addressable while the pager is disabled. When the pager is disabled, the 30–bit addresses generated by the effective–address calculation are passed to the Boot ROM. (The Boot ROM appears as if it were in slot 0.) Although the pager is disabled, it is possible for arithmetic overflow traps, pushdown overflow traps, page–failure traps, and MUUOs to occur. Therefore, the program must set up a vestigial User Process Table to contain addresses of appropriate handlers. A vestigial Executive Process Table is also required for unpaged operation; the vestigial EPT and vestigial UPT may share the same ROM page.

Page–fail traps (reflecting some hardware conditions; e.g., bus timeout, illegal reference to section 7777, parity errors) cannot write the flags and trap PC in ROM, but copies of those quantities are stored in MemA. Other traps do not store information in MemA.

At power turn–on, the microcode invalidates the cache and disables the pager and PI system. The processor is started in executive mode, with AC block 7 (`BTACB==:7`) current, at location 3000 (`BSTART==:3000`) in section `BTSECT` in the Boot ROM. The Boot ROM contains vestigial data structures for an initial Executive Process Table and User Process Table, which TDBoot tells the pager to use (with the pager otherwise disabled) for the sake of establishing the trap vectors for MUUOs and page failures. The unpaged program locates the physical memory and builds the tables needed for the linear page number calculation. Using the PMOVEM instruction, TDBoot sets up process tables and page tables for itself in memory. Thereafter, it writes the User Base Register, then it writes the Executive Base Register, enabling paging. The next instruction is fetched from paged memory. At this point, TDBoot is able to load the selected program into memory. For every section required by the memory image file containing the program, TDBoot creates the section in the linear address space and creates corresponding page tables and a primitive CST.[26] Immediately before running the loaded program, TDBoot sets the CSB, Process Use Register, CST Mask, User Base Register, and Executive Base Register to values that describe the loaded program.

The executive program always runs paged. It may create page tables, CST, SPT, and process tables more to its liking, understanding that doing so is like pulling on the rug that it stands upon. It must set up the first user or users, loading the User Process Table and page maps, bringing in whatever portions of user data and program as are consistent with good working–set management, and setting up the interval timer.

Finally, the Monitor performs a WRCTX to set up an initial AC block for the first user, a WRUBR to set up the first user's address space, and an XJRSTF to start the first user program.

For a call from the user via MUUO, the executive will arrange the MUUO new flags word to set Previous Context User and select appropriate AC blocks for current– and previous–context (usually, the user process is run with AC block 1, and the monitor with AC block 0). For a call from the executive via MUUO, the executive program will make other provisions that cause the caller's ACs to be set up as the previous–context ACs. When an MUUO occurs, the processor automatically

---

[26] The creation of the CST by TDBoot is optional; if no CST is created, TDBoot will set the CSB to zero.

sets the previous–context section from the PC of the MUUO. In handling a MUUO, the executive will leave the User Base Register set as it was for the previous–context, so that the correct address space is accessible for such references.

On transfer of control from the Monitor to a user, no previous–context values need be setup because the user cannot perform PXCT. To switch from one user's program to another, the executive will

- use WRCTX to set the old program's AC block as current,
- save the old program's accumulators,
- switch context to the new program by WRUBR,
- load the new program's accumulators,
- load other user–context variables such as the address–break, and
- resume the new program via an XJRSTF.

The usual procedure for administration of the AC blocks is to assign one block permanently for the use of the current user and to have one block for the use of the Monitor on behalf of users. Other AC blocks may be assigned to different interrupt levels or to special users. When switching from one ordinary user process to another, the first user process's ACs are generally stored in the per–process area and the second process's ACs are restored from its per–process area.

On a change from one process to another, the entire page table is invalidated; this is done automatically by WRUBR. If the system uses shared or indirect pointers, or if several virtual page numbers point to the same physical page, then the table must be invalidated whenever a page is removed from memory or a pointer is removed from a user supersection table, section table, or page map. On the other hand, deletion of a page with a unique mapping requires only a that a CLRPT be given to invalidate the single mapping. In multiprocessor operation, all page tables that may contain duplicate copies of the information that is being changed must be cleared whenever one such page table is changed. CST entries can be used to communicate paging information from one processor to another.

3.7.2.2 Use of Paging to Support TOPS–20

This section, to be added, describes how TOPS–20 uses the hardware facilities to effect such things as file mapping, shared pages, copy–on–write, read–only or full access by one process to another process's memory image, etc.

Although designed for TOPS–20, the processor is expected to be able to run TOPS–10 and other operating systems for the PDP–10 architecture.

3.7.3 MAP Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It is also necessary to determine where a particular virtual page is in the bus address space, for example, to set up command lists for input–output devices. For such purposes the processor has this instruction which, unlike all the other instructions described in this chapter, is not an input–output instruction, even though it is subject to the same restrictions.

MAP Map an Address

257	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the processor is in executive mode or in user input–output mode and the pager is turned on, map the extended page number of the virtual effective–address E and place the resulting physical address and other map data in AC,AC+1. The double word loaded into AC,AC+1 is of the form:

MAP Double Word

U	*V*	*WP*	*A*	*C*	*F*	0		*Failure Code*	
D			*Slot Number*				*In–Module Address*		
0	1	2	3	4	5	6	7	12 17	35

When V (Valid, `PF%VLD==:1B1`) is 1, the mapping returned in AC,AC+1 is valid and the failure code will be zero. The remaining flags in AC have the following significance:

U	User (`PF%USR==:1B0`): if set, this mapping corresponds to a user virtual–address. Otherwise, the mapping is for an executive virtual–address.
WP	Write Permitted (`PF%WRB==:1B2`): if set, this page may be written on.[27]
A	Address Break Active (`PF%ABA==:1B3`): if set, this bit signifies that the address debugging system is active and set to find an address on this page.
C	Cacheable (`PF%CHB==:1B4`): if set, data from (or for) this page is allowed to be in the cache.
F	Hard Map Failure (`PF%HMF==:1B5`): will be 0 in a successful mapping.

Bit 6 will be 0 to be consistent with page–failure double words.

In a valid mapping, AC+1 will contain the bus address word corresponding to the location that was mapped.

If the MAP instruction cannot generate a valid mapping, the result in AC,AC+1 will be in the form of either a "soft" or a "hard" page–failure double word: see §3.7.1.8. U will be set to denote a user virtual–address. The page–failure code will contain the reason why the given address could not be mapped: see Table 3.2. The given virtual–address is returned in AC+1.

The MAP instruction cannot be performed in User mode (except when User In–Out is set); instead, it executes as an MUUO.

---

[27] This bit is not the same as the W bit in a pager diagnostic read, which signifies that the page has been written on; WP is computed by the page refill microcode by means of pointer evaluation.

Notes: In normal operation, the instruction itself will not result in a "soft" page–failure trap because the page refill microcode will return to the instruction instead of trapping. (However, "soft" failures could occur during the effective–address computation.) "Hard" page–failures (busy, timeout, and self–reference) that may have been caused by incorrectly constructed page pointers result in a "hard" page–failure double word being returned in AC,AC+1; such failures will be marked with `PF%HMF` set to 1. "Hard" failures associated with improperly–functioning hardware (e.g., pager tag or data parity error, or memory parity error during the execution of the pager refill microcode, etc.) will result in a hard page–failure trap. Further, a valid mapping could describe an invalid address; such a result can be returned by MAP.

The accumulators have no mapping. When the effective–address E specifies an accumulator, the result will reflect the mapping of the virtual page (either page 0 or page 1000) on which the accumulator's effective–address appears.

3.7.4 Previous–Context Reference

Ordinarily an instruction in a user program is performed entirely in user address space, and an instruction in the executive is performed entirely in executive address space. However, to facilitate communication between the Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by means of the previous–context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it used merely to indicate an XCT with non–zero A bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous context. A PXCT can be given only in executive mode, but the previous context may be the user, such as when following a call to the Monitor by the user. However, the previous context can be the executive, to allow communication between one level of the executive and another, such as when the Monitor gives an MUUO to itself. (Note, it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.)

It is very important to understand exactly which operations are affected by PXCT and which are not. The only difference between an instruction executed by PXCT and an instruction performed in normal circumstances is in the way certain of its memory and index register references are made. To work as a PXCT, an XCT must be given in executive mode, and the bits in its A field (9–12) must not all be zero. (In user mode, A is ignored.) Otherwise, there is no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. That is, the effective–address of the PXCT is computed in the current context and the target instruction identified by E of PXCT is fetched from the current context. Moreover, in the executed instruction, all accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (The executive can access a previous–context accumulator in PXCT merely by addressing it as a fast–memory location: E of the target instruction can address a previous–context accumulator.) If the instruction makes no memory operand references, as in a shift or immediate–mode instruction, and it has no indexing or indirection (i.e., the instruction word gives E directly), then its execution differs in no way from the normal case. The only difference is in memory and index register references. In general, the A field of PXCT specifies whether or not to use the previous context in the computation of the effective–address of the target instruction, and then whether or not to read and/or write memory operands in the previous context.

The previous context is specified by three quantities: PCU (previous–context user), PCS (previous–context section), and PAC (previous–context AC block). The first of these is a PC flag; the other two are components of the processor context (accessible via RDCTX and WRCTX). Following a call by an MUUO, the section in which the calling program was running (its PC section) and the fast–memory block assigned to it appear as the PCS and current–context AC block (CAC) fields in the word read by RDCTX. For the called program, these two quantities can then be assigned as the previous context by WRCTX. The current AC block of the calling program also appears in the process–context word supplied by the MUUO. Various levels of the Monitor may all use fast memory block zero; or a separate block may be assigned to that part of the Monitor that uses PXCTs in handling MUUO calls from other parts of the Monitor.

Just as the current mode is indicated by the User flag, the mode in which the calling program was running is indicated by PCU. At an MUUO call, this flag is set up automatically; alternatively, it may be manipulated via a flag–PC double word. Note that the restrictions on references made in the previous context are those of the previous context—not those of the context in which the PXCT is given—with the single exception that, if the current program is running in section zero, the previous context is also limited to section zero. For example, if the executive executes an instruction that attempts to write in write–protected memory, that reference would fail.

3.7.4.1 Previous–Context Execute

PXCT **Previous–Context Execute**

256	$A \neq 0$	I	X	Y
0 8	9 12	13	14 17	18 35

Execute the instruction found in the word addressed by E, making some references in the previous context. Which references in the executed instruction are made in the previous context are determined by 1s in the A portion of the PXCT instruction word as follows:

Bit	*Reference Made in Previous Context if Bit is 1*
9	Effective–address calculation of instruction, including both instruction words in EXTEND (index registers, address words by indirection); also EXTEND effective–address calculation of source pointer if bit 11 is 1 and of destination pointer if bit 12 is 1.
10	Memory operands specified by E, whether fetch or store (for example, PUSH source, POP or BLT destination); byte pointer; second instruction word in EXTEND.
11	Effective–address calculation of byte pointer; source in EXTEND; effective–address calculation of EXTEND source pointer if bit 9 is 1.
12	Byte data; stack in PUSH or POP; source in BLT; destination in EXTEND; effective–address calculation of EXTEND destination pointer if bit 9 is 1.

Previous–context referencing is useful and reasonable in some instructions but inapplicable to others. There is no trap of any kind, and the effect of using the feature with an instruction to which it does not apply is undefined.

Applicable

MOVE class, XMOVEI
EXCH, BLT, XBLT
Half word, XHLLI
Arithmetic
Boolean
DMOVE class
CAM, CAI classes
SKIP, SOS, AOS classes
Logical Test
PUSH, POP, ADJSP
Byte
MOVSLJ
MAP, CLRPT

Inapplicable

LUUO, MUUO
JUMP, AOJ, SOJ
AOBJN, AOBJP
JSR, JSA, JRA, JSP, JRST
PUSHJ, POPJ
XCT, PXCT
Shift and Rotate
String, except MOVSLJ
UMOVE, UMOVEM

Only the combinations shown in Table 3.4 are permitted.

No jumps can use previous–context referencing. Even among the instructions to which such referencing is applicable, only a limited number of the sixteen possible bit combinations are useful or meaningful. Doing an effective–address computation in the previous context (selected by bit 9 or 11) usually makes sense only when the corresponding data access is also in the previous context (as selected by bit 10 or 12, except 11 or 12 in EXTEND).

When bit 9 is zero, the effective–address computation occurs in the current context. Despite the previous–context section being zero, the effective–address computation can "escape" to a non–zero section when this bit is zero.[28] (This may be a programming error.) If this behavior is not desired, it can be avoided either by setting bit 9, which forces the effective–address computation to behave according to the rules of section zero, or the components of the effective–address computation (usually an index register) can be adjusted to compute a section zero address. This caution applies also to the UMOVE and UMOVEM instructions.

Execution of a BLT by a PXCT is limited to these three cases:

- Where all operands, regardless of context, are in section zero.
- Where the previous–context fast–memory block is being saved in or restored from the current context, which may be any section. (However, remember that, regardless of context, a BLT–given in–section address in the range 0–17 always refers to fast–memory. Hence, an AC block can never be saved in or restored from the first sixteen storage locations in any section.)
- Where all operations are confined to a single section in the previous context, as would be the case when clearing a user page.

In all other circumstances, XBLT must be used instead. When XBLT is performed by PXCT, PCS is ignored because AC+1 and AC+2 are interpreted as global addresses.

---

[28] The KL10 has special hardware to force a section zero reference in this case when PCS is zero; the TOAD-1 System does not.

Table 3.4: XKL-1 Permissible PXCT Addressing Modes

Instructions	*9*	*10*	*11*	*12*	*References in Previous Context*
General, Immediate, MAP†, CLRPT†	0	1	0	0	Data
	1	1	0	0	E, Data
BLT	0	0	0	1	Source
	0	1	0	0	Destination
	0	1	0	1	Source, Destination
	1	1	0	0	E, Destination
	1	1	0	1	E, Source, Destination
XBLT	0	0	0	1	Destination
	0	0	1	0	Source
	0	0	1	1	Source, Destination
Stack	0	0	0	1	Stack
	0	1	0	0	Memory Data
	0	1	0	1	Memory Data, Stack
	1	1	0	0	E, Memory Data
	1	1	0	1	E, Memory Data, Stack
Byte	0	0	0	1	Data
	0	0	1	1	Pointer E, Data
	0	1	1	1	Pointer, Pointer E, Data
	1	1	1	1	E, Pointer, Pointer E, Data
MOVSLJ	0	0	0	1	Destination
	1	0	0	1	$E(=Y)$, Destination Pointer, Destination
	0	0	1	0	Source
	1	0	1	0	$E(=Y)$, Source Pointer, Source
	0	0	1	1	Source, Destination
	1	0	1	1	$E(=Y)$, Pointers, Source, Destination

† MAP and CLRPT are considered to be memory reference instructions (even though data is neither read nor written in memory), because E is interpreted by hardware as a memory address.

3.7.4.2 Other References to the Previous Context

In addition to PXCT, the XKL-1 provides the UMOVE and UMOVEM instructions that move data between the current and previous contexts. These instructions are equivalent in effect to PXCT 4,[MOVE AC,E] and PXCT 4,[MOVEM AC,E], respectively, but they are somewhat faster to execute. Note that the effective–address computation is performed in the current context (which is problematical is some situations); only the memory operand is in the previous context.

Where appropriate, these operations are preferred because they promote instruction stream locality and reduce memory references in comparison to their PXCT counterparts.

UMOVE Move From User Context

716	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the effective–address computation to determine E in the current context. Using E as an address in the previous context, copy the location addressed by E to AC.

UMOVEM Move To User Context

717	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the effective–address computation to determine E in the current context. Using E as an address in the previous context, copy AC to the location addressed by E.

3.7.5 Address Debugging

The address–failure, or address–break, feature of the pager implements the traditional program debugging technique of trapping a selected type of memory reference to a particula storage location. (The address–failure mechanism does not trap fast–memory references of any kind.) It may be used to determine whether a given program is modifying a particular location, is executing a particular piece of code, or is simply using a particular piece of data.

WRADB Write Address–Break Data (APR0 3,)

700	3	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Select the break address and the break conditions according to the contents of E, as shown below.

U	*I*	*R*	*W*		*Break Virtual Address*
0	1	2	3	4 5	6 35

The break conditions are selected by 1s in bits 0–3, as follows:

U	User (`ADB%US==:1B0`). If 1, this break is set for a user virtual–address; otherwise, it is for an executive virtual–address.
I	Instruction fetch (`ADB%EX==:1B1`). If 1, this break is set for an instruction fetch. The trap will occur when data is fetched from the selected address under control of the PC. (This does not trap if the location is being executed by XCT.)
R	Read (`ADB%RD==:1B2`). If 1, this break is set for a data read. The trap will occur when data is fetched from the selected address in any circumstance other than under the control of the PC. (Traps if this location is executed by XCT; traps if this location is read during an effective–address calculation, etc.)
W	Write (`ADB%WR==:1B3`). If 1, trap when an attempt is made to write data in the selected address.

Break Virtual Address

Address (`ADB%AD==:7777777777`). This is the virtual address being watched by the address–break system.

Note: the address–break system depends on the pager; the break address is a virtual–address, and the pager must be on for address–break to function. The address–break system cannot trap references by peripheral devices, references which bypass the pager (e.g., PMOVEM), or references through another map in which the target location appears at a different virtual–address. A CLRPT instruction should be given to flush any mapping the PTB may already have for the page on which the break is being set. (If the UBR has recently been changed, or if it is about to be changed, the CLRPT may not be necessary.)

XKL-1 implementation note: this description notwithstanding, the XKL-1 hardware recognizes only two conditions: any reference, and write. The selections *I* and *R* are implemented as break on any reference. Monitor software, activated to handle the address–failure trap, will examine the trap data to determine if the address–break is to passed onward to the user program.

The address–break system is disabled by selecting no break conditions.

Whenever the processor attempts one of the selected types of reference to the virtual–address specified by the break address in the specified address space, a page failure results unless the Address Failure Inhibit (AFI) flag is set. This flag, bit 8 of the program flags, can be set only by an instruction that restores flags. When set, it prevents an address failure during the next instruction—the completion of the next instruction automatically clears it. (If an interrupt or trap intervenes, the effect of the flag is deferred: it is saved and cleared when the flags and PC are saved; it is restored when the interrupt or trap returns to the interrupted program.) The AFI flag affects the instruction following a JRST in which it is restored with PC. Using the inhibit flag, the Monitor (or user–mode address–break handler) can return to the instruction that caused an address failure and execute it once without getting another address–break.

Since the address–break facility is entirely under the control of the privileged WRADB instruction, it can be used quite flexibly for the the executive to debug its own routines or to debug a single user program without bothering the executive or other users. The break conditions presently in effect can be ascertained by giving the following instruction:

RDADB Read Address Break Data (APR0 1,)

700	1	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the current break conditions and address into the location addressed by E. The data is in the same format as used in WRADB.

3.8 Timing

The XKL-1 includes a subsystem for keeping track of the passage of time and providing periodic interrupts.

3.8.1 Interval Timer Programming

The interval timer is an 8–bit hardware counter that is incremented every 128 microseconds. The interval timer loads itself with the 1's complement of the interval period (a number in the range 1 to 127) and counts up by 1 at each "tick" of the 128–microsecond clock. When the counter contains the value 127, the next tick produces an interrupt on the selected priority level and the interval timer reloads itself with the 1's complement of the interval period.

A WRITM instruction that clears the interval timer and loads the interval period is asynchronous with the ticking of the 128–microsecond clock. This means that the next tick of the 128–microsecond clock may occur at any time from 0 to 128 microseconds following a WRITM instruction. Thus, the first interrupt (after the WRITM) may occur from 0 to 128 microseconds earlier than the specified period. The second and subsequent intervals will be of the correct duration, until the interval timer is reinitialized by another WRITM.

An interval period of 1 specifies interrupts every 256 microseconds; the maximum period, 127, specifies interrupts every 32.768 milliseconds.

A WRITM instruction that specifies a new interval period should also request that the interval timer be cleared, because, if the interval period is set without clearing the interval timer, the counter continues counting until the currently loaded interval period is exhausted.

The interval timer can be used for any purpose by the software, but it is employed principally to signal the Monitor when a user process ties up the system for too long a time without blocking. There is just one flag, "Interval Done", which is set when the counter reaches the value the program specifies as its period. When the counter reaches that value, it resets itself (to the 1's complement of the specified interval period) and continues to count toward 127 again. Setting Interval Done requests an interrupt on the priority level assigned to the timer; the processor accepts the interrupt from the interval timer by performing the equivalent of XPCW with location 100 (through 103) of the Executive Process Table as the effective–address.

On power–on reset, the interval period is set to zero, disabling the interval timer.

WRITM Write Interval Timer (APR2 14,)

702	14	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction decodes E and performs selected functions, such as setting the interval period, setting the priority level, and otherwise controlling the interval timer, as described below.

Decode E and perform the functions specified by bits 18–21, as shown:

CIC	*CIF*	*SIP*	*SPI*	*Interval Period*		*Priority Level*
18	19	20	21	22 29		33 35

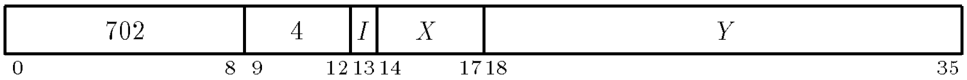
CIC	Clear Interval Counter (`TIMCIC==:1B18`). Initialize the interval counter to the 1's complement of the interval period (use either the previously set interval period or, if *SIP* (bit 20) is 1, use the value in the Interval Period field).
CIF	Clear Interval Done (`TIMCIF==:1B19`). After receipt of an interval–done interrupt, the program must clear the interval–done flag in order to receive an interrupt on the completion of the next interval period.
SIP	Set Interval Period (`TIMSIP==:1B20`). If this bit is 1, set the interval period from bits 22–29. The interval counter should be cleared in the same instruction that sets the interval period.
SPI	Set interval timer Priority Interrupt level (`TIMSPI==:1B21`). If this bit is one, set the priority interrupt level from bits 33-35.
Interval Period	If *SIP* (bit 20) is one, set the period between interrupts from the value contained in this field (`TIMPER==:377B29`). When the counter has incremented the indicated number of times, it will set Interval Done and request an interrupt; the counter will reset itself to count another interval of the same length. The period between interrupts is $128 \times (interval\_period + 1)$ microseconds. Note that the first interrupt may occur as much as 128 microseconds earlier than this formula would indicate. An interval period of zero disables the interval counter; the priority level should also be set to zero in this case.
Priority Level	If *SPI* (bit 21) is 1, set the priority level assignment as specified by bits 33–35 (`TIMPIA==:7`).

Some cleverness is needed when programming the interval timer, as the interval periods do not map readily to more traditional units. For example, to interrupt (on the average) sixty times per second, the calculation $1000000/(60 \times 128) = 130.208333$ suggests that a period of 130×128 μs is just a bit too short for long–term accuracy. Use a period of 130 (decimal) twenty–three times and then use a single period of 135:

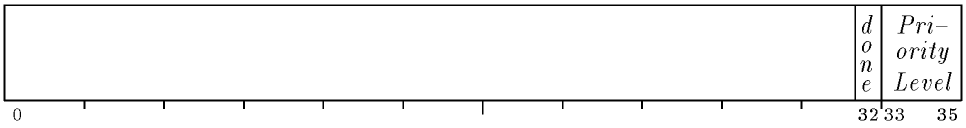
$$24/60 = .4\,\text{sec} = 400000\,\mu\text{s} = 128\,\mu\text{s} \times (23 \times 130 + 135)$$

Thus, the long–term average is 60 interrupts per second. If the long interval is especially unappealing, a sequence of values 130 or 131 can be fashioned where in 24 consecutive values some particular five are 131 and the others are 130.

RDITM Read Interval Timer (APR2 4,)



Read the condition of the interval timer into the word addressed by E. The data is returned in this format:



Bit 32 being set indicates "Interval Done" (`TIMDON==:1B32`). This can be cleared by **WRITM** with *CIF* (bit 19, `TIMCIF`) set.

3.8.2 Time–Base

The time–base keeps a 60–bit count, in which only the low–order sixteen bits are implemented in hardware. The actual counting is done in a 16–bit hardware counter, while the overall count is kept in a double word in MemA. Microcode increments the double word time–base count (at bit 43) when the hardware counter overflows. When the software requests that the time–base be read, the processor combines the double word in MemA with the value of the hardware counter.

A double word count is a 60–bit unsigned quantity. The entire first word comprises the high–order thirty–six bits; the low–order twenty–four bits are in bits 1–24 of the second word.[29] Eleven bits are reserved for expansion at the low–order end so that future systems with higher–resolution clocks may use the same format. The format of the time–base count and its relationship to the hardware counter is as shown here:

---

[29] Remember, it is a property of twos complement arithmetic that the sign can be used as an additional magnitude bit in an unsigned number. However, as the hardware is set up for signed arithmetic, bit 0 of any low–order word must be skipped.

High–Order Part of Count	0	*Low–Order Part of Count*		*Reserved*
0 35	0	1 8 36 43	9 24 44 59	25 35
			Counter	

The time–base is a double word count, kept in locations 310–311 (`AM%TIM`) of MemA. The hardware counter counts elapsed time in intervals of 500ns (a rate of 2 MHz).[30] The time–base is implemented as a 16–bit counter; when that count overflows (about 31 times per second), the microcode increments the time–base in MemA by adding 1 at bit 8 of the low–order part of the count.

The XKL-1 provides the following instruction to read the hardware time–base:

RDTIME Read Hardware Time–Base (APR2 5,)

702	5	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the hardware time–base count and the double word time–base count (in MemA). Combine these by placing the value of the hardware time–base count in bits 9–24 of the low–order word of the double word result which is stored in E,E+1.[31]

The time–base guarantees a monotonic increasing function during the operation of the system. The time–base is not necessarily related to the time of day or the calendar.

The time–base (as an unsigned integer) overflows after about 18 thousand years.

WRTIME Write Hardware Time–Base (APR2 7,)

702	7	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

The Monitor may initialize the double word time–base count to a value of its choice to relate the time–base to to the calendar. The double word contained in E,E+1 is copied to the time–base count (in MemA). Bits 9–35 of the data supplied in E+1 are not significant and their effect is not defined.

3.8.3 Keep–Alive Timer

The processor and microcode provide a “watchdog timer” known to the operating system as the “Keep–Alive” timer. The purpose of this timer is to detect the circumstance of software or hardware malfunction that prevents the normal operation of the system. This is accomplished by the software

---

[30] The clock actually counts one interval of 480ns followed by two intervals of 510ns each, thus averaging one count each 500ns.

[31] The processor microcode that implements this instruction is careful to be sure that the hardware value does not include an unprocessed overflow.

setting the timer and then periodically resetting the timer before it expires; if the software fails to reset the timer within the allotted period, a Keep–Alive interrupt (see Table 3.1, page 220) is performed to jolt the software from a presumably malfunctioning state.

The Keep–Alive timer is not counted while the processor is halted, nor is it counted while `CF%KPA` (accessed via WCTRLF) is disabled.

The action of the processor when the Keep–Alive interval expires is called an "interrupt"; however, this action is unlike other interrupts: it has no priority level; it takes effect regardless of the state of the PI system; and it does not change the state of the PI system. This action is also similar to a trap; however, unlike other traps, it is not synchronous with the execution of the running program.

WRKPA Write Keep–Alive Timer (APR0 13,)

700	13	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction stores the immediate value *E* as the initial value of the microcode "watchdog timer"; this value is stored in the MemA location `AM%KPA`.

While the processor is running a program (`MS%RUN` set, Section 3.2), at approximately 32.8 millisecond intervals, the processor microcode examines the contents of `AM%KPA`. If the location contains zero, no other action is taken. Otherwise, the contents are decremented and the new value is stored in `AM%KPA`. If the value becomes zero after decrementing, the processor performs the Keep–Alive interrupt.

This facility is not suitable for fine timing: since setting a new value into `AM%KPA` by this instruction may occur at any point within the interval between decrements, the initial decrement may occur at any time between 0 and 32.8 milliseconds.

The maximum count, 777777, provides an interval of approximately 140 minutes.

3.9 Other CPU Controls and Status

3.9.1 Error Monitoring

WRAPR Write Processor Conditions (APR0 4,)

700	4	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction decodes *E* to control the processor. "APR" in the instruction mnemonic stands for "Arithmetic Processor," the traditional name for the CPU. The effective–address bits are used as follows:

SPI	*IOR*	*EPC*	*DPC*	*CPC*	*SPC*	*Reserved*	*Int*	*Sht*	*PwF*		*Pri–ority Level*
18	19	20	21	22	23	24 28	29	30	31		33 35

SPI	`AP%SPI==:1B18` If 1, set the PI level assignment for the CPU from bits 33–35.
IOR	`AP%IOR==:1B19` Reset CPU internal devices: clear processor flags; clear interval timer; clear PI level assignment in the console terminal status.
EPC	`AP%EPC==:1B20` Enable the Processor Conditions selected by bits 24–31 to cause interrupts. (The result of this instruction is undefined if both bits 20 and 21 are on.)
DPC	`AP%DPC==:1B21` Disable the Processor Conditions selected by bits 24–31. The selected conditions will not cause interrupts. (The result of this instruction is undefined if both bits 20 and 21 are on.)
CPC	`AP%CPC==:1B22` Clear the Processor Condition flags selected by bits 24–31. (The result of this instruction is undefined if both bits 22 and 23 are on.)
SPC	`AP%SPC==:1B23` Set the Processor Condition flags selected by bits 24–31. (The result of this instruction is undefined if both bits 22 and 23 are on.)
flags	`AP%FLG==:377B31` Flags that may be selected by the *EPC*, *DPC*, *CPC*, and *SPC* functions, described above. These bits represent individual processor flags that can be set, cleared, enabled, or disabled with the appropriate combination of bits 20–23.
Reserved	Bits 24–28 are reserved for use as future processor flags. Bit 32 is also reserved.
Int	`AP%INT==:1B29` Console Interrupt (Flag or Enable).
Sht	`AP%SHT==:1B30` System Shutdown (Flag or Enable).
PwF	`AP%PWF==:1B31` Power Failure or Thermal Warning (Flag or Enable).
Priority Level	`AP%PIA==:7` If bit 18 is set, set the PI level assignment for the CPU.

Note: WRAPR does not apply the selected operations in any particular sequence. Only operations that are not order–dependent should be performed in one instruction.

RDAPR Read Processor Conditions (APR0 5,)

700	5	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction reads the processor flags into the memory location addressed by E. The resulting status word is depicted below:

	Reserved	*I t E*	*S t E*	*P w E*		Rsvd	*N V B*	*I n t*	*S h t*	*P w F*	*I R q*	*Pri– ority Level*
	6 10	11	12	13		24 27	28	29	30	31	32	33 35

Bits 6–13 report which APR conditions are enabled to cause interrupts; of these, bits 6–10 are reserved for future use.

Bits 24–31 are the Condition flags. These flags indicate which APR conditions prevail when this instruction is executed; of these, bits 24–27 are reserved for future use. If a flag and its corresponding enable are both set, an interrupt will be requested.

The defined bits in the status word are decoded as follows:

ItE	If 1, the processor is enabled to accept an interrupt command from the console terminal.
ShE	If 1, the processor is enabled to accept a shutdown command from the console terminal.
PwE	If 1, the processor is enabled to accept an interrupt to signal that an AC Power Failure (or Thermal Warning) has occurred.
NVB	`AP%NVB==:1B28` NVRAM Battery is low flag. This flag is set by the processor microcode when its initialization sequence detects that the battery supplying the NVRAM is low. The data in the NVRAM is suspect.
Int	This is the Console Interrupt Request flag. It is set by the console command processor when it wants to inform the CPU of a change in the status of the console. Generally, this means the system operator has issued a command to enter kernel mode DDT.
Sht	This is the Shutdown Request Flag. It is set by the console command processor in response to the system operator's Shutdown command.
PwF	This is the Power Failure Flag. It is set from the assertion of the PFAIL– backplane signal. This signal signifies that either the AC power is low or that a Thermal Warning is present. When this flag is set, the CPU has a short period of time[32] during which it can bring the system to quiescence before DC power is shut down. See **RCTRLF**.
IRq	`AP%IRQ==:1B32` Interrupt Request. This bit is the inclusive–OR of the AND of bits 24–31 (condition flags) with bits 6–13 (interrupt enables). That is, this bit is set if any condition flag and its corresponding interrupt enable are both set.
Priority Level	Priority interrupt assignment (level) for CPU.

SZAPR Skip if Zero, Processor Conditions (APR0 6,)

700	6	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

---

[32] The time available varies with the state of the battery charge.

This instruction tests bits 18–35 of the processor conditions (as would be read by RDAPR) against bits 18–35 of E. If all the status bits selected by 1s in E are zero, the next instruction in sequence is skipped.

SNAPR Skip if Non–zero, Processor Conditions (APR0 7,)

700	7	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction tests bits 18–35 of the processor conditions (as would be read by RDAPR) against bits 18–35 of E. If any status bit selected by a 1 in E is not zero, the next instruction in sequence is skipped.

3.9.2 Control Flags

Additional processor, system, and console flags and functions are controlled and/or monitored by the XKL-1 by means of the following two instructions.

WCTRLF Write Control Flags (APR0 10,)

700	10	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

The word at location E contains a pair of command bits and several flag or function bits as depicted below. Those bits that correspond to read–only indicators cannot be changed by this instruction; they are read by RCTRLF.

Control Flags for WCTRLF and RCTRLF

Set	*Clr*	*DMP*	*DIA*	*BOO*	*ATO*	*DBG*	*Reserved*	*ACF*	*THW*	*BTF*	*BTL*	*NPW*	*KPA*	*NDC*	*SAL*	*RI*	*CD*	*DTR*	*LED3*	*LED2*	*COK*	*APE*	*LED1*	*RTS*	*LED0*
0	1	2	3	4	5	6	7 17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Set `CF%SET==:1B0` When this bit is 1 the flags selected by 1s in bits 2–6, 23–24, and 26–35 will be set to 1, and if bit 25 is 1, the function selected by that bit will be performed. If this bit and *Clr* are both 0, this instruction has no effect. If this bit and *Clr* are both 1, this instruction is reserved.

Clr `CF%CLR==:1B1` (Clear) When this bit is 1, the flags selected by ones in bits 2–6, 23–24, and 26–35 will be cleared to zero. If this bit and *Set* are both 0, this instruction has no effect. If this bit and *Set* are both 1, this instruction is reserved.

DMP `CF%DMP==:1B2` (Dump Request) This flag informs TDBoot that a memory dump to a file is wanted. TDBoot reads this flag after the program executes a HALT instruction.

DIA `CF%DIA==:1B3` (Diagnose Request) This flag informs TDBoot that diagnostics are wanted. TDBoot reads this flag after the program executes a HALT instruction. Diagnostics are performed subsequent to the dump, if requested.

BOO `CF%BOO==:1B4` (Boot Request) This flag informs TDBoot that the system should be rebooted, using the existing default boot parameters. TDBoot reads this flag after the program executes a HALT instruction. Boot is performed subsequent to diagnostics, if requested.

ATO `CF%ATO==:1B5` (Automatic) This flag is used by TDBoot to inform the timesharing monitor that the reboot occurred without manual intervention.

DBG `CF%DBG==:1B6` (Debugging) This flag is used by by TDBoot to inform the timesharing monitor that the operator has requested the monitor to boot for stand–alone operation.

ACF `CF%ACF==:1B18` (AC Fail) This read–only indicator is set when the system power control has detected an AC Low condition.

THW `CF%THW==:1B19` (Thermal Warning) This read–only indicator is set when the temperature in the system card cage is excessive. (Generally, there is no corrective action that the software can undertake. However, as Thermal Warning is often followed by a thermal shutdown, the software may treat this condition as a power failure.)

BTF `CF%BTF==:1B20` (Battery Fault) This read–only indicator is set when the system power control is unable to charge the battery. Generally, this means either that the battery is disconnected or that the battery has exceeded its useful life.

BTL `CF%BTL==:1B21` (Battery Low) This read–only indicator is set when the battery is unable to sustain system operation much longer.

NPW `CF%NPW==:1B22` (Need Power) This read–only indicator is set when any device in the system is asserting the Need DC Power backplane signal. (This processor can assert Need DC Power by setting the `CF%NDC` flag, below.)

KPA `CF%KPA==:1B23` (Keep–Alive Counter) This flag enables the microcode's "Keep–Alive" counter (see §3.8.3). When set, the microcode will decrement the Keep–Alive counter periodically; when clear, the Keep–Alive counter is disabled. (The Keep–Alive counter is normally enabled while the operating system is running; however, programs such as KDDT, EDDT, or TDBoot will disable the Keep–Alive mechanism while they are active.)

NDC `CF%NDC==:1B24` (Need DC Power) This flag controls the processor's contribution to a wire-ORed signal on the backplane. Any device on the backplane may assert this signal to signify that it has not yet completed its power–failure shutdown routine. The power control will attempt to maintain DC power by using the battery while any device asserts this signal.

SAL `CF%SAL==:1B25` (System Active Light) This function causes the processor to light the yellow System Active light, on system front panel, for 15 milliseconds. (There is no corresponding flag.)

RI `CF%RI==:1B26` (Ring Indicate) This read–only flag signifies the state of the Ring Indicate lead on the auxiliary console port's modem.

CD `CF%CD==:1B27` (Carrier Detect) This read–only flag signifies the state of the Carrier Detect signal on the auxiliary console port's modem.

DTR `CF%DTR==:1B28` (Data Terminal Ready) This flag controls the state of the Data Terminal Ready lead at the auxiliary console port's modem.

LED3 `CF%LD3==:1B29` This flag controls the state of the green LED labeled "3", visible through the module cover panel of the processor board.

LED2 `CF%LD2==:1B30` This flag controls the state of the green LED labeled "2", visible through the module cover panel of the processor board.

COK `CF%COK==:1B31` (Console OK) This flag controls the state of the green LED labeled "Port OK", visible through the module cover panel of the processor board, above the console port connector.

APE `CF%APE==:1B32` (Auxiliary Terminal Port Enable) When set, this flag permits the use of the auxiliary terminal port. When enabled, the auxiliary port is connected "in parallel" to the console terminal port: characters typed on either terminal are seen by the program as if they came from the console terminal; characters sent to the console terminal are also sent to the auxiliary port. While the auxiliary terminal port is enabled, the green LED labeled "Port OK", visible through the module cover panel of the processor board, above the auxiliary port connector, will be lit.

When the auxiliary terminal is disabled, input characters from the port are ignored and console port output is not copied to the auxiliary port.

LED1 `CF%LD1==:1B33` This flag controls the state of the green LED labeled "1", visible through the module cover panel of the processor board.

RTS `CF%RTS==:1B34` (Request to Send) This flag controls the state of the Request to Send signal at the auxiliary console port's modem.

LED0 `CF%LD0==:1B35` This flag controls the state of the green LED labeled "0", visible through the module cover panel of the processor board.

RCTRLF Read Control Flags (APR0 11,)

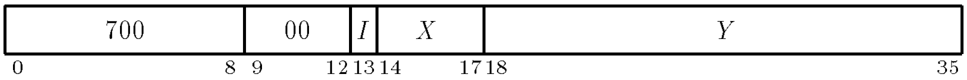
700	11	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

This instruction reads the condition of the control flags and stores the result in the word at E. The format is the same as that used in WCTRLF, except *Set*, *Clr*, and *SAL* all are read as zero.

3.9.3 Processor and System Identification

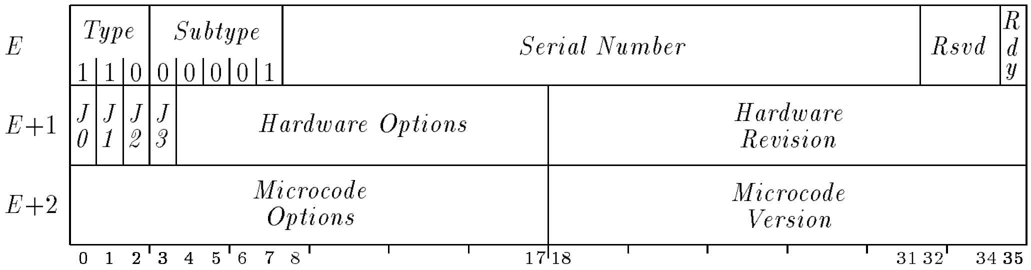
For initial setup, the executive must be cognizant of certain fundamental characteristics that can vary from one system to another. For this purpose, the following instructions are the means by which the program can identify unique processor characteristics.

APRID **Arithmetic Processor Identification (APR0 0,)**



Read the microcode version number, the processor serial number, and a listing of the fundamental characteristics of the system into locations E, $E+1$, and $E+2$ as shown:

Processor Identification Tripleword



These words are also the processor's response to a "Device_Status_Request" bus cycle directed to it at in–module addresses 0, 1, and 2.

In the locatation addressed by E, the fields have the following meaning:

Type	This field (`DS%TYP`) contains the value 6 (`AP%TCP`), signifying that this is a processor. (Remember, this word is also the processor's response to a "Device_Status_Request" sent to it at address 0.)
Subtype	This field (`DS%STY`) is reserved for major changes to the processor, including processors of architecture substantially different from the XKL-1.
Serial Number	(`AP%SNM==:77777777B31`) This field uniquely identifies the processor board. In contrast to the System Serial Number, this number is used primarily for tracking the history of this particular board.
Rdy	Ready (`AP%RDY`). This bit will always be read as 1 by **APRID**. (The processor is not ready while it is halted, but of course, **APRID** will never see this condition. However, the not ready condition can be reported to another device that directs a Device_Status_Request to the halted processor.)

In the location addressed by $E+1$, the fields have the following meaning:

J0 – J3	The status of four option jumpers on the CPU board, numbered J2–0, J2–1, J2–2, and J2–3, is reported in bits 0–3, respectively. An installed jumper is read as a "1" in the corresponding bit. The J2 jumpers are located on the XKL-1 board near the connector to the auxiliary console port. With the board in its normal orientation, J2–0 is at the top. If jumper J2–0 is installed, the microcode will disable the macro–console and not attempt to run macro code. (See also the .M command in Appendix E.) This jumper is

	installed only for diagnostic purposes. The significance of the other jumpers is reserved.
Hardware Options	At present, there are no defined hardware options.
Hardware Revision	The hardware revision number is reported in this field.

In the location addressed by $E + 2$, the fields have the following meaning:

Microcode Options	At present, the defined microcode options are Bit 0 Debugging microcode Bit 1 Exotic microcode: the microcode differs in some way from the standard version. All other microcode option bits are reserved.
Microcode Version	This is the microcode version number.

SYSID System Identification (APR0 2,)

700	02	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Read the system serial number and the backplane serial number into locations E and $E+1$ as shown:

Data Format for SYSID

0 23	24 35
System Identification Number	*Reserved*
Backplane Serial Number	*Reserved*

The system identification number is in bits 0–23 of the word stored at E (`SY%SID==:77777777B23`). The system identification number is unique to each TOAD-1 System. It is held in a socketed ROM on the backplane. The system identification number is shared by all processors attached to the backplane. The system identification number, rather than a processor serial number, can be used identify a particular system of multiprocessors.

In the event that a backplane is replaced, it is intended that the system identification number ROM stay with the system.

Bits 24–35 of the word stored at E (`SY%SOP==:7777`) is reserved for reporting system options.

The backplane serial number is in bits 0–23 of the word stored at $E+1$ (`SY%BSN==:77777777B23`). The backplane serial number, unique to the backplane, is held in ROM permanently affixed to the backplane; it is used for tracking the history of the backplane.

Bits 24–35 of the word stored at $E+1$ (`SY%BOP==:7777`) is reserved for reporting backplane options.

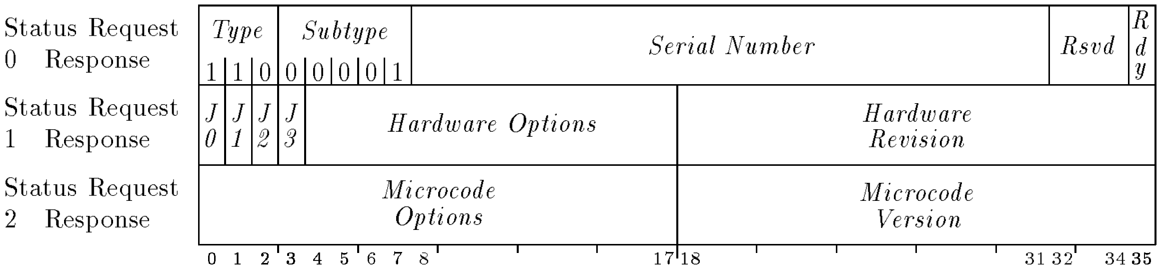
3.10 Response by the XKL-1 Processor as a Device

In multi–processor systems, a XKL-1 processor may be addressed as a device by another processor. The responses of the XKL-1 are described here.

3.10.1 Processor Response to Device_Status_Request

The processor responds to a Device_Status_Request backplane cycle directed to its in–module addresses 0, 1, and 2 by returning the first, second, and third words of the APRID data, respectively:

Processor Response to Device_Status_Request



The type code pattern 110 (`AP%TCP==:6`) in bits 0–2 (`DS%TYP`) of the Device_Status_Request 0 response word identifies this as a processor.

A 1 in the *Rdy* (Ready) flag (`AP%RDY==:1B35`) tells other devices on the bus that the processor is ready. (The processor is not ready while its microcode is being loaded; it is not ready while it is halted.)

At addresses 3–7, the processor responds by supplying the contents of MemA locations 323–327, respectively (`AM%SY0==:320`, etc.). These cells are used in multiprocessor systems to provide inter–processor synchronization before the memory and operating system are fully functional.

3.10.2 Processor Response to Device_Control

The processor's response to Device_Control bus cycles is reserved, pending the design of multiprocessor systems.

3.11 XMG-1 Memory System

There are two memory systems available for the TOAD-1 System: the XMG-1 and the XMG-3. Either or both types might be present.

3.11.1 The XMG-1 Memory Board

The XMG-1 memory board is available in either 16– or 32–million word configurations.

The XMG-1 responds to Line_Read_Request, Line_Write, Word_Read_Request, and Word_Write bus cycles; these cycles are used for the usual storage transfers. The XMG-1 also responds to Status_Read_Request and Device_Control bus cycles (as generated by `PMOVE` and `PMOVEM`, respectively, when the *D* bit is 1); these cycles are used to obtain status information and to effect some control operations.

3.11.1.1 XMG-1 Device_Status_Request Functions

The memory responds to Device_Status_Request at even–numbered in–module addresses 0 through (octal) 76, and at in–module address 1.

Even–numbered in–module addresses 0 through 76 respond as follows:

0	1	0	*Subtype* 0				1	0	*ID ROM Data*	*Brdy*	*Ardy*	*BTO*	*PED*	0	*32M*	*DC4*	*OfL*	*OnL*	*WBP*	*Rdy*
			0	0	0	0	1													
0	1	2	3	4	5	6	7	8 15	16 23	24	25	26	27	28 29	30	31	32	33	34	35

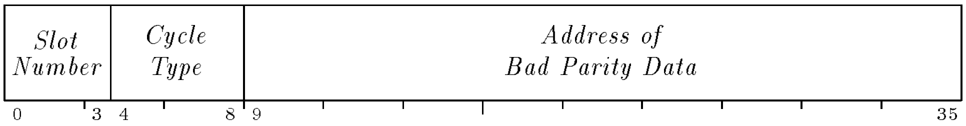
The memory type code pattern 010 (`MD%TCP==:2`) in the device type field (bits 0–2, `DS%TYP`) together with the subtype code 00001 (`MD%MG1==:1`) in the device subtype field (bits 3–7, `DS%STY`) identify the board as the XMG-1 memory type.

The other status bits are described below.

ID ROM Data	One 8–bit byte of data from the 32–byte on–board ID ROM is returned in this field (`MD%ROM==377B23`). ID ROM addresses 0 through (octal) 37 are selected by even–numbered in–module addresses 0 through (octal) 76, respectively. ID ROM addresses 0, 1, and 2 (accessed by in–module addresses 0, 2, and 4, respectively) hold the three bytes of the board's serial number, where address 0 reports the least–significant byte. ID ROM address 3 through 37 are reserved.
Brdy	Memory bank B is ready. (`MD%BRY==:1B24`)
Ardy	Memory bank A is ready. (`MD%ARY==:1B25`)
BTO	Busy Timeout: the busy counter has timed out while the memory was attempting a data return. (`MD%BTO==:1B26`)

PED	Parity Error Detected. (`MD%PED==:1B27`)
32M	Memory capacity is 32 M words; if this bit is zero, the memory capacity is 16 M words. (`MD%32==:1B30`)
DC4	Device Control Bit 4. Unassigned.
OfL	Offline. This flag is set to 1 by the system configuration software after a processor has tested this memory and found that the memory failed. (`MD%OFL==:1B32`)
OnL	Online. This flag is set to 1 by the system configuration software after it has successfully run test patterns in this memory. When the flag is set, a green LED (visible through the module cover panel) will be on to signify that the board is online. (`MD%ONL==:1B33`)
WBP	Write Bad Parity. When set to 1, the memory will compute bad parity bits for the data words that it stores. While set to 1, words read from memory that have good parity cause parity errors; words read from memory that have bad parity are accepted without complaint. This flag is intended for memory diagnostics only. (`MD%WBP==:1B34`)
Rdy	Ready. When set to 1, this flag signifies that Power–on reset has been completed and the memory is ready. (`MD%RDY==:1B35`)

The XMG-1 reports parity error information in the Error Register, at in–module address 1 (`.MDERR==:1`). Performing a status read from address 1 clears the *PED* (parity error detected) and the *BTO* (bus timeout) flags (which are reported at address 0).



Slot Number	(`MD%ESN==:17B3`) Backplane slot number of the device to which bad parity data was sent.
Cycle Type	(`MD%ECT==:37B8`) Type of backplane bus cycle in which the bad parity data was delivered.
Address of Bad Parity Data	`MD%EMA==:777777777` This field reports the in–module address of the reference that found a word that has bad parity.

The first parity error that is discovered causes the memory to latch the information reported here and to set the *PED* flag (reported in status word 0). Subsequent parity errors are not latched until *PED* has been cleared (which is done by reading the error register). While *PED* is set, parity errors are still reported to recipients of data (as explained in §3.1).

The least significant bit of the `MD%EMA` field is misleading: the memory actually stores and retrieves 72–bit LongWords. If a parity error exists in a LongWord, reading from either of the two addresses that refer to this LongWord will set the error latch. So, the least significant address bit reflects the address of the first reference to the LongWord, it does not indicate which of the two words contains the error. The bus parity error indication occurs only when transferring the actual word that contains the parity error.

3.11.1.2 XMG-1 Device_Control Functions

The XMG-1 responds to the Device_Control function at address 0 (`.MDSTS==:0`).

The data sent to the memory should be in the following form:

0	*En4*	*EFL*	*EOL*	*EBP*	*EnR*	0	*DC4*	*OfL*	*OnL*	*WBP*	*RR*
0–12	13	14	15	16	17	18–30	31	32	33	34	35

Bits 13–17 are enables for bits 31–35, respectively. The significance of bits 31–35 is as follows:

DC4 Device Control flag 4 will be set if both *DC4* (`MD%DC4==:1B31`) and *En4* (Enable 4, `MD%EN4==:1B13`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *DC4* is zero when *En4* is 1 in a Device_Control cycle addressed to location 0.

DC4 is setable, clearable, and readable. It has no assigned function in the device or in software.

OfL Offline will be set if both *OfL* (`MD%OFL`) and *EFL* (Enable Offline, `MD%EFL==:1B14`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *OfL* is zero when *EFL* is 1 in a Device_Control cycle addressed to location 0.

The offline bit is setable, clearable, and readable. Although it has no effect on the operation of the memory itself, it is used by the TDBoot program to signal that the memory has failed one of its tests.

OnL Online. This flag will be set if both *OnL* (`MD%ONL`) and *EOL* (Enable Online, `MD%EOL==:1B15`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *OnL* is zero when *EOL* is 1 in a Device_Control cycle addressed to location 0. When the flag is set, a green LED (visible through the module cover panel) will be on to signify that the board is online.

The TDBoot program sets this flag in the memory after it has successfully run test patterns. Other software may clear the flag if the memory is found to be failing.

WBP The Write Bad Parity flag will be set to one if both *WBP* (`MD%WBP`) and *EBP* (Enable Bad Parity, `MD%EBP==:1B16`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *WBP* is zero and *EBP* is 1 in a Device_Control cycle addressed to location 0.

When this flag is set, the parity bit is inverted on write operations, and the meaning of the parity bit is inverted on read operations. Thus, when the Write Bad Parity flag is set in the memory control, all data written will be stored with bad parity. When good parity data is read, those data items will be reported as parity errors, but data with bad parity will not be reported. This control flag is for memory diagnostic purposes only.

RR When the Reset Request flag (`MD%RR==:1B35`) and *EnR* (Enable Reset, `MD%ENR==:1B17`) are both set to 1 in a Device_Control cycle addressed to location 0, the memory control will reset itself to its power–on condition. While the memory is resetting, the *Rdy* flag (`MD%RDY`, as reported by a Device_Status_Request to location 0) will be 0 and the memory will not respond normally; when the memory has finished its reset process, *Rdy* will be 1.

3.11.1.3 XMG-1 Response to Memory Cycles

The XMG-1 memory system responds as a memory to bus cycles of the types Word_Read_Request, Word_Write, Line_Read_Request, and Line_Write.

The word–mode cycles are generated by the central processor's `PMOVE` and `PMOVEM` instructions and by it references to uncached pages. Also, other devices (e.g., the network interface) may generate these cycle types.

The line–mode cycles are generated by the processor (or other devices) to read or write a cache line. In the case of reading a cache line, the requesting device will want one particular word first, so the low–order address bits 33–34 of the desired word are presented to the memory along with the rest of the address in a Line_Read_Request, and the memory's response will send the requested double word first and then cycle through the other three double words.

3.11.1.4 XMG-1 Initialization

The XMG-1 contains a very limited state–machine; it does not do much initialization.

The contents of the memory at power–up are undefined; it is completely plausible that memory contains bad parity data. On power–up, TDBoot usually tests and clears memory.

3.11.2 The XMG-3 Memory Board

The XMG-3 memory board is available in varying capacities from 64 million 36–bit words to 512 million words, dependending on how many 64 Mw memory banks are installed on the board.

The XMG-3 responds to Line_Read_Request, Line_Write, Word_Read_Request, Word_Write, LongWord_Read_Request, and LongWord_Write bus cycles; these cycles are used for the usual storage transfers. The XMG-3 also responds to Status_Read_Request and Device_Control bus cycles (as generated by `PMOVE` and `PMOVEM`, respectively, when the D bit is 1); these cycles are used to obtain status information and to effect control operations.

3.11.2.1 XMG-3 Device_Status_Request Functions

The memory responds to Device_Status_Request at even–numbered in–module addresses 0 through (octal) 76, and at in–module address 1.

Even–numbered in–module addresses 0 through 76 respond as follows:

0	1	0	*Subtype* 0 0 0 1 1	1	1	1	1	1	1	1	*BnkP*	*ID ROM Data*	*FBsy*	*DRdy*	*BTO*	*PED*	*DC7*	*OfL*	*OnL*	*DC4*	*WBP2*	*WBP1*	*WBP0*	*Rdy*
0	1	2	3 4 5 6 7	8	9	10	11	12	13	14	15	16 23	24	25	26	27	28	29	30	31	32	33	34	35

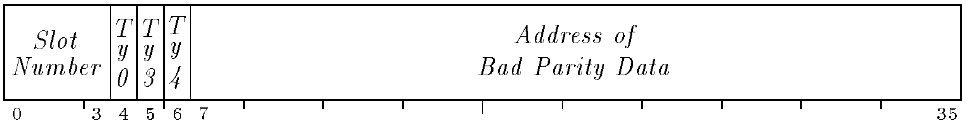
The memory type code pattern 010 (`MD%TCP`) in the device type field (bits 0–2, `DS%TYP`) together with the subtype code 00011 (`MD%MG3==:3`) in the device subtype field (bits 3–7, `DS%STY`) identify the board as the XMG-3 memory type.

The other status bits are described below.

Bits 8–14	These bits report a pattern unique to the DRAM part that is installed in the bank (at position 0–1). The expected value is all 1s.
BnkP	Bank Present (`M3%BKP==:1B15`): if this bit is zero it signifies that the memory bank is present. Bank numbers 0 through 7 are selected by even–numbered in–module addresses 0 through (octal) 16, respectively. If this bit is one, the bank is absent. (Address 20 repeats the bank data for address 0, etc.) Although it is possible to populate the board with DRAMs arranged in non–consecutive banks, software constraints mandate the presence of bank 0 and do not allow the utilization of non–consecutive banks. A bank requires two DRAMs, designated 0–1 and 2–3 (corresponding to the least significant two address bits of the words that they store). *BnkP* actually reports the status of the 0–1 DRAM. If that DRAM is present, the memory reports the entire bank as present. (If the DRAM 2–3 for that bank is absent, words at addresses ending in binary 10 and 11 will not work.)
ID ROM Data	One 8–bit byte of data from the 32–byte on–board ID ROM is returned in this field (`MD%ROM`). ID ROM addresses 0 through (octal) 37 are selected by even–numbered in–module addresses 0 through (octal) 76, respectively. ID ROM addresses 0, 1, and 2 (accessed by in–module addresses 0, 2, and 4, respectively) hold the three bytes of the board's serial number, where address 0 reports the least–significant byte. ID ROM address 3 through 37 are reserved.
FBsy	Flash card is busy (or absent). (`M3%FBY==:1B24`)
DRdy	DRAM Ready after power up. (`M3%DRY==:1B25`)
BTO	Busy Timeout: the busy counter has timed out while the memory was attempting a data return. (`MD%BTO`)
PED	Parity Error Detected. Set if the memory has detected a parity error. (`MD%PED`)
DC7	Device Control Bit 7. Unassigned.
OfL	Offline. This flag is set to 1 by the system configuration software after a processor has tested this memory and found that the memory failed. (`M3%OFL==:1B29`)
OnL	Online. This flag is set to 1 by the system configuration software after it has successfully

	run test patterns in this memory. When the flag is set, a green LED (visible through the module cover panel) will be on to signify that the board is online. (`M3%ONL==:1B30`)
DC4	Device Control Bit 4. Unassigned.
WBPn	Write Bad Parity (bytes 0, 1, and 2). For the purpose of parity computation, each word in memory is divided into three 12–bit bytes, each byte having its own parity bit. Each of these flags controls the sense of the parity bit for one byte of each word in memory. When one of these flags is set to 1, the memory will compute and store a “bad” parity bit corresponding to the data byte that it stores. While one of these flags is set to 1, a byte read from memory that has “good” parity will cause a parity error to be reported, and a byte read from memory that has “bad” parity is accepted without complaint. These flags are intended for memory diagnostics only. (`M3%WB0==:1B34`; `M3%WB1==:1B33`; `M3%WB2==:1B32`) *WBP0* controls the parity for bits 0–11 of a memory word. *WBP1* is for bits 12–23; *WBP2* is for bits 24–35.
Rdy	Ready. When set to 1, this flag signifies that Power–on reset has been completed and the memory is ready. If 0, reset is stuck. (`MD%RDY`)

The XMG-3 reports parity error information at in–module address 1 (`.MDERR`). Performing a status read from address 1 clears the *PED* (parity error detected) and the *BTO* (bus timeout) flags (which are reported at address 0).



Slot Number	(`MD%ESN`) Backplane slot number of the device to which bad parity data was sent.
Cycle Type	(`M3%ECT==:7B6`) The type of backplane bus cycle in which the bad parity data was delivered. Bit 4 records type bit 0; bits 5 and 6 record type bits 3 and 4, respectively.
Address of Bad Parity Data	`M3%EMA==:3777777777` This field reports the in–module address of the reference that found a word that has bad parity.

The first parity error that is discovered causes the memory to latch the information reported here and to set the *PED* flag (reported in status word 0). Subsequent parity errors are not latched until *PED* has been cleared (which is done by reading the error register); while *PED* is set, parity errors are still reported to recipients of data (as explained in §3.1).

The least significant bit of the `M3%EMA` field is misleading: the memory actually stores and retrieves 72–bit LongWords. If a parity error exists in a LongWord, reading from either of the two addresses that refer to this LongWord will set the error latch. So, the least significant address bit reflects the address of the first reference to the LongWord, it does not indicate which of the two words contains the error. The bus parity error indication occurs only when transferring the actual word that contains the parity error.

3.11.2.2 XMG-3 Device_Control Functions

The XMG-3 responds to the Device_Control function at address 0 (`.MDSTS==:0`).

The data sent to the memory should be in the following form:

0–9	10	11	12	13	14	15	16	17	18–27	28	29	30	31	32	33	34	35
0	*En7*	*EFL*	*EOL*	*En4*	*EBP2*	*EBP1*	*EBP0*	*EnR*	0	*DC7*	*OfL*	*OnL*	*DC4*	*WBP2*	*WBP1*	*WBP0*	*RR*

Bits 10–17 are enables for bits 28–35, respectively. The significance of bits 28–35 is as follows:

DC7 Device Control flag 7 will be set if both *DC7* (`M3%DC7==:1B28`) and *En7* (Enable 7, `M3%EN7==:1B10`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *DC7* is zero when *En7* is 1 in a Device_Control cycle addressed to location 0.

DC7 is setable, clearable, and readable. It has no assigned function in the device or in software.

OfL Offline will be set if both *OfL* (`M3%OFL`) and *EFL* (Enable Offline, `M3%EFL==:1B11`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *OfL* is zero when *EFL* is 1 in a Device_Control cycle addressed to location 0.

The offline bit has no function in the memory itself. Instead, the TDBoot program uses this flag to signal that the memory has failed one of its tests.

OnL Online. This flag will be set if both *OnL* (`M3%ONL`) and *EOL* (Enable Online, `M3%EOL==:1B12`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *OnL* is zero when *EOL* is 1 in a Device_Control cycle addressed to location 0. When the flag is set, a green LED (visible through the module cover panel) will be on to signify that the board is online.

The TDBoot program sets this flag in the memory after it has successfully run test patterns. Other software may clear the flag if the memory is found to be failing.

DC4 Device Control flag 4 will be set if both *DC4* (`M3%DC4==:1B31`) and *En4* (Enable 4, `M3%EN4==:1B13`) are set to 1 in a Device_Control cycle addressed to location 0. The flag will be cleared to zero if *DC7* is zero when *En7* is 1 in a Device_Control cycle addressed to location 0.

DC4 is setable, clearable, and readable. It has no assigned function in the device or in software.

WBPn A Write Bad Parity flag (`M3%WB`*n*, n = 0, 1, or 2) will be set to one if both *WBPn* and *EBPn* (Enable Bad Parity *n*, `M3%EB2==:1B14`; `M3%EB1==:1B15`; `M3%EB0==:1B16`) are set to 1 in a Device_Control cycle addressed to location 0. A flag will be cleared to zero if *WBPn* is zero and *EBPn* is 1 in a Device_Control cycle addressed to location 0.

When one of these flag is set to 1, the parity bit for the corresponding byte is inverted on write operations, and the meaning of the parity bit is inverted on read operations. Thus, when a Write Bad Parity flag is set in the memory control, all data bytes (corresponding to that flag) that are written will be stored with bad parity. When good parity data is read, those data items will be reported as parity errors, however, data with bad parity will not be reported. This control flag is for memory diagnostic purposes only.

RR When the Reset Request flag (`MD%RR`) and *EnR* (Enable Reset, `MD%ENR`) are both set to 1 in a Device_Control cycle addressed to location 0, the memory control will reset itself to its power–on condition. While the memory is resetting, the *Rdy* flag (bit 35, as reported by Device_Status_Request to location 0) will be 0 and the memory will not respond normally; when the memory has finished its reset process, *Rdy* will be 1.

3.11.2.3 XMG-3 Response to Memory Cycles

The XMG-1 memory system responds as a memory to bus cycles of the types Word_Read_Request, Word_Write, LongWord_Read_Request, LongWord_Write, Line_Read_Request, and Line_Write.

The word–mode cycles are generated by the central processor's `PMOVE` and `PMOVEM` instructions and by its references to uncached pages. Also, other devices (e.g., the network interface) may generate these cycle types.

The longword–mode cycles are generated by devices other than the central processor. In the case of reading a longword, the requesting device will provide an even–numbered in–module address and a bus cycle type LongWord_Read_Request. The memory will supply the 72–bit data item in a bus cycle of type DataR2 that will be directed to the requestor. In writing a longword, the requesting device will provide an even in–module address in a cycle of type LongWord_Write; it will then supply the 72–bit data item in a cycle of type DataW2.

The line–mode cycles are generated by the processor (and other devices) to read or write a cache line. In the case of reading a cache line, the processor will want one particular word first, so the low–order address bits 33–34 of the desired word are presented to the memory along with the rest of the address in a Line_Read_Request, and the memory's response will send the requested double word first and then cycle through the other three double words.

3.11.2.4 XMG-3 Initialization

The XMG-3 contains a very limited state–machine; it does not do much initialization.

The contents of the memory at power–up are undefined; it is completely plausible that memory contains bad parity data. On power–up, TDBoot usually tests and clears memory.

3.11.2.5 XMG-3 Flash Card

To be added.

3.12 XRH-1 Mass–Storage Interface Processor

The XRH-1 Mass–Storage Interface Processor provides a highly efficient, buffered connection between the TOAD-1 System backplane bus and four independent, wide, fast SCSI–2 buses.

The XRH-1 responds to backplane bus cycles of the types Device_Status_Request and Device_Control. It generates bus cycles of types Device_Status_Return, Interrupt, Line_Read_Request, and Line_Write_Request.[33] The description that follows is organized as an explanation of the I/O registers (those that respond to the Device_Control and Device_Status_Request bus cycles) and a description of the protocol of communication between the CPU (operating system) and the XRH-1.

The XRH-1 Mass–Storage Interface Processor contains an 18 MB cache of recently read and written data. (The cache is equivalent to 4 Mw or 16 sections of main memory.) The XRH-1 and the SCSI devices internal to the TOAD-1 System chassis operate from a power source that contains a battery to operate the devices through short power failures and sufficient to unload the cache to the disks in the event of a longer power outage. The cache is used to supply copies of data recently read or written, thereby increasing the effective disk bandwidth.

The XRH-1 includes a microprocessor that controls its general activity, including the management of the cache and the scheduling of individual transfers.

The usual form of communication between the operating system and the XRH-1 involves two in–memory data structures. The first of these is an eight–word communications region, through which lists of messages are passed; the XRH-1 utilizes just one such region at a time. The second data structure, many instances of which may exist at any time, is called a Mass–Storage Control Block (MSCB). The MSCBs are found in main memory, either individually or in lists. As explained below, some of the locations in the communications region and some of the I/O registers contain the address of an MSCB (which may be the head of a list of MSCBs) or the address of the communications region; such addresses are in the form of bus address words (BAWs, see §3.1.4).

3.12.1 XRH-1 Mass–Storage Interface Processor I/O Registers

I/O registers are implemented by the action of the microprocessor's microcode, with a small assist from hardware. Due to this choice of implementation, access to the I/O registers may be very slow; an access attempt may even result in a bus timeout or bus busy, even when the XRH-1 is operating normally. Thus, software to control the XRH-1 must minimize access to these registers; fortunately, the design of the XRH-1 provides for efficient communication to it via memory cells in the communications region.

Following the CPU's access to any of the XRH-1 I/O registers, the XRH-1's backplane bus interface is designed to respond "busy" to any subsequent access. The bus interface remains busy until the microprocessor has accepted the first backplane bus event and enabled the bus interface to be receptive again. Moreover, a Device Status request, which requires a response from the microcode, may result in a bus timeout if the microcode fails to respond swiftly. Therefore, the use of the XRH-1 I/O registers is held to a minimum during system operations.

---

[33] The XRH-1 hardware also can generate Word_Read_Request and Word_Write; however, these capabilities are not be used by the XRH-1's operating microcode.

3.12.1.1 Device Status

The XRH-1 does not have dedicated hardware that responds to Device_Status_Requests; the responses are made by the action of the firmware. When the firmware is busy (for example, while resetting after power is applied), responses may be so slow as to create a bus timeout or a persistent "busy" condition. Therefore, the program must be aware of the possibility of such exceptions and take appropriate action to recover if any exceptions occur.

A Device_Status_Request directed to the XRH-1 at address 0 will elicit the main status of the device, as described below. However, the preferred means by which to obtain this status word is from word 0 of the communications region. When there is no communications region or when the status of communications is not known, this status word can be accessed via the XRH-1's response to a Device_Control cycle to address 3.

Mass–Storage Interface — Response to Device_Status_Request to Address 0

Type	*Subtype*	*CRR*	*BCR*	*CRE*	*IBT*	*FUL*	*CTO*	*CPE*	*MTO*	*MPE*	*BB0*	*BB1*	*BB2*	*BB3*	*DPE*		*INV*	*Err*	*Msg*	*Atn*	*Rdy*
0 0 0	0 0 0 0 1																				
0 1 2	3 4 5 6 7	8	9	10	11	12	13	14	15	16	17	18	19	20	21		31	32	33	34	35

This response word is decoded as follows:

Type	Bits 0–2 (`DS%TYP`) will contain the type–code pattern 000 (`MX%TCP==:0`) to identify this backplane slot as containing a mass–storage interface subsystem.
Subtype	Bits 3–7 (`DS%STY`) identify the subtype (i.e., particular model) of mass–storage controller. The initial model XRH-1 is of subtype 1.
CRR	Communications region rejected (`MX%CRR==:1B8`). The XRH-1 has been told to use a new communications region at a time when a communications region was already established for which there were MSCBs still outstanding. (This flag bit is never actually seen when accessing the status via a Device Status request to address 0; instead, it is written in the in–memory status word in the rejected communications region. There is no interrupt.)
BCR	Bad communications region (`MX%BCR==:1B9`). The XRH-1 cannot use the communications region that was assigned. The CPU may have provided a bad bus address word. In any event, it is useless to look in the communications region for more information, because there is no such region. (This condition does not cause an interrupt, because no priority level assignment can be made without a valid communications region. Of course, this condition will not be written to the status word in the communications region.)
CRE	Communications region established (`MX%CRE==:1B10`). The XRH-1 has accepted and is using the communications region assigned to it.
IBT	Invalid bus transaction detected (`MX%IBT==:1B11`).
FUL	Buffer space is full. (`MX%FUL==:1B12`). The XRH-1 has no room in its buffer space for additional MSCBs. Either the operating system has overrun the XRH-1 with too many outstanding requests, or there is an error internal to the XRH-1 or its microcode.
CTO	Memory timeout in access to communications region (`MX%CTO==:1B13`). This error flag and

	the next three that follow cannot reliably be transmitted via the normal communications mechanism, so they are reported in this status. When any of these four flags is set, a Device_Status_Request to address 2 will elicit the bus–address word corresponding to the most recent error of this kind. The Device_Status_Request to address 0 that reports these bits will clear them.
CPE	Memory parity error in access to communications region (`MX%CPE==:1B14`).
MTO	Memory timeout in access to an MSCB (`MX%MTO==:1B15`).
MPE	Memory parity error in access to an MSCB (`MX%MPE==:1B16`).
BB0	Bus Bad 0 (`MX%BB0==:1B17`). The XRH-1 has discovered a problem in SCSI bus 0; the bus is not usable.
BB1	Bus Bad 1 (`MX%BB1==:1B18`). The XRH-1 has discovered a problem in SCSI bus 1; the bus is not usable.
BB2	Bus Bad 2 (`MX%BB2==:1B19`). The XRH-1 has discovered a problem in SCSI bus 2; the bus is not usable.
BB3	Bus Bad 3 (`MX%BB3==:1B20`). The XRH-1 has discovered a problem in SCSI bus 3; the bus is not usable.
DPE	DRAM parity error detected (`MX%DPE==:1B21`). During a transfer involving the XRH-1's internal DRAM, a parity error has been found.
INV	Status Stale (`MX%INV==:1B31`). The status word in the communications region needs to be updated.
Err	Error (`MX%ERS==:1B32`). The XRH-1 has an error condition to report. If the XRH-1 is enabled to interrupt, this condition causes an interrupt. Many error conditions are treated in the same way as asynchronous status is treated: if an MSCB is available in which to report error conditions, the MSCB will be used instead of causing an interrupt. If an interrupt occurs, the CPU must respond by providing an MSCB in which to report the error.
Msg	Message (`MX%MSG==:1B33`). Unused.
Atn	Attention (`MX%ATN==:1B34`). The XRH-1 has asynchronous (or unsolicited) status to report. If the XRH-1 is enabled to interrupt, this condition causes an interrupt. (Typically, during system operation, asynchronous status is reported via MSCBs reserved for that purpose.)
Rdy	Ready (`MX%RDY==:1B35`). The XRH-1 is at (or near) normal operating conditions. It is ready to receive a communications region assignment if none has yet been made. This flag will be off during such circumstances as the XRH-1 resetting itself and during such times when the XRH-1 is busy for an extended period processing existing requests.[34]

A Device_Status_Request directed to the XRH-1 at address 1 will elicit the bus–address word of the assigned communications region, or zero if no assignment has been made.

---

[34] As presently implemented, the XRH-1 may provide no response (i.e., bus busy or bus timeout) while it is performing a hard reset.

Table 3.5: XRH-1 Status Read Request Addresses

Address bits 32–35 (`MX%REG`)	*Function*
`.MXSTS==:0`	Return XRH-1 Status Word.
`.MXCOM==:1`	Return XRH-1 Communications Region Address.
`.MXERA==:2`	Return the latest bus error address.
`.MXSRM==:3`	Return SRAM data from the SRAM address specified in address bits 16–31 (`MX%SRA==:177777B31`). The SRAM data is returned in bits 20–35 (`MX%SRD==:177777`).
`.MXDRM==:4`	Return DRAM data from the DRAM address specified in address bits 10–31 (`MX%DRA==:17777777B31`). Although the DRAM is 72 bits wide, this command returns only 36 bits. Address bit 31 selects which half of the DRAM double word is returned.
`.MXUCV==:5`	Return the XRH-1 microcode version number. Bits 20–27 report the major version number and bits 28–35 report the minor version number.
`.MXALU==:6`	Return data from the ALU register addressed by bits 26–31 (`MX%ARA==:77B31`). The ALU registers are 16 bits wide; the data is returned in bits 20–35 (`MX%ALD==:177777`).
`.MXDSN==:7`	Return the XRH-1 serial number. The serial number is 24 bits; it is reported in bits 12–35 (`MX%SND==:77777777`).
`10—17`	Reserved

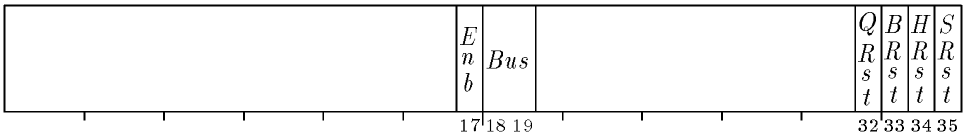
A Device_Status_Request directed to the XRH-1 at address 2 will elicit the bus address word corresponding to the most recent system bus error (timeout or parity error) as specified in the XRH-1 status register 0.

Device_Status_Requests to other addresses are defined in Table 3.5. Address bits 32–35 select which group of registers to access (`MX%REG==:17`); other address bits select a particular register from the group.

3.12.1.2 Device Control

A Device_Control data word (i.e., the data word associated with a Device_Control backplane bus cycle) directed to the XRH-1 at address 0, the control word that is sent can effect the following functions:

XRH-1 Mass–Storage Interface Processor — Device_Control to Address 0



Enb	Enable (`MX%ENB==:1B17`). When set, this bit enables the reset functions described below.
Bus	Bus selection field (`MX%BN==:3B19`). This field is used to select a particular SCSI bus for the *BRst* function.
QRst	Quietus Reset (`MX%QR==:1B32`). When set while `MX%ENB` is also set, this flag causes the XRH-1 to finish all work currently assigned to it and then to go to a state of inactivity. Upon receipt of this command, the XRH-1 will cease to use any previously assigned communications region, and it will clear its priority interrupt assignment. If the XRH-1 currently has pending MSCBs in its memory, the work described by those MSCBs will be completed (if possible); the completed MSCBs are discarded (i.e., they are not returned to the CPU). If the contents of the XRH-1's cache memory are valid and not yet written to disk, that information will be copied to the disk. After completing these tasks, the XRH-1 reports itself ready when inquiries are made to Device Status address 0. It is an error for this bit to be set when any of *SRst*, *BRst*, or *HRst* is also set. Subsequent to requesting a Quietus Reset, the program should monitor the status of the ready flag `MX%RDY` and not send commands to the XRH-1 until it reports itself ready. Because the communications region assignment is invalidated by this reset, the preferred way to access the XRH-1 status word is via the XRH-1's response to a Device Control request to address 3.
BRst	SCSI Bus Reset (`MX%BR==:1B33`). When set while `MX%ENB` is also set, this flag causes the XRH-1 to reset the SCSI bus identified in the *Bus* field. Bus reset is appropriate after a SCSI error has occurred on a particular bus. It is an error for this bit to be set when any of *QRst*, *HRst*, or *SRst* is also set. See also "SCSI Bus Reset" on page 311.
HRst	Hard Reset (`MX%HR==:1B34`). When set while `MX%ENB` is also set, this flag causes the XRH-1 to reset itself to its power–on condition.[35] The contents of the XRH-1 buffer memory are ignored. Pending commands are discarded. The priority interrupt assignment, communications region address, and SCSI bus identification assignments are discarded. The XRH-1 will reload its microcode, run diagnostics, and reset the SCSI buses. It is an error for this bit to be set when any of *QRst*, *BRst*, or *SRst* is also set.
SRst	Soft Reset (`MX%SR==:1B35`). When set while `MX%ENB` is also set, this flag causes the XRH-1 to stop its current activities, reset the DMAs and the SCSI buses, and reinitialize its data structures. Work in progress, cache contents, and any previous communications region assignment are lost. When the reinitialization is complete, the XRH-1 will report itself ready in its status word. It is an error for this bit to be set when any of *QRst*, *BRst*, or *HRst* is also set.

---

[35] Note, however, as presently implemented, when the XRH-1 is "hung" (e.g., in a microcode loop in which it does not pay attention to the backplane bus), this command may be ignored.

A Device_Control data word directed to the XRH-1 at address 1 (`.MXCRA`) identifies (as a bus–address word) the location of the communications region. Changing the communications region address is an error unless communications between the CPU and the XRH-1 have been brought to quiescence.

A Device_Control data word directed to the XRH-1 at address 2 (`.MXTO`) identifies (as a bus address word) the location of the next “To XRH-1” MSCB. That MSCB may be the head of a list of “To XRH-1” MSCBs. From the point of view of the CPU, this location is write–only. It is an error to write to this cell unless a communications region has been established.

A Device_Control data word directed to the XRH-1 at address 3 (`.MXSTR==:3`) contains a bus–address word. The XRH-1 will respond by storing its current status at the location specified by the BAW. Although it is preferred that the status be read from the communications region, there are some circumstances (for instance, when there is no communications region assigned) when this is the best way to read the status. (In contrast to a Device_Status_Request cycle to address 0, this method cannot result in a bus timeout.)

A Device_Control data word directed to the XRH-1 at address 4 (`.MXCRR==:4`) contains a bus–address word. The XRH-1 will respond by storing its current communications region assignment at the location specified by the BAW. When the program does not know what communications region is assigned (for example, when TDBOOT gains control after a program terminates), this is the best way to obtain the location of the communications region. (In contrast to a Device_Status_Request cycle to address 1, this method cannot result in a bus timeout.)

Device_Control cycles directed to other addresses are reserved.

3.12.2 Communication Between the CPU and the XRH-1

Communication between the CPU (operating system) and the XRH-1 involves a communications region and instances of a data structure called a Mass–Storage Control Block (MSCB). The communications region and every MSCB is contained within the main memory space of the system.[36] An MSCB must occupy precisely one memory (cache) line; thus, the address of an MSCB must be a multiple of 8 (octal 10) and the length of the MSCB is 8. This restriction is due to the XRH-1's preference for using the more efficient Line_Read_Request and Line_Write backplane bus cycles to reference memory. The communications region is also eight words long and aligned on an eight–word boundary. The memory pages containing the communications region and the MSCBs should not be cached; otherwise, the operating system would have to flush the cache to validate the in–memory versions of these data structures.

3.12.2.1 Communications Region

The communications region is an eight–word (`.MXCRL==:10`) area of main memory identified by the bus–address word that specifies the first word of the region. The communications region must be allocated at an address that is a multiple of eight. The format of the communications region is shown in Figure 3.6.

The communications region is allocated by the CPU. The CPU informs the XRH-1 of the loca-

---

[36] Copies of these data structures exist also in the XRH-1; however, this explanation is directed towards explaining the action of the XRH-1 from the point of view of the CPU or Monitor.

Figure 3.6: XRH-1 Communications Region Format

	0 3	4 11	12 19	20 27	28 35	
0	*XRH-1 Status*					`.MXCRS`
1	*XRH-1 Communications Region (a Bus Address Word)*					`.MXCRA`
2	*To XRH-1 (a Bus–Address Word)*					`.MXTO`
3	*From XRH-1 (a Bus–Address Word)*					`.MXFRM`
4	*Priority–Interrupt Level Assignment*					`.MXPIA`
5	0	*Bus 0 ID*	*Bus 1 ID*	*Bus 2 ID*	*Bus 3 ID*	`.MXBID`
6						Reserved
7						Reserved

tion of the communications region by sending the bus–address word as data in a Device_Control cycle directed to the XRH-1 at address 1. If the XRH-1 does not already have a communications region assigned, the XRH-1 accepts this assignment and acknowledges it by writing the XRH-1 status (including Communications Region Established) in word 0 of the communications region. In normal operations, this status word will thereafter be the same as the one reported by a Device_Status_Request to address 0; abnormal situations are described in the next paragraph. Upon successful assignment of a communications region, the XRH-1 will read the new priority–interrupt level assignment and the SCSI bus addresses from words 4 and 5 of the new region. These are read just once: subsequent changes to these memory locations are ignored by the XRH-1.

If the XRH-1's status write is unsuccessful or if the communications region address is not a multiple of 8, the Bad Communications Region bit will be set in the status word available via Device Status address 0. If a previous communications region had been assigned, and if there are MSCBs outstanding for that region, then the XRH-1 rejects the new communications region assignment: it writes the address of the established region in word 1 of the new, rejected communications region, and it sets Communications Region Rejected (and all other approriate status bits, including bits 0–7) in word 0 of the rejected region.

Note: The correct way to change communications regions is to bring all activity between the CPU and the XRH-1 to quiescence by means of a "Release Communications Region" command in an MSCB sent to the XRH-1. A CPU insistent on changing communications regions immediately can do so by sending the XRH-1 a Soft Reset via a Device Control function to address 0.

The contents of the communications region are as described here:

0 XRH-1 Status (`.MXCRS`). This is the preferred location from which to read the status of the XRH-1. The data found here is in the same format as described above for the XRH-1's

response to Device_Status_Request cycle to address 0. The CPU initializes this word to zero in memory before telling the XRH-1 the address of the communications region. The CPU can verify that the XRH-1 has accepted this region by this word reporting "Communications Region Established" along with the XRH-1 device type, subtype, and possible other status.

1 Communications Region Address (`.MXCRA`).

In normal operations, this location contains the same bus–address word as was written into the XRH-1's Device_Control_ Address 1 to assign a communications region. However, if the newly assigned communications region is rejected by the XRH-1 because of a pre–existing assignment, the XRH-1 will write the backplane bus address word corresponding to the established communications region in this location; apart from this and the error status (*CRR*, etc.) that will be written into location 0, the XRH-1 will otherwise ignore this region.

2 To XRH-1 (`.MXTO`). This word contains the bus–address word that specifies the location of an MSCB or zero. When the CPU wants to send an MSCB to the XRH-1, it checks this location. If this location contains zero, the XRH-1 is available to accept another MSCB: the CPU writes the address of the MSCB (which might be the head of an MSCB list); the CPU writes the same address to the XRH-1 as data in a Device_Control bus cycle to address 2.

If this location is non–zero, the CPU must wait until it becomes zero before writing into it; when the XRH-1 writes zero into this location, it will provide an interrupt to the CPU.

The XRH-1 responds to the Device_Control cycle to address 2 by adding the specified MSCB to its list of MSCBs to process. At some point, the XRH-1 will write zero into word 2 of the communications region (and, if enabled to do so, it interrupts the CPU) to signal its readiness to accept further MSCBs.

3 From XRH-1 (`.MXFRM`). When this location contains a non–zero value, the value is a bus–address word that defines the location of an MSCB (or the head of a list of MSCBs) being returned to the CPU by the XRH-1. When the CPU sees a non–zero value in this location, it copies that value as an MSCB list to be processed and then sets this location to zero.

When the XRH-1 has MSCBs to return to the CPU, it checks the value of this location. If the location is zero, the XRH-1 stores the address of the returned MSCB (list) in this location and, if enabled, it requests an interrupt. If the location is non–zero, the XRH-1 accumulates a list of MSCBs to be returned, and it checks periodically to see if the location has become zero.

4 XRH-1 Priority–Interrupt Level Assignment (`.MXPIA`). This location contains the slot number of the CPU to be interrupted by the XRH-1 and the priority level to which the XRH-1 is assigned. The CPU slot number is present in bits 3–6, in the same format as a bus–address word. The priority level is in bits 33–35. This location must be initialized by the CPU prior to assigning the communications region to the XRH-1, because the XRH-1 reads this location only once, following the assignment of the communications region. Changing the priority of the XRH-1 requires the assignment of a new communications region. A zero in the location directs the XRH-1 to avoid the use of interrupts.

5 XRH-1 SCSI Bus Identification Numbers (`.MXBID`). This location is set by the CPU (usually from data in its non–volatile memory) prior to the assignment of the communications region to the XRH-1. The location contains four SCSI ID numbers to be used by the XRH-1 on its four SCSI buses (`MSKSTR (MX%ID,.MXBID,377B11)`). The most significant bit of each ID number field (`MX%OFL==:200`), if set, signifies that the channel is to be kept offline.

3.12.2.2 Communications Protocol

Requests for service are synchronous with the activity of the CPU. That is, the CPU generates all requests for service. These requests are fashioned into MSCBs, the MSCBs are linked together as a list. A list of MSCBs is handed to the XRH-1 via its "To XRH-1" location in the communications region and by a Device Control function to address 2.

Acknowlegements, that is, a list of completed requests, are handed back to the CPU via the "From XRH-1" location in the communications region and an interrupt (if enabled).

Asynchronous events (e.g., a disk drive coming on–line) are announced by the XRH-1 via Attention status and an interrupt (if enabled). The CPU responds to an Attention interrupt by sending the XRH-1 an MSCB of status "Report Asynchronous or Error Status". The XRH-1 fills the MSCB with a Request Sense command and information to identify the clamoring unit, performs the Request Sense, and returns the MSCB in due course. The XRH-1 holds any MSCBs that the CPU supplies that have status "Report Asynchronous or Error Status" until it is necessary to report such an event. If the XRH-1 is holding such a command block at the time an asynchronous or error event occurs, the XRH-1 will report the event in that block and return it in due course (without explicitly requesting an interrupt to service the Attention condition).

To ensure that each interrupting condition is processed properly, on processing an interrupt the CPU will perform the following steps in sequence:

- clear the XRH-1 interrupt request for the backplane slot in question,
- examine each possible condition that could have caused the interrupt and handle each, and
- then dismiss the interrupt.

Similarly, the XRH-1 will treat each condition that can interrupt as follows: make appropriate changes to the communications region (To XRH-1, From XRH-1, and/or XRH-1 Status) before requesting the interrupt. In this way, system programmers can ensure that no source of interrupts may be lost.

A message of type "Release Communications Region" from the CPU to the XRH-1 directs the latter to bring all outstanding requests to completion. As the XRH-1 does so, the CPU will drain messages from the "From XRH-1" location and refrain from sending additional messages. When the process is complete, the XRH-1 will acknowledge the "Release Communications Region" message, write a status word in which "Ready" is zero in location 0 of the communications region, and release the communications region. Thereafter, the XRH-1 will be idle until a new communications region is assigned.

3.12.2.3 Mass–Storage Control Block

A Mass–Storage Control Block is an eight–word (`.MXBLN==:10`) region of memory that must be aligned on an address that is a multiple of eight. The format of an MSCB is depicted in Figure 3.7. Note that the 4–bit fields in words 4–6 are reserved for the XRH-1: the XRH-1 ignores the initial contents of those fields in an MSCB, and the XRH-1 may change the contents of those fields.

The meaning of the fields in the Mass–Storage Control Block is as follows:

Figure 3.7: Mass–Storage Control Block Format

Word	0–3	4–11	12–19	20–27	28–35	
0	*Link to next MSCB (a Bus–Address Word)*					.MXLNK
1	*CBS*	*CPU*	*Format*	*Byte Count*		.MXBYC
2	*Data Buffer Address or Command List Address (a Bus–Address Word)*					.MXDBA
3	*Bus*	*Target*	*LUN*	*AEC*	*MAL*	.MXTAR
4		*Command 0*	*Command 1*	*Command 2*	*Command 3*	.MXCMD
5		*Command 4*	*Command 5*	*Command 6*	*Command 7*	
6		*Command 8*	*Command 9*	*Command 10*	*Command 11*	
7	*ErrC*	*Sense Code*	*Status*	*ASC*	*ASCQ*	.MXSTA
	Request Sense Buffer Address (a Bus–Address Word)					

```
MSKSTR  (MX%ST,.MXBYC,17B3)      ;Command Block Status
MSKSTR  (MX%CP,.MXBYC,377B11)    ;CPU
MSKSTR  (MX%FT,.MXBYC,377B19)    ;Format
MSKSTR  (MX%BC,.MXBYC,177777)    ;Byte Count
MSKSTR  (MX%BS,.MXTAR,17B3)      ;Bus number
MSKSTR  (MX%TR,.MXTAR,377B11)    ;SCSI Target ID number
MSKSTR  (MX%LN,.MXTAR,377B19)    ;SCSI Logical Unit Number
MSKSTR  (MX%AE,.MXTAR,377B27)    ;Additional Error Code
MSKSTR  (MX%MA,.MXTAR,377B35)    ;MSCB Allocation
MSKSTR  (MX%C0,.MXCMD,377B11)    ;Command Byte 0
MSKSTR  (MX%EC,.MXSTA,17B3)      ;Error Code
MSKSTR  (MX%SN,.MXSTA,377B11)    ;Sense Byte
MSKSTR  (MX%SK,.MXSTA,17B11)     ;SCSI Sense Key
MSKSTR  (MX%SS,.MXSTA,377B19)    ;SCSI Status Byte
MSKSTR  (MX%AS,.MXSTA,377B27)    ;Additional Sense Code byte
MSKSTR  (MX%CQ,.MXSTA,377B35)    ;Additional Sense Code Qualifier
MSKSTR  (MX%AQ,.MXSTA,177777)    ;Both ASC and ASCQ
```

Link This is a link to the next MSCB on the list. Zero means that there are no further items on the list. Otherwise, the value is a bus–address word that specifies a main memory location at which an additional MSCB can be found.

CBS This is the Command Block Status field. When this MSCB is passed from the CPU to the XRH-1, the *CBS* (Command Block Status) field will indicate the action to be taken by the XRH-1. When this MSCB is passed back to the CPU, the *CBS* field will indicate the disposition of the command.

From the CPU to the XRH-1, this field is used as follows:

0 (`MX.CSS==:0`) SCSI command for immediate execution. This is the usual case. The particular command is encoded in six to twelve command bytes in the MSCB. The Bus, Target, and LUN fields specify the SCSI device that is the target for this command.

1 (`MX.CSA==:1`) Report asynchronous or error status. If the XRH-1 has asynchronous status from a SCSI device, it will report that status in this block. Otherwise, the XRH-1 will hold this command block until needed. This provides a means by which the XRH-1 can report asynchronous status or errors to the CPU.

The CPU is responsible for supplying correct data in the following words and fields of this MSCB: *CBS*, *CPU*, *Format*, *Byte Count*, *Data Buffer Address*, and the error reporting mode in the `.MXSTA` word.

The XRH-1 will return this MSCB to the CPU in any of the following circumstances:

- A SCSI target generates an Asynchronous Event.
  The returned MSCB will contain `MX.CSA` (Asynchronous Status) in CBS; the Bus, Target, and LUN fields will be filled in by the XRH-1 to identify the unit presenting asynchronous status.

- A SCSI bus has hung.
  The returned MSCB will contain `MX.CSX` (System Error) in CBS; the Bus field will be set. Asynchronous Return (`.MXASR==:200`) and Target Unknown (`.MXTUK==:100`) bits will be set in the `MX%AE` byte of the MSCB.
  If Target Blocking mode is enabled, the XRH-1 will set every target on the affected bus to the Target–is–Blocked state. The XRH-1 will find all MSCBs pending for targets on the affected bus and return them marked `MX.TIB` (Target is Blocked) in CBS. With respect to each target, the MSCBs will be returned in the same sequence as they were sent to the XRH-1. (If the XRH-1 has any cache writes pending for targets on this bus, it will hold them until the bus is reset and then restart them.)
  If Target Blocking mode is disabled, the XRH-1 will destroy all MSCBs pending for targets on the affected bus. (Cache writes pending for targets on this bus are held until the bus is reset and then they are performed.)
  The system will send a command to reset the affected bus. (Resetting the bus clears the Target–is–Blocked state for all targets on the bus.) The system will issue any commands needed to reconfigure devices after they have been reset. Then, the system will reissue any MSCBs that were returned marked as Target is Blocked.

- An asynchronous transfer (e.g., from the XRH-1's cache to disk) failed due to a SCSI error or a system error.

  If a SCSI command completed with status other than "good", the returned MSCB will be marked `MX.CSS` (SCSI Command was Performed) in the CBS field. If this (Report Asynchronous or Error Status) MSCB contains a Request Sense Buffer Address, return the Request Sense data there. Return appropriate values in all fields of the `.MXSTA` word.

  If the SCSI command was not performed because of a system error, the MSCB will be returned with `MX.CSX` (System Error) in CBS, and appropriate data will be returned in the `.MXSTA` word.

  The affected bus, target and LUN fields will be set. The *Command* bytes will be set to the write command (including the logical record number) that failed. The Asynchronous Return (`MXASR`) bit will be set in the `MX%AE` byte. The data buffer address returned in the MSCB represents the XRH-1 DRAM address of the data buffer for which this write failed.

  If Target Blocking mode is enabled, the XRH-1 will put the unit into the Target–is–Blocked state and return any MSCBs relating to this target, marking them Target is Blocked in CBS. (If the XRH-1 has any other cache writes pending for this unit, it will hold them until the Target–is–Blocked state is cleared.)

2 (`MX.CSM==:2`) XRH-1 Command.

This form of MSCB is used by the CPU to pass information directly to the XRH-1 instead of to one of the SCSI target devices connected to it. When the *CBS* field contains `MX.CSM`, only the *Link* and *CBS* fields have their normal meaning, although some of these XRH-1 commands will interpret fields such as *Bus*, *Target*, *LUN*, etc. as having their usual meanings (i.e., as specifying a SCSI bus and target). The interpretation of the *Command* bytes is particular to each XRH-1 command, as explained here.

The byte *Command 0* will contain the command code for the XRH-1. Additional bytes following *Command 0* may contain parameters for the command. Among the XRH-1 commands are the following:

0 Release Communications Region (`MX.RCR==:0`). This command directs the XRH-1 to bring all outstanding requests to completion. MSCBs presently held in the XRH-1 are to be completed. Data in the XRH-1's cache memory intended for output to the disk must now be written to disk. The XRH-1 may presume that this Release Communications Region command is the last MSCB in the "To XRH-1" queue. The CPU is responsible for draining any MSCBs returned via "From XRH-1" (including Asynchronous and Error Status blocks that are returned unused). Moreover, the CPU refrains from sending additional commands. The XRH-1 will acknowledge the completion of this command by returning the command block with "Good" Status. The XRH-1 will clear the status word in the communications region and refrain from using it again. The XRH-1 will remove "Communications Region Assigned" from its status word. (Thereafter, the status word is readable via Device_Status_Request to address 0, or by a Device_Control cycle directed to address 3.)

1 SCSI Bus Reset (`MX.SBR==:1`). The *Bus* field specifies which SCSI bus to reset. Bus reset is appropriate in circumstances such as "Emulex gross

error" or when the software believes that the bus or a device on the bus is hung.

SCSI Bus Reset causes all targets on the affected bus to return to their power–on condition. The Target–is–Blocked state is cleared. If the Monitor normally sends initializing commands to a target, those initializing commands must be repeated subsequent to a bus reset. If a target had been marked as cacheable prior to the bus reset, data that is already cached for that target will be retained as valid. However, as the target will be marked as uncacheable by the bus reset, the retained data is inaccessible (and it will not be written to the target, nor supplied in lieu of actual target data) until the Monitor completes any other initializing commands and sets the target to be cacheable again.

When this command is received by the XRH-1, it will locate all MSCBs relating to the affected bus and return each of them marked "Bus is Being Reset"; all such MSCBs will be returned on one list (which may contain other MSCBs as well). After all those MSCBs are queued to the operating system, the XRH-1 will return this reset command MSCB to signify that the reset operation has been completed.

2 Set Cache Use Parameters (`MX.CUP==:2`). The *Command 1* byte specifies the subfunction as follows.

  0 Invalidate cache (all) (`MX.CIA==:0`). Set all units uncacheable. Mark all cached data as invalid.

  1 Invalidate cache (unit) (`MX.CIU==:1`). Invalidate any cached data pertaining to the unit specified by the Bus, Target, and LUN fields. Set the specified unit as uncacheable.

  2 Unload cache (all) (`MX.CUA==:2`). Write all cached data to their respective disk units. Set all units uncacheable. Mark all cached data as invalid.

  3 Unload cache (unit) (`MX.CUU==:3`). Write any cached data pertaining to the unit specified by the Bus, Target, and LUN fields to that unit. Mark cached data for this unit as invalid. Set the specified unit as uncacheable.

  4 Validate cache (all) (`MX.CVA==:4`). Write all cached data to their respective disk units.

  5 Validate cache (unit) (`MX.CVU==:5`). Write any cached data pertaining to the unit specified by the Bus, Target, and LUN fields to that unit.

  6 Enable caching for unit (`MX.ECU==:6`). Mark the unit specified by the Bus, Target, and LUN fields as cacheable.

  7 Release cached data (`MX.RCD==:7`). Release the block of cache memory specified by the bytes *Command 2* and *Command 3*.

  When a cached write to disk encounters an error subsequent to the

XRH-1's report of "good" status, the XRH-1 sends an asynchronous error report MSCB. That error report contains the identification of the bus, target disk, disk address, etc. The data associated with the transfer is preserved in the XRH-1's memory and identified as a particular cache block from which the CPU can read the data. Following error recovery steps, this command is issued by the CPU to tell the XRH-1 that it can reuse the indicated cache block.

3 Negotiate wide transfer. (Not presently used.) The XRH-1 automatically enters negotiation with targets to determine whether or not they support 16–bit wide transfers. If a target supports 16–bit transfers, the XRH-1 does them.

4 Negotatiate synchronous data transfer (`MX.SYN==:4`). (Not presently used.) This command directs the XRH-1 to negotiate synchronous (or asynchronous) transfer mode with all units or with a selected unit. The *Command 1* byte selects a particular function, as follows:

0 Negotiate Asynchronous Transfer Mode, all units (`MX.ATM==:0`).

1 Negotiate Synchronous Transfer Mode, all units. (`MX.STM==:1`). Synchronous transfers will be made to all units that support synchronous transfer mode.

2 Negotatiate Asynchronous Transfer Mode on the unit specified by the Bus and Target fields; all LUNs are affected (`MX.ATU==:2`).

3 Negotatiate Synchronous Transfer Mode on the unit specified by the Bus and Target fields; all LUNs are affected. (`MX.STU==:3`).

5 Set target timeout. (Reserved for future use.)

6 Set target disconnect privilege. (Reserved for future use.)

7 Set SCSI target priorities. (Reserved for future use.)

10 Write DRAM (`MX.WDB==:10`). This command directs the XRH-1 to transfer information from system memory to its internal buffer (cache). This command, together with the Read DRAM command (below), provide a means of testing the data path between system memory and the XRH-1; further, these commands provide a means of extensively testing the XRH-1 internal memory.

Additional parameters are as follows:

- *Byte Count*: the count of 8–word memory lines to transfer.
- *Data Buffer Address*: the bus–address word corresponding to the system memory from which data will be read.
- *Command 1—Command 3*: the DRAM address into which to write the data. The DRAM address is a 19–bit field whose 5 most–significant bits are right–justified in the byte *Command 1*, whose next 8 bits are in *Command 2*, and whose least significant 6 bits are left-justified in

Command 3; all otherwise undefined bits in *Command 1* and *Command 3* must be zero.

- Bus: Specifies which one of the XRH-1's five "DMA machines" is used for this transfer. Values 0–3 address the DMA machine of the corresponding SCSI Bus; value 4 signifies the DMA machine associated with transfers between main memory and the XRH-1's DRAM (data cache).

Upon completion of this command, the MSCB will be returned with the Status field set either to zero, indicating a successful operation, or to a bitwise encoding of the reason for failure:

1	DRAM/Main Memory transfer failure
2	DMA and DDMA counts don't match
4	System bus busy timeout
10	System bus timeout
20	System memory reported a parity error
40	Final DMA count is greater than 1
100	DMA count is zero
200	Invalid parameter in the MSCB

11 Read DRAM (`MX.RDB==:11`). Parameters are as described in the Write DRAM command, above.

12 Target Blocking Control (`MX.TBC==:12`). This command controls the behavior of the XRH-1 when SCSI bus errors occur; see also "Error Reporting" and "Error Handling", pages 321 and 322, respectively. Target Blocking mode is a global state of the XRH-1, either enabled or disabled, which affects all targets. The specific command function is selected by the *Command 1* byte, as follows:

0 Disable Target Blocking mode (`MX.DTB==:0`). The XRH-1 makes no special efforts to help effect error recovery. This is the mode established by default following a successful assignment of the communications region.

1 Enable Target Blocking mode (`MX.ETB==:1`). The XRH-1 will set the Target–is–Blocked state for a target device when the device returns a command status other than "good". While a target is in the Target–is–Blocked state, the XRH-1 will refuse to perform any MSCB that addresses the blocked target and it will return the MSCB marked "Target is Blocked" in the CBS field. All LUNs of a blocked target are blocked. The target remains blocked until a Clear Target–is–Blocked command is received by the XRH-1.

2 Clear Target–is–Blocked (`MX.RTB==:2`). The XRH-1 will clear the Tar-

get–is–Blocked state for the target specified by the Bus and Target fields. All LUNs of the affected target are unblocked. Any MSCBs in the XRH-1, pending for this target, are returned marked "Target–is–Blocked" before the MSCB containing this command is returned. Thus, this command forces all pending MSCBs to be returned to the CPU before unblocking the target.

3 Set Target–is–Blocked (`MX.STB==:3`). The XRH-1 will set the Target–is–Blocked state for the target specified by the Bus and Target fields. All LUNs of the affected target are blocked.

13 Heartbeat (`MX.HBT==:13`). This function tells the XRH-1 that a unit of time has elapsed. The XRH-1 uses the heartbeat to age the data in the write cache and other purposes. Generally, the Monitor will attempt to send this message approximately once per second. The byte *Command 1* will describe the operating system's power condition: 0–"green"; 1–"yellow"; 377–"red".

14 Set Inquiry Response (`MX.SIR==:14`). This function tells the XRH-1 to change its response as a target to an Inquiry command by supplying in its response bytes 21–31 a copy of the data found in the bytes *Command 1–Command 11*, respectively.

15 Set Serial Number (`MX.SSN==:15`). This function tells the XRH-1 to change its response as a target to an Inquiry/EVPD command with page code 0x80 by supplying in its response 0x0b in byte 3 (page length) and in bytes 4–14 a copy of the data found in the bytes *Command 1–Command 11*, respectively.

3–15 Reserved

From the XRH-1 to the CPU, this field is used to communicate the status of the completed operation:

0 SCSI Command was Performed (`MX.CSS==:0`). The SCSI operation requested by this MSCB was performed. The *Status* byte contains the ending status. "Good" status indicates that the command was completed successfully; in this case the `.MXSTA` word will be set to zero. Other values of status indicate error and abnormal situations; in the automatic error reporting modes (§3.12.2.4), further information will be found in the *ErrC* (Error Code), *Sense*, *ASC* (Additional Sense Code), and *ASCQ* (Additional Sense Code Qualifier) fields. Errors reported in this way are those that the target devices report; in contrast see "SCSI Error Status Report" below.

1 Asynchronous Status Report (`MX.CSA==:1`). This MSCB reports that the device specified by the *Bus*, *Target*, and *LUN* fields has provided asynchronous (unsolicited) status to the XRH-1 Mass–Storage Interface Processor.

2 XRH-1 Command Complete (`MX.CSM==:2`). The command bytes are returned as the CPU had set them. The `.MXSTA` word will be returned as zero.

3 SCSI Error Status Report (`MX.CSE==:3`). The XRH-1 reports errors that it has detected while attempting to communicate with a particular SCSI target device. Errors detected by the XRH-1 are disjoint from errors that the target reports to

the XRH-1. The *Status* byte (which, in this case, will not contain SCSI status) will contain details of the particular error.

1 SCSI Selection Timeout (`.MXSTO==:1`). The indicated target does not exist or it lacks power.

2 Bus Protocol Error.

4 (`MX.CSX==:4`) System Error Report. The XRH-1 reports errors pertaining to its attempts to use system resources; e.g., bus timeout, parity errors, etc. Messages relating to errors found in command blocks are reported this way too. The *Status* byte (which, in this case, will not contain SCSI status) will contain details of the particular error:

1 Byte count insufficient (`.MXBCI==:1`). This MSCB specifies a command in which the transfer length (or allocation length or parameter list length) exceeds the given byte count.

2 Not Implemented.

3 Memory timeout (`.MXMTO==:3`). The XRH-1 attempted to use the given *Data Buffer Address* or *Command List Address*, but the memory did not respond.

4 Byte count excessive (`.MXBCE==:4`). The *Byte Count* field of this MSCB contains a value that exceeds the count specified or implied by the command bytes.

5 Format wrong (`.MXFTW==:5`). This MSCB specifies a *Format* that is inconsistent with the data format implied by the command bytes.

6 Memory parity error (`.MXMPE==:6`). The XRH-1 attempted to read from the data buffer in system memory, but the memory reported a parity error. The failing address (the address of the memory line on which the failure occurred) will be stored so that it is accessible via the `.MXERA` device status register (until a subsequent error address is stored). If a Request Sense buffer was associated with this request, the XRH-1 will return the failing address there also. If Target Blocking mode is enabled, the affected target device will be blocked.

7 Non–recoverable SCSI parity error (`.MXSPE==:7`). After several attempts, this command has been abandoned because of parity errors reported on the SCSI bus.

10 Final DMA/DDMA byte count error (`.MXDBC==:10`).

11 Not implemented.

12 DRAM parity error (`.MXDRP==:12`). The XRH-1 detected a parity error in its internal memory, which is used as a cache for the peripheral devices. If the error is associated with a read operation from a peripheral device, the meaning of this error is that the data in the DRAM was a modified copy of the data on the device (which was supposed to be written to the device) and the DRAM data is now corrupted with bad parity. If this

error is presented asynchronously, it means that a device write (which had previously been acknowledged as complete) was in fact cached and cannot now be completed because the cached data is corrupted.

13 (`.MXIES==:13`) Internal Error Status. Further information is reported in the *ASC* byte:

  01 Obsolete.

  02 Obsolete.

  03 DMA finished during a device write odd–byte transfer.

  04 Obsolete

  05 Obsolete

  06 DMA count not zero after DMA interrupt.

  07 Premature phase change during selection—target or hardware error.

  10 Obsolete.

  11 Internal Check Condition failed.

  12 Data structure error in target block during reselection.

  13 Reselection error; microcode error.

  14 Unexpected status during command phase.

  15 Expecting an Identify message and did not get one.

  16 Illegal SCSI phase (4); hardware error.

  17 Illegal SCSI phase (5); hardware error.

  20 Obsolete.

  21 Obsolete.

  22 Hardware error. The error condition vector was taken, but no problem was found. (A transient power fluctuation may cause this.)

  23 Obsolete.

  24 Emulex reports a “gross error” during a data transfer.

  25 Emulex reports that it was given an illegal command.

14 Final Emulex transfer count non-zero (`.MXENZ==:14`).

15 *Byte Count* field too large (`.MXBTL==:15`). A single transfer or a single component of a long transfer exceeds 255 cache lines. (The precise byte count at which this occurs depends on the alignment and transfer mode.)

16 CBS Field Invalid (`.MXCBX==:16`). The value found the CBS field is not one of the legal values. The original value found in the CBS field is returned in the *Sense* field.

5 (`MX.TIB==:5`) Target is Blocked. The XRH-1 is presently blocking commands to the target device identified by the Bus and Target fields. Access to the target's LUNs is also blocked. Commands returned with this status have not been attempted. They should be repeated after error–handling routines have issued a Clear Target–is–Blocked command (see page 313).

6 (`MX.REB==:6`) Returned Error Block. After the XRH-1 has been given a "Release Communications Region" command, unused MSCBs of the type Asynchronous or Error Status Report are returned with this value.

7 (`MX.BBR==:7`) Bus is Being Reset. The XRH-1 has returned this MSCB because the selected SCSI bus is in the process of being reset. When a bus is reset, all the MSCBs pertaining to the bus are returned to the operating system so that they can be retried later; see page 311. The SCSI Status byte will be returned with a code that indicates the state of this MSCB when the bus was reset: a 0 indicates the command was not yet started; a 1 indicates that the command had been started, in which case the *ASC* byte indicates the progress of the command, as follows:

  0 Data Out phase (Write operation).
  1 Data In phase (Read operation).
  2 Command phase.
  3 Status In phase.
  4 Disconnect state.
  5 End of SCSI command.
  6 Message Out phase.
  7 Message In phase.
  10 Selection with ATN and Stop started.
  11 Selection with ATN started.
  20 Long transfer and Data Out phase (write operation).
  21 Long transfer and Data Out phase continuation.
  22 Long transfer, Data Out, Emulex complete, but DMA is still busy (sometimes OK for tape transfers).
  23 Long transfer, Data Out, error restarting the last transfer.
  25 Long transfer, Data Out complete.
  40 Long transfer, Data In phase (read operation).
  41 Long transfer, Data In phase, continuation.
  42 Long transfer, Data In phase, Emulex complete, but DMA is still busy (sometimes OK for tape transfers).
  43 Long transfer, Data In, error restarting the last transfer.
  45 Long transfer, Data In complete, waiting for Emulex to finish.

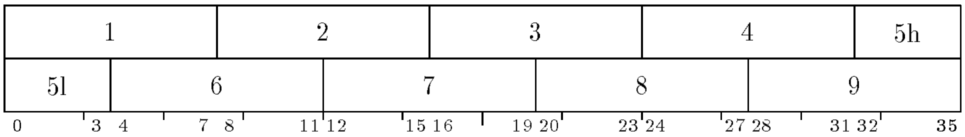
CPU The CPU records its own slot number in this field so the XRH-1 will know to whom to respond.

Format This field specifies the format used to pack data into or unpack data from 36–bit words. Further, the most significant bit of the *Format* field (`.MXFCL==:200`) controls the interpretation of the contents of the word at `.MXDBA` as either a *Data Buffer Address* or as a *Command List Address*.

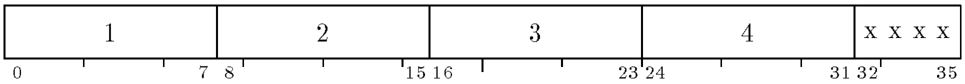
The data formats supported by the XRH-1 are listed below. The format names, `.MXF36`, `.MXF32`, and `.MXF40`, have the follow mnemonic significance: the number represents either the number of bits on the recording medium per 36–bit word or, equivalently, the number of 8–bit bytes on the medium per TOAD-1 System memory line.

0 36–bit mode (`.MXF36==:0`). Nine 8–bit bytes from two consecutive 36–bit words, are written to (or read from) the selected target in the order indicated; the fields labeled “5h” and “5l” are the high– and low–order portions of byte 5, respectively.

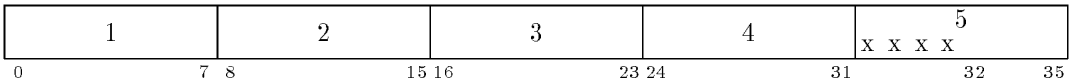
This format is used for data transfers between the CPU and SCSI disks. It may be used also in transfers to high–density tapes. In this format, the transfer–length field within the *Command n* bytes and the *Byte Count* field in the MSCB indicate the number of 8–bit bytes transferred. The transfer length (and byte count) must be a multiple of 9; that is, an even number of words must be transferred.



1 32–bit mode (`.MXF32==:1`). Four 8–bit bytes, left–justified in a 36–bit word, are written to (or read from) the selected SCSI target in the order indicated. Bits marked “x” are ignored during writes and are set to zero on reads. In this format, the transfer–length field[37] is the number of 8–bit bytes transferred by this command. This format is used in commands that transfer operating information from the target device to the CPU and in commands that send parameters from the CPU to the target. This format may be used for transfers to 9–track tape or to support devices that are not formatted for the TOAD-1 System (e.g., CD-ROMs).



2 “Dump” mode (`.MXF40==:2`). Each 36–bit word is written to (or read from) five consecutive bytes on the target device. The figure below depicts the relationship between five consecutive bytes on the recording medium and a computer word in memory.



The first four bytes on the recording medium correspond to bits 0–7, 8–15, 16–23, and 24–31 in the data word, respectively; the least–significant 4 bits of the fifth recorded byte correspond to bits 32–35 in the data word. The most–significant 4 bits of the fifth recorded byte are written to the device as zero, and they are

---

[37] The transfer–length field is contained within the *Command n* bytes; it may be called either “transfer length”, “allocation length”, or “parameter list length”, depending on the particular command. The CPU (operating system) is responsible for making this field consistent with the *Byte Count* field of the MSCB.

discarded on input; these bits are marked "x" in the figure above.

In this format, the transfer–length field within the *Command n* bytes and the *Byte Count* field should both contain the number of 8–bit bytes transferred by this command; i.e., five times the number of words transferred. This format may be used for transfers to or from tape. This mode is compatible with a popular format used to record 9–track tape on the DECsystem–10 and DECSYSTEM-20; it is intended for reading tapes made on such systems.

Byte Count From the CPU to the XRH-1, this field contains the length of the data buffer area measured in 8–bit bytes. Note that, in a Read command, the byte count is a multiple of 32 in 32-bit mode, or in 36–bit mode a multiple of 36, because the XRH-1 writes in memory only on the basis of memory lines (8 words). The byte count and transfer format are combined to form the transfer length, allocation length, or parameter length field within the command bytes. In no event shall the XRH-1 overstep the region defined by the *Data Buffer Address* and *Byte Count* fields.

From the XRH-1 to the CPU, the *Byte Count* field will be set by the XRH-1 to the residual byte count; i.e., the number of bytes allocated (in this field, from the CPU) but not used by the actual data transfer.

The residual Byte Count should be zero in all disk data operations and in all tape data writes. A non–zero residual is acceptable in commands that do not transfer data from the medium (e.g., Inquiry, Request Sense, Mode Sense); the program may not know in advance how many bytes the device will transfer. In such cases the residual can be ignored, or it can be compared to counts contained within the data. In read operations from a tape, a non-zero residual indicates that a short record has been read; this should be coupled with Check Condition status and sense data that corroborates the short length of the data record.

Commands in which the device has more data than the given allocation length will end with a zero residual byte count. Commands that do not transfer data from the medium (e.g., Inquiry) will end with "good" status, but the excess of data over allocation can be determined by examination of counts contained within the data. In transfers of data to or from the medium, when the data buffer is too short for the medium, the command will end with Check Condition status and sense ILI (illegal length indicator, for tape and other sequential–access devices) and sense information containing a negative residual (excess of data over allocation).

Data Buffer Address This is a bus–address word in which *D*, the device bit, must be zero. If the most–significant bit (`.MXFCL`) of the *Format* field is 0, this word specifies the main memory address of the data source (write) or destination (read). If the most–significant bit of the *Format* field is 1, this word specifies the main memory address of a command list: see "Long Transfers", below.

When this field is the address of the data buffer, then if the bus–address word is a multiple of 8, the address is said to be aligned. Many cautions must be observed for transfers that are not aligned; see "Unaligned Transfers", below. Data will be fetched from (or stored in) consecutive physical locations[38] to the extent defined by the *Byte Count* field or the actual amount of data transferred, whichever is smaller. In a device read operation, if the actual number of data bytes transferred does not precisely fill an 8–word memory line, an entire memory line will be written to memory containing

---

[38] The locations are all in the same backplane slot: the XRH-1 will not alter the slot–number field of the BAW.

the final data bytes and sufficient zero bytes to fill the remainder of the line. The byte count and transfer alignment must specify the transfer of 256 or fewer memory lines (2 Kw or 4 pages in memory; this is 8, 9, or 10 K bytes on the medium, depending on the data mode). Longer transfers must be split into multiple transfer commands: see "Long Transfers", below. A single transfer must not cross the boundary from physical address ...3777 to ...4000 (or from ...7777 to ...0000).

Bus This is the number of the particular SCSI–2 bus being addressed. The legal values are in the range 0–3.

Target This field stores the SCSI identification number of the targeted device interface. The legal values are in the range 0–15.[39]

LUN This is the logical unit number of the targeted device. The legal values are in the range 0–7. For targets that support only one logical unit, this field should be set to zero. The XRH-1 uses this value to generate the Identify message to the target interface.

AEC Additional Error Code. When a Report Asynchronous or Error Status MSCB is returned with error status, this field will be set to alert the CPU software to the unusual nature of the report. Bus errors are marked as asynchronous returns and target unknown. An error from an asynchronous (cache) write will be marked as an asynchronous return and will provide the status of the affected target.

MAL MSCB Allocation. In order to control the flow of requests from the processor to the XRH-1, the XRH-1 returns a count of available resources in this field. The resources reported correspond approximately to MSCBs. However, an MSCB that calls for a long transfer will consume more of the XRH-1's resources. Further, an MSCB that transfers into the data cache will occupy XRH-1 resources until the data cache is transferred to disk. The normal value of this field is 255; lower values indicate that the operating system should limit the rate at which requests are made, until the value increases.

Command This field contains a six–, ten–, or twelve–byte SCSI command directed to the target device specified by the *Bus*, *Target*, and *LUN* fields. The values and the command length are as defined in the SCSI–2 specification. Some of the SCSI commands are listed below.

000 Test Unit Ready. This command provides a means to check whether a logical unit is ready.

010 Read (tape) The operating system will specify a transfer length in bytes, embedded in the SCSI command. Elsewhere in the MSCB, the system specifies *Format*, *Byte Count*, and *Data Buffer Address*.

The number of words read from the tape depends on the byte count and the format.

012 Write (tape).

022 Inquiry. The CPU asks for information from the target device so that it can determine system configuration.

---

[39] Targets numbered 8 and above can be accessed only via the "P" cable, as defined in the SCSI–3 specification.

050 Read (disk). The operating system will specify an in–unit address (Logical Block Number) and the transfer length (Block Count) embedded in this command. Elsewhere in the MSCB, the operating system will specify the *Format*, the *Data Buffer Address*, and a *Byte Count*. For disk read and write, the format is usually 36–bit.

052 Write (disk).

If XRH-1 disk caching is enabled for the specified unit, the XRH-1 will copy the data to its cache and report the operation as being complete before the actual cache–to–disk transfer is performed. (If the target device's write cache is enabled, it will report the operation as being complete as soon as it recieves the data, before the actual transfer to the medium is performed.

`.MXSTA` *word* This word specifies the method by which the XRH-1 will report errors associated with this command. When this word is sent from the CPU to the XRH-1, it should contain one of the following:

- All Zero, to select "Automatic Short" error reporting.
- A Bus–Address Word, selecting "Automatic Long" error reporting.
- Bit 0 set to 1, selecting "Manual" error reporting.

In the automatic modes, the XRH-1 will report some error information in this word. Additional information about error reporting and error handling is presented below.

3.12.2.4 Error Reporting

SCSI devices hold "sense" information relating to the error conditions encountered by the most recent command. This information can be read by means of the Request Sense command. However, a successful Request Sense command clears the error conditions, so only the Request Sense command that immediately follows an error can obtain the relevant sense data.

The XRH-1 implements three modes of sense reporting: Automatic Short, Automatic Long, and Manual. The mode is selected by the value in the `.MXSTA` word of the MSCB that the CPU provides to the XRH-1.

Automatic Short mode is selected by an all–zero value in the `.MXSTA` word of an MSCB. In automatic short mode, if no error occurs, the `.MXSTA` word will be returned as zero. However, if an error occurs, the various fields of the `.MXSTA` word will be filled as follows:

ErrC This field is set from the Error Code byte of the data returned by Request Sense. If the Error Code byte contains 0xF0 or 0x70, this field will be set to 0, signifying a current error. If the Error Code field contains 0xF1 or 0x71, this field will be set to 1, signifying a deferred error. Any other values of Error Code cause the XRH-1 to set this field to 2, undefined.

Sense Code This field is a copy of the Sense Code byte found in the Request Sense data.

Status The ending status of the SCSI transaction engendered by this command. Status 2, Check Condition, usually means that an error has occurred and corrective action may be necessary.

ASC This is a copy of the Additional Sense Code byte from the Request Sense data.

ASCQ This is a copy of the Additional Sense Code Qualifier byte from the Request Sense data.

Automatic Long mode is selected when the CPU supplies a bus–address word in the `.MXSTA` word of an MSCB. In Automatic Long mode, if no error occurs, the `.MXSTA` word will be returned as zero. However, if an error occurs, the various fields of the `.MXSTA` word will be filled as described above for Automatic Short mode; further, up to 256 bytes of Request Sense data will stored in memory at the address specified by the original .MXSTA word. Automatic Long mode provides more comprehensive sense monitoring than that provided by Automatic Short mode.

Manual mode is selected by setting bit 0 of the `.MXSTA` word to 1. In manual mode, the XRH-1 does not perform a Request Sense command. If the command ends with "Good" status, the XRH-1 will set the `.MXSTA` word to zero. Otherwise, when a command returns a status other than "Good", that status will be reported in the *Status* field of the `.MXSTA` word; the rest of the word will be set to zero. After an error, it is the program's responsibility to issue a Request Sense command to elicit the sense data.

3.12.2.5 Error Handling

For an application such as TDBoot which makes a single request and waits for it to complete, errors create no special difficulty in synchronizing the program, the XRH-1, and the SCSI device. For this use, the XRH-1 is not required to perform any special synchronization operations.

However, for the Monitor, which may have multiple requests queued to a single target device, resynchronization when errors occur requires a special effort from the XRH-1 and the Monitor.

When a SCSI transfer ends with Check Condition status (or any status other than "Good"), further use of the affected target device is prevented by the XRH-1 until the program finishes its error recovery and logging function. Upon detection of an error on a target device, the XRH-1 sets the affected unit to the Target–is–Blocked state. While the XRH-1 has a unit marked Target–is–Blocked, MSCBs that attempt to reference that device are returned to the program with "Target–is–Blocked" status (CBS is returned with the value 5, `MX.TIB`). Target–is–Blocked state will persist on the device until the program sends a "Clear Target–is–Blocked" command (see page 313).

This effects "pipeline clear" and resynchronization that allows the operating system software to perform error recovery and error logging appropriate to the specific MSCB associated with the error and to finish the error–recovery process for that MSCB before deciding how (or whether) to continue processing other MSCBs that were pending for the affected device.

If Target Blocking mode is enabled, a target will be blocked if a system error occurs during a transfer involving the target. It is the responsibility of the operating system to clear any SCSI check condition that might be associated with the system error.

To accomodate the simple, synchronous access mode of TDBoot and similar programs, the XRH-1 will not do target blocking until it receives an Enable Target Blocking Mode command from the CPU. Target blocking mode is disabled initially and following any assignment of a communications region.

3.12.2.6 Long Transfers

A 1 in the most significant bit in the Format field (`.MXFCL`) signifies a long transfer; i.e., a transfer that is composed of one SCSI command and more than one contiguous region of physical memory. The memory regions of a long transfer are described by a command list. The command list is a collection of command entries; each command entry is a pair of words. The first command list entry is at the address specified in word `.MXDBA` of the MSCB; this address must be aligned at the first word of a memory line. The XRH-1 will interpret each command list entry either as a transfer command, a jump command, or a halt command:

Transfer Command — The first word contains 0 in bit 0 and a non–zero byte count in bits 20–35; the second word is the bus–address word describing the start of the data buffer for the indicated number of bytes; the data buffer must be contained entirely in the one physical module identified by the slot number field of the BAW. After performing the indicated part of the transfer, the XRH-1 will fetch another command list entry (a pair of words) from the next consecutive memory locations. The byte count and transfer alignment (see Unaligned Transfers, below) must specify the transfer of fewer than 256 memory lines. Moreover, a single transfer must not cross the boundary from physical address ...3777 to ...4000. Longer transfers (and those that cross the address boundary) must be split into multiple Transfer Commands.

0	*Reserved*	*Byte Count*
Data Buffer Address (a Bus–Address Word)		
0	1 19	20 35

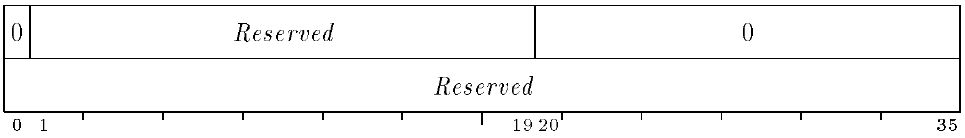
Jump Command — The first word contains 1 in bit 0; the second word is a bus–address word specifiying the address from which the XRH-1 will continue to fetch the next command list entry. The specified address must be aligned to the first word of a memory line.

A jump command is permitted only in words 6 and 7 of a memory line containing command list entries.

No data is transferred by this command.

1	*Reserved*
Command List Continuation Address (a BAW)	
0	1 35

Halt Command — The first word contains 0 in bit 0 and zeros in bits 20–35. Command list processing terminates.



The sum of the byte counts in the transfer commands must exactly match the byte count field given in the MSCB.

If the transfer is not aligned (to an address and length in words that are both multiples of 8), the information that follows in "Unaligned Transfers", below, applies also.

3.12.2.7 Unaligned Transfers

The XRH-1 Mass–Storage Interface Processor will handle transfers that are not aligned, subject to the following restrictions and limitations. Regardless of how a transfer is described, the XRH-1 will perform data transfers only by moving (reading or writing) entire memory lines (which are 8 words long and aligned to addresses that are multiples of 8).

On device write (memory read) operations, the XRH-1 will read words unrelated to the transfer before and/or after the actual region being transferred. For disk writes in which the data transfer fills an integral number of sectors and for writes to tape (where the output record length is determined by the number of words transferred), no problem arises from reading and discarding a few extra words at either end of the specified data buffer.

However, when writing a partial record to disk, the transfer must be padded (usually with zero words) to fill a whole record. To accomplish this padding, generally two buffers are required: the first, called an "edge buffer" is a single memory line, the second is an aligned buffer, the size of a disk sector, containing zeros. If the last word of the data portion of the transfer is not at the end of a memory line, then the partial last line of the transfer is copied to the edge buffer and padded with zeros. Then a long transfer command list must be created that specifies the orginal source of the data, the edge buffer containing the last line of data with padding, and the zero buffer. For example, assuming 200–word (decimal 128) disk sectors and a transfer in 36–bit format (9 bytes per double word), a transfer of 101–words (65 decimal) starting at bus address 024000123455 (ending at bus address 024000123555) can be accomplished by the following steps:

- Copy words from addresses 024000123550–024000123555 to the edge buffer (at addresses 024072001000–024072001005). Zero the words at addresses 024072001006–024072001007 (zero pad to the end of the edge buffer).
- Assuming the buffer of zeros is 200 words long, at address 024072002000 construct the command list shown below:

```
Byte(1)0(19)0(16)411     ;73 words (123455-123557) * 9 bytes/2 wd, round down
024000123455             ;from the specified data area
Byte(1)0(19)0(16)44      ;6 data + 2 pad words * 9 bytes/2 words
024072001000             ;address of the edge buffer
Byte(1)0(19)0(16)423     ;75 words * 9 bytes/2 words, rounded up
024072002000             ;from the zero buffer
0                        ;halt command
0                        ;total byte count 411+44+423 = 1100 = 9/2 * 200
```

On device read (memory write) operations, other delicate maneuvering is needed to prevent the destruction of important information. The XRH-1 will obliterate any data in the first memory line of the transfer prior to the transfer starting address and any data remaining in the memory line following the last word transferred. For example, a 128-word transfer starting at bus address 024000123455 will affect the words at addresses 024000123450–024000123454 and the words at addresses 024000123655–024000123657.

Operating system software is responsible for preserving the words that the XRH-1 would otherwise destroy. Among the strategies for doing this is to treat an unaligned transfer as a long transfer. The first partial memory line is read into an aligned in–system memory line buffer. The largest part of the transfer is aligned and goes directly to the intended addresses. The last partial line of the transfer is read into a second, aligned, in–system line buffer. At the conclusion of the transfer, the system must copy the data from the in–system line buffers to the intended addresses. The words in the in–system line buffers that were not transferred are obliterated, but they did not contain information that had to be retained. For example, conversion of the example above would result in the following command list (assuming 36–bit format):

```
Byte(1)0(19)0(16)15      ;3 words * 9/2 bytes words, rounded down
024072001005             ;in--system buffer, aligned for first part of transfer
Byte(1)0(19)0(16)1034    ;120 words * 9/2 bytes words
024000123460             ;intended buffer, Alignment Zero
Byte(1)0(19)0(16)27      ;5 words * 9/2 bytes, rounded up
024072001110             ;second in--system buffer.  Alignment Zero
0                        ;halt command
0
```

This example shows 36-bit mode, in which two words are transferred as 9 bytes. In the command list, each transfer command must have a byte count from which the XRH-1 will determine the precise number of 8–word lines to transfer; in a transfer with a non–zero alignment, the program should truncate any fractional byte count to the next–lowest integer. The sum of the transfer byte counts must exactly match the MSCB Byte Count field.

When a disk read operation specifies a transfer that is not an integral number of sectors, following the transfer of the last memory line into the edge buffer, the remainder of the disk sector must be transferred to the “bit bucket”, a buffer capable of holding a sector.

The XRH-1 determines the number of memory lines in a transfer from three quantities: the byte

Table 3.6: Byte Count Adjustment and Divisors

Alignment	Transfer Format		
	`.MXF32`	`.MXF36`	`.MXF40`
0	0	0	0
1	4	5	5
2	8	9	10
3	12	14	15
4	16	18	20
5	20	23	25
6	24	27	30
7	28	32	35
Divisor	32	36	40

count, the alignment of the transfer address, and the format. In a long transfer, the byte count and transfer address are as specified in a transfer command; otherwise they are the byte count and data buffer address as specified in the MSCB. The alignment is a number in the range 0–7 taken from bits 33–35 of the transfer (or data buffer) address. The XRH-1 determines the number of memory lines in a transfer by adding to the given byte count the adjustment from Table 3.6 and dividing the sum by the format–specific divisor, as shown in the table. (The divisor, which depends on the selected format, is the number of 8–bit bytes on the medium per 8–word memory line.) If the division results in a remainder, the quotient (the number of memory lines) is increased by one.

In an unaligned transfer, the first transfer command may specify an unaligned address and the byte count need not specify an integral number of cache lines. If the first transfer command does not specify the entire unaligned transfer, then the last transfer command is expected to have an address that is aligned; but, again, the byte count need not specify an integral number of cache lines. The transfer commands that come between the first and last commands must specify aligned addresses and entire cache lines.

3.12.3 Operation of the XRH-1 as a SCSI Target

[This section is incomplete, as is the implementation of target microcode in the XRH-1. There are not presently any plans to advance this implementation.]

To this point the operation of the XRH-1 as a SCSI initiator has been described. This section describes the operation of the XRH-1 as a SCSI target.

When two or more TOAD-1 System systems are arranged as a “loosely coupled” multiprocessor (a “cluster”), their XRH-1s are connected to the same, shared SCSI bus. In this situation, one XRH-1 may act as a SCSI target with respect to another acting as a SCSI initiator.

3.12.3.1 Commands recognized as a Target

When operating as a target, the XRH-1 recognizes and supports only the mandatory commands required by the SCSI specification. These commands are

- Inquiry
- Request Sense
- Send
- Send Diagnostic
- Test Unit Ready

3.12.3.2 Response to Request Sense Command

(To be written.)

3.12.3.3 Response to Inquiry Command

The XRH-1 will respond to the SCSI Inquiry command by supplying 0x03 in byte 0 (processor), 0x1F in byte 4 (additional length), the eight–bit ASCII text "`XKL`" left justified and padded with blanks in bytes 8–15 (vendor identification), the text "`XRH-1`" left justified and padded with blanks in bytes 16–31 (product identification) and a product revision level text string, representing the version of target microcode, in bytes 32–35.

This response can be modified by the XRH-1's host CPU. An MSCB with CBS = 2 and the *Command 0* byte = 14 will set response bytes 21–31 from the MSCB's bytes *Command 1–Command 11*, respectively.

When initialized, the XRH-1 will respond to the SCSI Inquiry command with EVPD and page code 0x80 by returning byte 3 = 0, no serial number present. This response can be changed by the XRH-1's host CPU. An MSCB with CBS = 2 and *Command 0* byte = 15 will set response bytes 4–14 from the MSCB's bytes *Command 1–Command 11*, respectively. Also response byte 3 will be set to 0x0B, indicating 11 bytes of serial number string are present.

3.12.3.4 Response to Send Command

(To be written.)

3.12.3.5 Response to Send Diagnostic Command

(To be written.)

3.12.3.6 Response to Test Unit Ready

The XRH-1 will respond to a Test Unit Ready command with one of the following responses:

- Good. (Status 0.) This XRH-1 is ready to function as a target.
- Not Ready. (Status 2, check condition; Sense 2, not ready; ASC/Q = 0x0401, Unit is in process of becoming ready.) This XRH-1 has received initializing commands from its host CPU and is expected to be ready soon.
- Not Ready. (Status 2, check condition; Sense 2, not ready; ASC/Q = 0x0403, Manual intervention required.) This XRH-1 has not yet received appropriate initializing commands from its host CPU.
- Illegal Request. (Status 2, check condition; Sense 5, Illegal Request, ASC/Q = 0x0500, LUN not supported.) Logical units other than zero are not supported.

3.12.3.7 Initialization for Operation as a Target

(To be written.)

3.13 XNI-1 Network Adapter

The XNI-1 Network Adapter (XNI-1) provides a highly efficient connection to four independent networks.

The XNI-1 generates bus cycles of the types "Word Read", "Word Write", and "Interrupt"; that is, it is capable of reading and writing in main memory and interrupting the CPU.

The XNI-1 responds to bus cycles of the following types: "Device Status", "Device Control", "Word Read", and "Word Write". That is, the XNI-1 is controlled in part as a peripheral device and in part by reading and writing as if it is a memory. The description that follows is organized as an explanation of the I/O registers (those that respond to the "Device Control" and "Device Status" bus cycles), an explanation of the memory registers, and a description of the protocol of communication between the CPU (operating system) and the device.

In addition to the memory addressable from the CPU, the XNI-1 also contains a local on–board memory, called Memd, which consists of 262,144 16–bit words. This memory normally contains received datagrams, etc.

3.13.1 XNI-1 Network Adapter I/O Registers

3.13.1.1 Device Status

A Status Read Request directed to the XNI-1 at address 0 (`.NASTS==:0`) will elicit the response described below:

XNI-1 Network Adapter — Status Read from Address 0

1	0	0	*Subtype* 0 | 0 | 0 | 0 | 1	*x*	*Hardware Revision*	*Microcode Address*	*NEr*	*Rdy*
0	1	2	3 4 5 6 7	8 12	13 17	18 33	34	35

(*x* denotes a field not presently used.)

Bits 0–2 (`DS%TYP`) will contain the type code pattern 100 (`NA%TCP==:4`) to identify this as a Communications I/O device. The rest of the response is decoded as follows:

Subtype	Bits 3–7 (`DS%STY`) identify the subtype of network controller. Subtype 1 denotes the quadruple interface to the 10 MHz Ethernet. All other values of the subtype field are reserved for future interfaces.
Hardware Revision	(`NA%HRV==:37B17`) This field contains the hardware revision number of this device.
Microcode Address	(`NA%UAD==:177777B33`) The current value of the microcode program counter. This may be useful for diagnostic purposes.
NEr	(`NA%NER==:1B34`) No Error. When 0, this bit signifies that the device hardware has detected a microcode parity error.

Rdy (`NA%RDY==:1B35`) Ready. When set, the device is in its normal operating condition: it is ready to allow access to its control registers (as described below). After the software has seen “Ready”, unless the software has requested that the XNI-1 reset itself, a subsequent negation of “Ready” signifies that the XNI-1 microprocess is busy doing other work and that an attempt to access a XNI-1 control register is likely to result in a bus timeout.

A Status Read Request directed to the XNI-1 at address 1 (`.NABUS==:1`) will elicit the response described below:

XNI-1 Network Adapter — Status Read from Address 1

Q Slot	*Q Type*	*QA*	*B Slot*	*B Type*	*BA*	*RAc*	*RFl*	*RFr*	*BTO*	*DST 0 Slot*	*DST 0 Type*	*RTO0*	*DST 1 Slot*	*DST 1 Type*	*RTO1*
0 3	4 6	7	8 11	12 14	15	16	17	18	19	20 23	24 26	27	28 31	32 34	35

This status word reflects the state of the XNI-1’s activities regarding the backplane bus. The status word is decoded as follows:

Q Slot (`NA%QSL==:17B3`) Queued slot number. This field contains the device slot number to which the current head of the queued requests is directed. The XNI-1 keeps a queue of memory requests that haven’t yet been made. Valid only when *QA* is 1, this field and the *Q Type* field reflect the state of the queue. When the XNI-1’s bus operation unit (described below, starting with *B Slot*) becomes available, the head of the queued requests becomes the next active transfer.

Q Type (`NA%QTY==:7B6`) Queued operation type. This three–bit field, valid only when *QA* is 1, defines the type of operation presently at the head of the queue. The field is decoded as follows:

000	None, or status return.
001	Read return.
010	DMA read.
011	DMA write.
100	ALU read.
101	ALU write.
110	Interrupt.
111	ALU read return.

QA (`NA%QAC==:1B7`) Queue Active. If 1, the queue is non–empty and the *Q Slot* and *Q Type* fields listed above are valid. If 0, the queue is empty; the fields listed above are stale.

B Slot (`NA%BSL==:17B11`) Bus transfer slot number. The device slot number to which the current operation is directed. Valid only when *BA* is 1, this field and the *B Type* field describe the state of the XNI-1's current bus operation.

B Type (`NA%BTY==:7B14`) Bus transfer operation type. This three–bit field, valid only when *BA* is 1, defines the type of operation presently at the head of the queue. The field is decoded as shown above under *Q Type*

BA (`NA%BAC==:1B15`) Bus transfer active. If 1, the bus transfer buffer is active: the *B Slot* and *B Type* fields are valid. If 0, the transfer buffer is empty and the fields listed above are stale.

RAc (`NA%RAC==:1B16`) Read–active. This flag is 1 to signify that the XNI-1 is waiting for one or two read responses. Up to two read requests may be outstanding at any instant. This flag being set indicates that the XNI-1 is waiting for the response to at least one outstanding read request. If this flag is 0, the device has no read requests outstanding.

The following fields and flags are meaningful only when read–active is 1: *RFl* (Read–full), *RFr* (Read–first), *DST 0 slot*, *DST 0 type*, *DST 1 slot*, and *DST 1 type*; these relate to the bus interface logic within the XNI-1.

RFl (`NA%RFL==:1B17`) Read–full. The XNI-1 has two read requests outstanding. A read response must be received before the XNI-1 can issue another read request.

RFr (`NA%RFR==:1B18`) Read–first. If 0, the XNI-1 internal unit that issued the DST 0 request gets the first response from the device whose slot number matches the DST 0 slot. If 1, the unit that issued the DST 1 request gets the first response from the device whose slot number matches the DST 1 slot. The read–first flag resolves the ambiguity of which read unit gets the next response when both are waiting for the same device.

When read–full is 0 and read–active is 1, only one read request is outstanding. In this case, read–first identifies which of DST 0 type or DST 1 type specifies the requesting unit.

When read–full is 1 and the values of DST 0 slot and DST 1 slot are different, the memory data will be sent to the read unit whose DST n slot value matches the slot number of the responding memory.

BTO (`NA%BTO==:1B19`) Busy time–out. The target was busy on 256 consecutive attempts.

DST 0 Slot (`NA%D0S==:17B23`) This field specifies the backplane slot number of the memory unit to which a read request was directed by the XNI-1 internal unit identified by DST 0 Type. This data is valid if read–full is 1 or if read–active is 1 and read–first is 0.

DST 0 Type (`NA%D0T==:7B26`) The type field decodes as *Q Type*, above, but the only valid values are 2 and 4. This data is valid in the same circumstances as when the DST 0 slot is valid.

RTO0 (`NA%RTO==:1B27`) Read time–out 0: No data was returned during the 256 bus cycles that followed the target's acceptance of a request.

DST 1 Slot (`NA%D1S==:17B31`) This field specifies the backplane slot number of the memory unit to which a read request was directed by the XNI-1 internal unit identified by DST 1 type. This data is valid if read–full is 1 or if read–active is 1 and read–first is 1.

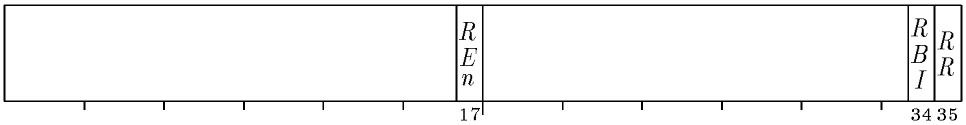
DST 1 Type (`NA%D1T==:7B34`) The type field decodes as *Q Type*, above, but the only valid values are 2 and 4. This data is valid in the same circumstances as when the DST 1 slot is valid.

RTO1 (`NA%RT1==:1B35`) Read time–out 1: No data was returned during the 256 bus cycles that followed the target's acceptance of a request.

3.13.1.2 Device Control

A device control bus cycle to the XNI-1 at any address reloads the microcode and, optionally, performs the reset function described below.

XNI-1 Network Adapter — Device Control to Address 0



The fields have the following significance:

REn (`NA%REN==:1B17`) Reset Enable. If this bit is 1, the XNI-1 is enabled to look at the other bits and to perform the requested action(s).

RBI (`NA%RBI==:1B34`) Reset Bus Interface. If this bit and `NA%REN` are both 1, the XNI-1 will reset its bus interface. The bus interface might possibly become hung if an incorrectly formatted Message Control Block (MCB) is presented to the XNI-1. (The MCB is a data structure used to pass information between the XNI-1 and the CPU; it is explained further in §3.13.3.) For example, if the MCB specifies a bus address word with an incorrect slot number, the XNI-1 may get a bus timeout every time it attempts to transfer data at that address; because the failing transfer will be retried, the bus interface may remain stuck for a while. This command clears it.

RR (`NA%RR==:1B35`) Reset Request. If this bit and `NA%REN` are both 1, the XNI-1 Network Adapter will reset itself to its power–on condition. This operation causes the XNI-1 to be "busy" and unresponsive for approximately 0.1 seconds. The reset operation destroys all MCBs held in the data registers. (Unless the operating system has an independent method by which it remembers where the MCB buffers have been placed, the system will lose the use of all memory allocated for input messages and current output messages.)

Since all device control cycles reload the microcode, it is recommended that only the combination `NA%REN!NNA%RR` be used to insure that the microcode gets a consistent hardware state after it is reloaded.

3.13.2 XNI-1 Network Adapter Memory Registers

The XNI-1 contains 8,192 36–bit words of addressable memory registers. These are divided into three categories: control registers, data registers, and packet snoop registers. The XNI-1's operating microcode allocates addresses 0–37 to the control registers, 40–7777 to the data registers, and 10000–17777 to the packet snoop registers. Access to any register is permitted whenever the register set is not "busy." The distinction among these register types is that access to any control register or to any packet snoop register causes the entire register set to become busy for a period of time during which no access to any register is permitted; access to a data register does not cause the register set to become busy. (Attempts to access a register while the register set is busy will result in a "busy" response and a possible bus timeout. The register set is typically busy for a period of 1–2 microseconds following access to a control register.) The control and data registers may be read or written; the packet snoop registers are read–only.

3.13.2.1 XNI-1 Control Register Addresses

The following control register addresses are of particular importance in programming the XNI-1:

00 Slot number of the XNI-1 (`NA%SLT==:0`): This register is written to by the CPU to tell the XNI-1 its own backplane address. The XNI-1 slot number will be written in bus address word format (§3.1.4); i.e., in bits 3–6 of the control register. The unused portions of the word will be written as zero. (The contents of this register are used by the XNI-1 hardware to determine whether a bus transaction is directed at it)

01 Priority Interrupt Assignment Register (`NA%PIR==:1`): This register holds the priority interrupt assignment level for the XNI-1 and the slot number of the processor to which the XNI-1 shall direct interrupts. This register may be written in to set the priority interrupt level and assign a processor to field the interrupts (typically, the processor writing this register will send its own slot number). The processor slot number will be written in bus address word format in bits 3–6 of this register; the priority level will be written in bits 33–35. This register may be read to discover what priority level has been assigned. The default priority level for the XNI-1 is 5.

02 Command Register (`NA%CMR==:2`). This register may be written to by the CPU to cause the XNI-1 to perform one of the commands that do not involve the memory transfers characteristic of most data transfers. Results of a command, if any, are available through the results register. Commands are specified in bits 28–35 of this register; the other bits may be used as parameters. Commands and results are described in §3.13.4.

03 Unused (`NA%CRR==:3`).

04 Result Register (`NA%RSR==:4`). If a command produces results, they are recorded in the Result Register. The meaning of the value in the the Result Register is dependant on the command that was issued. Commands and results are described in §3.13.4.

05 FromXR (`NA%FXR==:5`) or Transmit Done register: when the XNI-1 has one or more MCBs of messages that have been transmitted, it stores the address of the MCB (or the head of the list of MCBs) in this register, and it requests an interrupt at the assigned priority level. If the register already contains a non–zero value, the XNI-1 accumulates completed transmission MCBs until the register becomes available. When the CPU reads this register, the XNI-1 withdraws its interrupt request, but the register remains busy until the CPU writes the value zero into it. (There is no need for the CPU to send the XNI-1 a message to say that the register has become free.)

06 FromRCV (`NA%RCV==:6`) or Receive Data register: when the XNI-1 has one or more MCBs representing messages received, it will store the address of the MCB (or the head of the list of MCBs) in this register and it will request an interrupt at the assigned priority level. If

the register already contains a non–zero value, the XNI-1 accumulates completed transmission MCBs until the register becomes free. When the CPU reads this register, the XNI-1 withdraws its interrupt request, but the register remains busy until the CPU writes the value zero into it. (There is no need for the CPU to send the XNI-1 a message to say that the register has become free.)

07	Unused.
10	ToPr0 (`NA%TP0==:10`) or Transmit Port 0 register: when the CPU wants to transmit a datagram on the XNI-1's output port 0, it stores the address of a list of MCBs (all for port 0) in this register, provided the register already contains the value zero. (If the register contains a non–zero value, the CPU will accumulate a list of MCBs addressed to port 0 and waiting to be sent to the XNI-1.) There is no need for the CPU to send a message to the XNI-1 to tell it to look at the new value.
11	ToPr1 (`NA%TP1==:11`) or Transmit Port 1 register. This register is the port 1 analog of `NA%TP0` register.
12	ToPr2 (`NA%TP2==:12`) or Transmit Port 2 register. This register is the port 2 analog of `NA%TP0` register.
13	ToPr3 (`NA%TP3==:13`) or Transmit Port 3 register. This register is the port 3 analog of `NA%TP0` register.
14	ToRet (`NA%TRT==:14`): when the CPU has a receive MCB for the XNI-1, it stores the address of that MCB in this register, provided the register contents are already zero.
15	ToRef (`NA%TRD==:15`): when the CPU has an MCB describing a received message that must be reformatted by the the XNI-1, it places the address of the MCB in this register, provided the register contents are already zero.
16	ToMcm (`NA%MCM==:16`): this register is written by the CPU (assuming the register is nonzero). The address of a MCB is written into this register. Currently there is no defined processing by the XNI-1 microcode. This register and FrRmc are reserved for future out–of–band communication between the XNI-1 microcode and the CPU.
17	FrRmc (`NA%RMC==:17`): this register is written by the XNI-1 microcode, to return the MCB that was processed in response to the CPU writing into ToMcm. Currently this register is not used.
20–37	Unused.

3.13.2.2 XNI-1 Data Register Addresses

The data registers are in addresses 40–7777. The region 40–177 is reserved by XNI-1 microcode for Result Blocks and for identification data.

The identification data occupies fixed locations starting at `.NABSB==:102`. The addresses in the identification region are given symbolic names that are offsets from `.NABSB`, as follows:

`.NAUVR==:0`	This location contains the microcode version number.

`.NASN==:1`	Bits 12–35 of this location contain the serial number of this XNI-1 interface board.
`.NAPN0==:2`	This location and `.NAPN1==:3` contain the 48–bit IEEE MAC (media access control) address for the port 0 interface. The MAC address for ports 1, 2, and 3 are precisely 1, 2, and 3 higher, respectively. Bits 20–35 of `.NAPN0` contain the high–order 16 bits of the MAC address, and bits 4–35 of `.NAPN1` contain the low–order 32 bits of the MAC address.

The remaining data registers are divided into Message Control Blocks, as described below.

3.13.2.3 XNI-1 Packet Snoop Register Addresses

The packet snoop registers are virtual registers: the XNI-1 internal microcode responds to read requests addressed in the range 10000–17777 by supplying information from its internal packet buffer memory. This area is currently not used.

3.13.3 Communication Between the CPU and the XNI-1

The usual form of communication between the operating system and device involves a data structure called a Message Control Block (MCB). MCBs are contained within the Data Register memory of the XNI-1.

The XNI-1 microcode creates MCBs in response to a command from the CPU. This list of available/free MCBs is kept on–board (in the XNI-1 microcode) and the CPU can request free MCBs by issuing a command to XNI-1 microcode.

During system startup, the CPU system code requests 10 free MCBs for each interface that has been configured. The 10 MCBs are associated with system memory; the system software then returns the list of initialized input MCBs to the XNI-1 via the "ToRet" control register.[40] The XNI-1 retains the input MCBs until messages arrive.

When a message is received from the network, the XNI-1 removes an input MCB from its list of idle input MCBs and, using the data within the MCB, which describes a region in main memory, stores the input message in main memory. The XNI-1 then gives the MCB to the CPU by storing the MCB's address in the "FromRCV" control register and requesting an interrupt. Eventually, the operating system will process the incoming message; then it returns the MCB to the XNI-1 by storing the address of the MCB in the "ToRet" control register.

For messages to be output, the operating system uses an output MCB taken from its list (the typical list of output MCBs is one). It fills in the in–memory address, the size of the output message buffer, and format control. The operating system then passes the address of the MCB in the "ToPr*n*" control register, where *n* corresponds to the port number that is to be used for transmitting the message. After the XNI-1 Network Adapter has copied the transmitted data into its local memory, it returns the MCB in the "FromXR" control register, and requests an interrupt.

As described earlier in this document, if a "To*xxx*" register is non–zero when the operating system software wants to write to it, the system builds a list of pending messages until the register becomes

---

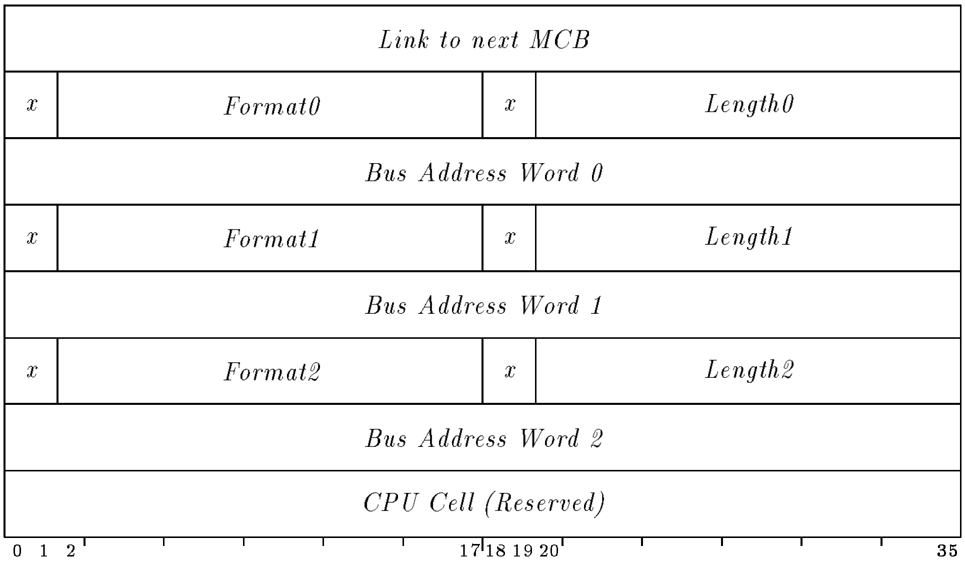
[40] The action of reading or writing a control register is noticed by the XNI-1 microprocessor, so no other action is needed to alert the XNI-1 to incoming messages.

zero.[41] Similarly, when the XNI-1 wants to store an MCB address in a "From*xxx*" register, it must wait until that register is zero; while waiting, it collects pending messages (MCBs) in a list and stores the list head in "From*xxx*" when it is able to do so.

3.13.3.1 Message Control Block

The format of a Message Control Block is dependent on its use. The format of a transmit Message Control Block is depicted below in Figure 3.8:

Figure 3.8: Transmit Message Control Block Format



(*x* denotes fields not presently used.)

The meaning of the fields in the transmit Message Control Block is as follows:

Link	This is a link to the next MCB on the list. Zero means no further items on the list.
Format0	The format field describes the mode of transfer that is to be performed on the data. The format field is broken down into

3b17	Two bits of information to describe the byte offset to start data transfer from buffer 0 memory location i.e. 0 would start on byte 1; 1 would start on byte 2 etc.
1b15	One bit of information to the describe the byte offset into the XNI-1 memory to start storing the buffer 0 data. This bit is always set to 0.
7b14	Three bits of information to describe how the data is stored in buffer 0 memory. This field can have the following values:

0 16 bit mode.

---

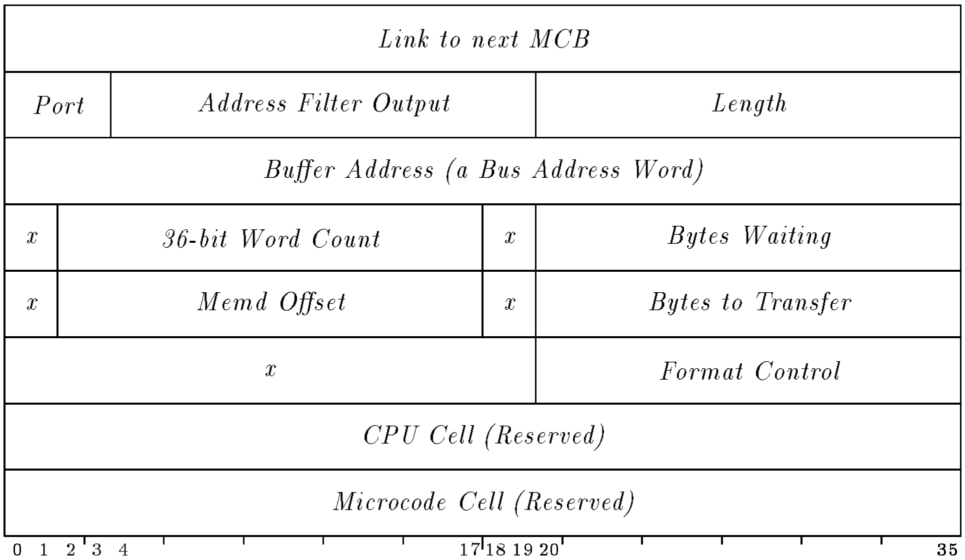
[41] The XNI-1 may be fast enough so that a CPU will not ever find a "To*xxx*" register busy.

1	32 bit mode.
2	36 bit mode.
3	35 bit mode.
4	16 bit byte swappped.
5	9 bit mode.

1b11	The start bit. This bit **must** be set for the transfer to begin.
1b3	The LOOPBACK bit, set to indicate loopback mode. If this bit is set the XNI-1 microcode will not transfer the data onto the wire. The transmit data will be be looped back within the XNI-1 and the CPU will see the datagram as a receive datagram.
1b2	The MORE bit. When set this indicates that at least one more buffer is present in this transmit MCB. If this bit is 0; the entire transmit datagram is described by Format0 and Length0 and is contained in the buffer 0 location.

Length0	The Length0 field (177777b35) contains the number of 8–bit bytes to be transferred from the buffer 0 location into the XNI-1's on–board memory under the control of Format0.
Bus Address Word 0	This is a bus address word that defines the location of the buffer0 area assigned to this MCB. The XNI-1 accepts only bus address words in which D, the device bit, is zero. This field is set by the operating system.
Format1	This field is used by the XNI-1 only if the MORE bit is set in the Format0 field. The field definition is identical to the description of Format0 except that they relate to the *Bus Address Word 1* and *Length1* fields.
Length1	This field contains the number of 8–bit bytes to be transferred from the buffer 1 location into the XNI-1's on–board memory under the control of Format1.
Bus Address Word 1	This is a bus address word that defines the location of the buffer 1 area assigned to this MCB. The XNI-1 accepts only bus address words in which D, the device bit, is zero. This field is set by the operating system.
Format2	This field is used by the XNI-1 only if the MORE bit is set in the Format1 field. The field definition is identical to the description of Format1 except that they relate to the *Bus Address Word 2* and *Length2* fields.
Length2	This field contains the number of 8–bit bytes to be transferred from the buffer 2 location into the XNI-1's on–board memory under the control of Format2.
Bus Address Word 2	This is a bus address word that defines the location of the buffer 2 area assigned to this MCB. The XNI-1 accepts only bus address words in which D, the device bit, is zero. This field is set by the operating system.
CPU Cell reserved	This cell is used by the system software to correlate a transmit MCBs with its associated system resources (caller, virtual addresses of the transmitted datagram etc); it is not used by the XNI-1 microcode.

Figure 3.9: Receive Message Control Block Format



(x denotes fields not presently used.)

The format of a receive Message Control Block is depicted in Figure 3.9. The data fields are as follows:

Link	This is a link to the next MCB on the list. Zero means no further items on the list.
Port	The port number of the particular ethernet interface that this message was received on. For the XNI-1 the legal port numbers are in the range 0–3. This field is set by the CPU when it contructs the receive MCB.
Address Filter Output	To assist in the classification of received datagrams, the XNI-1 microcode delivers the status of the address filter for this datagram. The system software controls the setting of the address filter and the possible values that can be delivered by the XNI microcode.
Data Length	The length of the received datagram, in 8–bit bytes. The XNI-1 stores this count into the MCB before delivering the receive MCB to the system software (see description of "FromRCV").
Buffer Address	A bus address word, specifying the location where to store the receive datagram. This value is set up by the system software before it writes the MCB address into the "ToRed".
36–bit Word Count	Set by the system software. This field contains the number of system memory locations that can be written into by the XNI-1 microcode. The XNI-1 microcode uses this value to determine how many locations can be written into, starting at the Buffer Address location. If this value is zero (0), the entire received datagram can be written into system memory by the XNI-1 starting at the address specified by the Buffer Address.

Bytes Waiting	This value is set by the XNI-1 microcode when a datagram has only been partially transferred into system memory. This occurs when the 36–bit Word Count is not sufficient to allow the XNI-1 hardware/microcode to transfer the entire receive datagram into system memory. This count tells the system software how many 8–bit bytes are still waiting to be transferred into system memory.
Memd Offset	This value is valid only if the Bytes Waiting count is non–zero. On a partially received datagram, the entire receive datagram still resides in the XNI-1's Memd. The Memd Offset counts how many 16–bit items from Memd were transferred to system memory; that is, it describes the offset to find the next bytes of the received datagram. It is used by the XNI-1 microcode to remember where to pickup re–formatting the receive datagram. A zero offset indicates the beginning of the datagram. This value could be changed by the system software, but it is not changed by the current system software.
Bytes to Transfer	This value is set to CPU software after it has received a partial receive datagram, i.e. Bytes Waiting is non–zero. When the receive MCB is given back to the XNI-1, this value tells the microcode how many bytes to transfer into the new Receive Buffer Address.
Format Control	This value is set by the CPU software after it has received a partial receive datagram. It controls the mode of transfer that the XNI-1 hardware will perform on the remainder of this datagram. The format of this field is identical to the Format field described in the transmit MCB section.
CPU Cell	This cell is not used by the XNI-1. It is reserved for use by system software to associate data structures with this MCB.
Microcode Cell	This cell is reserved for the XNI-1 microcode, which uses it to associate an MCB with a microcode data structure (Memd address).

3.13.4 Commands and Result Blocks

A command and response mechanism is implemented for transactions, such as status inquiries, which do not require large amounts of data transfer. This mechanism has been partially described in the discussion of the control registers, above[42].

Command codes are placed in bits 28–35 of the command register; bits 0–27 of the command register are reserved for arguments, if any. The following commands are defined:

001 Not Used

002 Create MCBs.

The system software controls the carving up of XNI-1 memory for use of MCBs. This command contains one argument: the size of each MCB. The maximum size is of limited by the width of the argument field. In this command, the argument field is 377b27. If the CPU software attempts to re–issue this command, the XNI-1 will return the current number of available MCBs, i.e. it remembers that it has already created MCBs and will not process the command.

---

[42] This explanation will have to be expanded adapted for multi–processor systems.

The response register will contain the number of MCBs that the XNI-1 microcode has created or has presently available.

003 Request free MCBs.

This command is issued by the system software to the XNI-1 microcode to obtain some free MCBs. This command contains one argument (377b27 i.e. limited to 255): the number of free MCBs requested. If the CPU software requests more MCBs than the XNI-1 microcode has available, then 1b2 will be set in the command register and the number available will be returned in 77777b17 of the command register. If the XNI-1 microcode has sufficent MCBs to satisfy the request, the linked list of MCBs will be returned in the response register.

004 Select Interface.

This command is issued by the system software to select a specific port on the XNI-1, because some commands require that an interface be selected. This command contains one argument (7b27): the port to be selected. The XNI-1 will return the following values in the response register; −1 (177777) if the interface is not present; zero (0) if the interface was successfully selected.

005 Disable Interface.

This command causes the XNI-1 microcode to disable the specified port on the XNI-1. This command contains one argument (7b27): the port to be disabled. The response register will contain −1 (177777) if the interface does not exist; one (1) if the interface is currently disabled and zero (0) if the XNI-1 microcode has successfully disabled the interface. Disabling an interface turns off the reception hardware in the XNI-1, as a result no further datagrams can be received until the interface is enabled.

006 Enable Interface.

This command causes the XNI-1 microcode to enable reception on a specific port on the XNI-1. This command contains one argument (3b27): the port to be enabled. Before an interface can be enabled, it must have receive MCBs assigned to it (see `NA%TRT`). The response register will contain −1 (177777) if the interface does not exist; 0x8001 if the interface has no receive MCBs and zero (0) if the interface has successfully been enabled.

007 Address Filter Size.

This command requests the XNI-1 microcode to return the size of the address filter for the specified port. This command contains one argument (3b27): the port number. The response register will contain the number of octets in the address filter. The current XNI-1 hardware has an address filter size of 32 bytes. This size allows for address and protocol recognition on the first 32 bytes of the received datagram.

008 Set protocol mask.

This command sets the protocol mask for the output of the address filter. The system software sets up the address filter (allows the reception of specific MAC address and protocol type). This mask is used by the XNI-1 microcode to determine the portion of the address filter output that is protocol specific. This command contains one argument (377b27): the protocol mask. The Select Interface command must have be issued prior to the issuance of this command. The default protocol mask is 37.

009 Set Protocol Assembly Mode.

This command sets the default assembly (input) mode for a specific protocol on a specific port. The assembly mode refers to the data translation for data being transferred to system memory. The protocol assembly mode bits controls the format under which data is transferred into system memory. The default protocol assembly mode is 0x004a (byte offset into system memory, two; byte offset into XNI-1 Memd, zero; 32–bit data mode; start bit on. (See the description of *Format0* in the transmit MCB). The assembly mode is 16–bits wide; if the 0x8000 bit is on, the XNI-1 microcode will verify the IP checksum in the received datagram. If the 0x7f80 bits are non–zero, then the XNI-1 microcode treats that field as a byte count and transfers the number of bytes into system memory.

010 Return Free MCBs.

This command returns an MCB, or a list of MCBs, to the free pool in the XNI-1 microcode. The command contains one argument (177777b17): the address of the first MCB (possibly linked to additional MCBs).

011 Write Address Filter.

This command is issued to the XNI-1 microcode to write values into the address filter. This command takes one argument: the address of a XNI-1 memory block[43] that contains the following information:

offset 0 177777b35 the port number, one of 0–3.

offset 1 177777b35 byte offset into the address filter (an even number).

offset 2 377b17 the value to write (even byte).

offset 2 377b35 the value to write (odd byte).

There are four address filters, one for each port.

Each address filter is logically a 32 by 256 byte array, where the first index corresponds to a byte number (0 to 31) in a message header, and the second index corresponds to the data value of that header byte. The address filter accepts and categorizes incoming messages by computing the logical AND of the data bytes fetched from this array. If, while processing the message header, the AND becomes zero, the message is discarded because it is not addressed to this system. If the header is processed with the AND being non–zero, the resulting value is passed to the operating software to assist it in categorizing the type of message and protocol.

An address filter is implemented as an array of 4096 16–bit words. Each word contains two consecutive data bytes of the address filter array (one byte for an even–valued header byte, and one for the next higher value). The first 128 words (256 bytes) are addressed for header byte 0; the next 128 words are addressed for header byte 1, etc.

It is the reponsibility of the operating system to fill the address filter with appropriate values.

012 Read Interface Errors.

This command will cause the XNI-1 microcode to return the current error counts for the selected port. Reading the interface error counts causes the microcode to zero out its running count, so the system software should maintain a running total. This command takes one argument (7b27): the interface or port number 0..3. The XNI-1 returns the address of a result block which contains the error counts.

The following error counts are accumulated for the interface by the XNI-1 microcode.

---

[43] The memory block is usually obtained as an MCB, and then diverted to this use.

offset 0	Jabber Error Count. This count indicates the number of times that the hardware attempted to transmit for an excessive time period (20–150ms).
offset 1	Babble Error Count. This counts the transmitter time–out errors. It indicates the number of times the transmitter has been on the channel longer than the time required to send the maximum packet. It is set after 1519 bytes (or greater) have been transmitted.
offset 2	Collision Error Count. Each count indicates the absence of the Signal Quality Error Test (SQE Test) message after a packet transmission.
offset 3	Receive collision count. Receive collisions are defined as receive frames which suffered a collision.
offset 4	Runt Packet Count. This is the count of runt packets addressed to our station. Runt packets are those in which the address filter indicates that we should receive this datagram, but the datagram is less than minimum allowed on ethernet (64 bytes).
offset 5	Missed Packet Count. Each count indicates an instance when the receive FIFO in the hardware overflowed. This is caused by the XNI-1 microcode being unable to service the Ethernet receive interrupt in a timely manner.
offset 6	Overflow Flag Count. This counts the number of times that the Receive FIFO overflowed due to the inability of the XNI-1 microcode to read data fast enough to keep pace with the receive serial bit stream and the latency provided by the Receive FIFO itself.
offset 7	Framing Error Count: Each count indicates an instance when the received frame contained a non–integer multiple of bytes and an FCS error.
offset 8	FCS Error Count; indicates that there is an FCS error in the frame.

013 Read Address Filter

This command will read the contents of the address filter for a specific port number. This command takes one argument: the address of a block of XNI-1 memory which the operating initializes to contains the port number and the byte offset to the address filter. Address filter reads always return two bytes: the data at the byte offset requested, and the subsequent byte.

The XNI-1 microcode will read the address filter and return data at offset 2:

offset 0	177777b35 the port number 0..3 for which we want to read the address filter.
offset 1	177777b35 byte offset into the address filter (the address to read, an even number).
offset 2	The two bytes will be returned at 377b17 and 377b35.

Chapter 4

Earlier Processors (Omitted)

Printed versions of this manual omit Chapter 4, Earlier Processors.

Chapter 4

Earlier Processors (Omitted)

Printed versions of this manual omit Chapter 4, Earlier Processors.

Chapter 4

Earlier Processors

Editors note: The information in this chapter is from the July 1980 edition of *DECsystem–10 DECSYSTEM–20 Processor Reference Manual*, published by Digital Equipment Corporation. The reader is cautioned that subsequent developments by Digital Equipment Corporation have rendered portions of this material obsolete or incomplete. For example, the material on TOPS–10 paging is largely obsolete, as TOPS–10 was enhanced to use TOPS–20 paging.

4.1 KL10 System Operations

The information presented in this section is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However, it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own I/O, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

WARNING

> KL10 functions are implemented in microcode, which can be revised much more easily than hardware. Although the user operations described in Chapter 2 are deliberately kept as compatible as possible from one machine to the next, Digital will change KL10 system microcode whenever such change will result in greater speed, efficiency, or effectiveness. Therefore, anyone writing system software should make sure to use the most recently updated version of this documentation, and before embarking on any project as enormous and critical as an operating system be sure to check with Large Systems Engineering for any changes not yet documented.

Programming for the system as a whole is programming in executive mode. Only the kernal program is without instruction restrictions, and only it can, if needed, access physical memory unpaged. The supervisor program labors under the same instruction restrictions as the user and has no way of bypassing them, although it can read but not alter concealed pages (the kernal program can supply data tables to the supervisor program, and latter cannot affect them).

The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working sets, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of a user. The kernal program accomplishes these objectives by handling all in–out for the system, setting up user page maps, trap locations, interrupt locations, and the like for both itself and the users, handling user accounts, communicating with the front end, and so forth. In other words, except for handling in–out, the activities of an operating system are the topics covered in this section. The amount of useful work done by the system depends on how efficiently and effectively the executive manages the physical resources of the system. These resources include the processor, memory, input–output devices, the file system, and the bandwidth of the paths between various components. The executive selects which process to run next. It manages the working sets of the various processes, responding to their changing needs. The executive reacts to error situations and even to unacceptable behavior on the part of a user. The executive accomplishes these objectives by handling all in–out for the system, setting up user page maps, trap locations, interrupt locations, etc. for itself and for the users. The executive handles user accounts, passwords, and level of privileges. It controls access to all system resources.

The activities of an operating system, particularly as they are implemented in the TOAD-1 System, are the topics of this chapter. Of course the system programmer must also be quite familiar with all of the material presented in chapters 1 and 2. In particular, the programmer must understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the JRST, MUUOs, and I/O instructions (§2.9.4, §2.16, §2.18).

System information for other processors is given in latter sections of this chapter, §4.2 and §4.3. The present section is devoted solely to the KL10, but contains two subsections on paging, only one of which is applicable to a given system. §4.1.3 describes the paging used with the TOPS–10 Monitor; this paging is similar to that of the KI10. §4.1.4 treats the paging associated with the TOPS–20 Monitor. Both kinds of paging employ essentially the same hardware — the difference lies principally in the microcode.

Much of the material presented here is related to the DTE20s, the channels, and the DIA20. Although the section does describe all activities of the microcode undertaken for these devices (e.g., the front end functions in §4.1.7), the descriptions of the devices themselves are not included.

CAUTION

> All IO instructions in this chapter are for internal devices (E bus functions). An address given by such an instruction for storing a result is always interpreted as global in the section containing the instruction. Hence, data or conditions cannot be stored in an AC unless the instruction is in section 0 or 1.

4.1.1 Priority Interrupt

The DECSYSTEM–20 is essentially a system of processors clustered around the E bus. The various controllers and interfaces are subsidary to the PDP–10, but maintain a considerable degree of independence from it. Each RH20 Massbus controller operates from its own command list in memory and handles all data transfers via the channels; but it must reach the Ten program to start a new list of if something should go wrong. Each PDP–11 is a whole computer with its own internal program;

but for handling I/O equipment or acting as the system console, it must communicate with Ten memory via the E bus (to which it is interfaced by a DTE20), and the peripheral computer must reach the Ten program for setting up mutual operations. Basically, the priority interrupt system allows the other processors to interrupt the central processor at various levels of priority, so that all can operate simultaneously. The hardware also allows conditions internal to the PDP–10 to signal its own program by requesting an interrupt.

In a DECsystem–10, the PDP–11 is limited to use as a system console and diagnostic facility, and the unit–record peripheral equipment is organized around a KI10–type I/O bus connected to the E bus via a DIA20 I/O bus interface. If the system lacks internal channels, Massbus controllers must be of the RH10 type, which the program controls via the I/O bus. For data purposes an RH10 is connected to external memory by a separate memory bus. It is recommended that those who program a DECsystem–10 read both this section and the first few pages of the discussion of the KI10 interrupt[1] (§4.3.2).

Interrupt Requests

Interrupt requests are handled on eight levels arranged in priority sequence. Levels are numbered 0–7, with 0 having highest priority. Level 0 is quite unlike the others, however, in that it is available only to the front end processors for simulating console functions and handling byte transfers. Moreover, level 0 is always active — it cannot be turned off even by inactivating the interrupt system. The program does control the enabling of level 0 in the DTE20s, but the master front end can even override that. Assignment of devices[2] to the remaining levels is entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35); a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be placed on the same level.

When a device requires service, it sends an interrupt request signal on its assigned level over the bus to the processor. A request is recognized by the processor if the level is active — meaning that both the interrupt system and the individual level[3] have been turned on. But the processor can accept no requests while it is processing a request or starting an interrupt at any level, or holding an interrupt on the same level or on a level with higher priority than those on which requests have been recognized (in other words, if the current program is a higher priority interrupt routine). The request signal remains on the bus however until turned off by an appropriate response from the processor: either given by the program (CONO, DATAO, or DATAI, depending on the device), or generated automatically by the hardware. Thus, if a request is not recognized or accepted when made, it will be when the necessary conditions are satisfied. A single level will even shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request.

The request signal is generally derived from a flag that is set by various conditions in the device. Often associated with these flags are enabling flags, where the setting of some device condition flag

---

[1] On the Ten side of the DIA20, the interrupt works as described here. But on the other side it acts more like the KI10 interrupt, with seven programmable levels, second–order priority determined by proximity to the DIA20, etc. Of course the processor activities and interrupt functions available are those of the KL10.

[2] As explained in §2.18, the program treats all E bus controllers, internal subsystems, and I/O bus peripherals as I/O devices. In other words, it monitors and controls them by means of I/O instructions using appropriate device codes. For a PDP–11, the device is the DTE20.

[3] Remember that level 0 is always active, even when the interrupt system is off. In other respects this discussion applies to all levels.

can request an interrupt on the assigned level only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device by CONO. For example, a device may have half a dozen flags to indicate various internal conditions that may require service by an interrupt; by setting up the associated enabling flags, the program can determine which conditions shall actually request interrupts in any given circumstances.

Processing a Request. The processor handles only one request at a time. When it is ready, it accepts the highest priority request currently recognized, provided that request is on a level higher than the current program (all levels are higher than a noninterrupt program). To process a request the hardware sends an interrupt service demand to the devices on the E bus to determine which ones are currently requesting an interrupt on the accepted level. Note that at this point the processor is accepting not an individual request, but rather a class of requests: namely all those being made on the same level. Should the bus be busy, the demand is sent as soon as it becomes available, taking precedence over any I/O instructions that may also be waiting (note that in this situation the program actually stops). From among the devices that respond to the demand on the accepted level, the processor selects the one of highest priority[4] according to this schedule:

Devices in Order of Decreasing Priority	*Physical Device Numbers*[5]
Interval Counter	
Other internal requests — processor error flags, program initiated requests	
Channels 0–7	0–7
DTE20s 0–3	10–13
DIA20 — i.e., any device on the I/O bus	17

If the device selected is internal, no further processing of the request is required. Otherwise the hardware sends a function demand to the selected device (by specifying its physical number along with the interrupt level), and the device responds by returning an interrupt function word. In either case, once all necessary information about the request has been gathered, the interrupt system waits for the interrupt to start. The microcode checks frequently for a waiting request, and upon discovering one departs from its normal routine to start an interrupt. At such time PC points to the interrupted instruction, so a correct return can later be made to the interrupted program.

Interrupt Functions and Instructions

The action taken by the microcode to start an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the executive process table are associated with each level, locations $40 + 2N$ and $41 + 2N$, where N is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 5. The processor starts a "standard" interrupt for level N by executing the instruction in the first interrupt location for the level, i.e., location $40 + 2N$. This type of interrupt is performed for a processor error or program–initiated request, for an external device whose function word specifies a standard interrupt, and also for an I/O bus device that returns no function word. The fixed locations however

---

[4] There are therefore two orders of priority associated with an interrupt: first the level, and then for all devices requesting interrupts simultaneously on the same level, physical device number. These physical numbers are not the device codes used in the I/O instructions; they are just for interrupt priority purposes and depend on position on the backplane (the RH20s are ordered opposite from the slot numbers).

[5] Physical numbers 14–16 are not used.

need not be used. The interrupt function word sent by the device may specify an equivalent interrupt using a pair of locations selected by the function word, or some other interrupt function entirely. The function word (which is stored in AC 3, block 7) has this format.

KL10 Interrupt Function Word

Addr Space	*Funct*	*Q*	*Device*	0	*Interrupt Address*
0 2	3 5	6	7 10	11 12	13 35

The microcode acts from a function word whether there is one or not; its absence is taken as a zero function. The DIA20 returns the word supplied over the I/O bus or simulates a zero word. Bits 7–10 identify the device by its physical number, but this is supplied by the interrupt hardware, not the device. The meanings of the other bits in the word are as follows.

0–2 Address space. In unrestricted examine and deposit functions, codes given in these bits select the space in which the address supplied in bits 13–35 is interpreted.

- 0 Executive process table
- 1 Executive virtual address space
- 4 Physical address space

Remaining codes are reserved.

3–6 Interrupt function (bits 3–5), sometimes qualified by Q (bit 6). When unspecified, Q is irrelevant. The microcode handles functions 4–6 even when it is in the halt loop.

- 0 Internal device or zero word: for the interval counter perform a vector interrupt (see function 2); otherwise perform a standard interrupt (see function 1).
- 1 Standard interrupt — execute the instruction in location $40 + 2N$ of the executive process table.
- 2 Vector interrupt — action depends on device type as follows:

  Interval counter — execute the instruction in location 514 of the executive process table.

  DTE20 — execute the instruction in location 2 of the corresponding DTE20 control block.[6]

  Channel — execute the instruction in the executive process table location specified by bits 27–35.

  DIA20 — dispatch interrupt: execute the instruction in the executive virtual location specified by bits 13–35.
- 3 Increment — depending on whether Q is 0 or 1, add 1 to or subtract 1 from the contents of the executive virtual location specified by bits 13–35.
- 4 Examine — send the contents of the specified location to the selected DTE20. If Q is 0, select the location according to bits 0–2 and 13–35. If Q is 1, use bits 14–35 as a physical address and restrict the function to the communication area defined in the

---

[6] For further information on front end interrupt functions, refer to §4.1.7.

DTE20 control block.[6] The examine is effected by performing a DATAO to the DTE20.

5 Deposit — load the word supplied by the selected DTE20 into the specified location. If Q is 0, select the location according to bits 0–2 and 13–35. If Q is 1, use bits 14–35 as a physical address and restrict the function to the communication area defined in the DTE20 control block.[6] The deposit is effected by performing a DATAI to the DTE20.

6 Byte transfer — increment the byte pointer for the direction specified by Q (0 out, 1 in) from the control block for the selected DTE20, and then move a byte between Ten memory and the DTE20 according to the altered pointer.[6]

7 Reserved (result indeterminate).

CAUTION

Because of the special cycle in which it is executed, an interrupt function that uses virtual addressing cannot employ indirect pointers in its paging procedure (§4.1.4).

13–35 The bits among these that supply the address when the function requires one depend on the address space

Executive process table	27–35
Executive extended virtual address space	13–35
Executive unextended virtual address space	18–35
Physical address space	14–35

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode, and are therefore in executive virtual address space unless the particular function selects some other form of addressing. A page failure that occurs in an interrupt operation is never trapped; instead it sets the In–Out Page Failure flag, which requests an interrupt on the level assigned to the processor (§4.1.8). These considerations of course do not apply to a service routine called by an interrupt instruction.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being performed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in the first or second interrupt location for a level, in direct response by the hardware (rather than by the program) to a request on that level. These locations may be the fixed ones for a standard interrupt or those given by the function word for a vector interrupt. §2.18 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. There are two types of interrupt instructions executed in a standard or dispatch interrupt; the effects of all other instructions are undefined.

BLKI, BLKO. If the pointer count is not zero, the processor dismisses the interrupt and returns immediately to the interrupted program (i.e. it returns control to the unchanged

PC). If the count is zero, the processor executes the instruction contained in the second interrupt location.

XPCW, JSR. The processor holds an interrupt on the level, takes the next instruction from the location specified by the jump (as indicated by the newly changed PC), and enters either kernel mode or the mode specified by the new flag word of the XPCW. Hence the instruction is usually a jump to a service routine handled by the Monitor. XPCW is the preferred instruction on the extended KL10.

The most important point of which the programmer must be aware is that even while User is set, the interrupt instructions are not part of the user program. They are executed in kernel mode and are therefore subject only to kernel mode restrictions. Regardless of the current PC section, the address part of an interrupt instruction is interpreted as referencing section 0, except in a dispatch interrupt, where it references the section specified by the interrupt function word. As an interrupt instruction, JSR automatically clears both User and Public to jump to a kernel mode service routine. An XPCW should be set up to produce the same result. The XPCW control block must be in section 0 unless the interrupt is a dispatch.

CAUTION

Because of the special cycle in which an interrupt instruction is executed, the paging procedure for it cannot employ indirect pointers (§4.1.4).

Interrupt Programming

The program can control the priority interrupt system by means of condition I/O instructions. The device code is 004, mnemonic PI.[7]

CONO PI, Conditions Out, Priority Interrupt

70060	*I*	*X*	*Y*
0 12	13	14 17	18 35

Perform the functions specified by the effective conditions E as shown.[8] (A 1 in a bit produces the indicated function, a 0 has no effect.)

Write Even Parity				Drop Prgm Req On Lvls	Clear PI Sys–tem	Make Prgm Req On	Turn On	Turn Off	Turn Off	Turn On	Select Levels for Bits 22,24,25,26						
Addr	Data	Dir				Selected Levels			PI System		1	2	3	4	5	6	7
18	19	20		22	23	24	25	26	27	28	29	30	31	32	33	34	35

22 On levels selected by 1s in bits 29–35, turn off any interrupt requests made previously by the

---

[7] Data instructions with device code PI are unassigned and execute as MUUOs. The block instructions are used for error and diagnostic purposes (§4.1.8).

[8] Bits 18–20 are for test purposes only. They are used to force errors and are discussed in §4.1.8.

program (via bit 24).

23 Turn off the priority interrupt system, turn off all levels, drop all program–set requests, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by 1s in bits 29–35, and force the processor to recognize them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a CONO PI, that selects the same channel and has a 1 in bit 22.

Remember that the processor allows the program to continue while it processes a request. Thus when this bit forces recognition of a request, many additional program instructions may be performed before the interrupt, even on the highest priority level. Moreover if the request is allowed to remain, additional instructions may be performed between successive interrupts. For other than the highest priority level, the greater the number of higher levels active, the greater the amount of program time available both initially and between successive interrupts. If the program forces an interrupt on the lowest level when all are active, there can be a very long time between CONO PI, and its interrupt.

25 Turn on the levels selected by 1s in bits 29–35 so interrupt requests can be recognized on them.

26 Turn off the levels by 1s in bits 29–35, so interrupt requests cannot be recognized on them unless made by a CONO PI, with a 1 in bit 24.

27 Turn off the interrupt system so no requests can be recognized.

28 Turn on the interrupt system so the hardware can process requests.

CONI PI, Conditions In, Priority Interrupt

70064	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the priority interrupt (and several diagnostic bits) into location *E* as shown.

	Program Requests on Levels							W E P A	W E P D	W E P R	Interrupt Holding on Levels							PI On	Levels On						
	1	2	3	4	5	6	7				1	2	3	4	5	6	7		1	2	3	4	5	6	7
0	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; and 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on, and 1s in bits 29–35 therefore indicate active levels.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18–20 reflect several diagnostic functions discussed in §4.1.8.

Dismissing an Interrupt. Unless the interrupt operation dismisses the interrupt automatically, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt request can be accepted on that level or any of lower priority

A routine dismisses the interrupt by using an instruction that restores the level on which the interrupt is being held at the same time it returns to the interrupted program. The proper instruction is XJEN (JRST 7,) in an extended KL10, otherwise JEN (JRST 12,). Once the level is restored, the hardware can again accept requests and start interrupts on it and lower priority levels. These instructions also restore the flags: XJEN from the flag–PC doubleword if the routine was called by an XPCW; JEN from the left half of the PC word if the routine was called by a JSR in section 0. XJEN also restores the previous–context section if the return is being made to an executive program.

CAUTION

> An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its level and all levels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Timing. The maximum time a device may wait for an interrupt to start depends on how many active devices are of higher priority and how long their service routines are. When a given request is of highest priority, its device need never wait longer than 10 μs.

Special Considerations. When an interrupt occurs, PC points to the interrupted instruction (or to an XCT that executed it), unless the interrupt occurred in an overflow trap instruction, in which case PC points to the instruction that overflowed. After taking care of the interrupt, the processor can always return to the interrupted instruction. Either a) the instruction did not change anything; b) the interrupt was in the second part of a two–part instruction, where First Part Done being set prevents the processor from repeating any unwanted operations in the first part; or c) the interrupt occurred at some point in a multipart instruction where the microcode rigged the various pointers and other quantities so the processor actually restarts the instruction where it stopped, rather than from the beginning. However, in a BLT and in byte manipulation, the very mechanism that facilitates the return results in special properties of which the programmer must be aware.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two–part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a flag word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an ILDB or IDPB would skip a byte. And if the routine restored the flag, the interrupted ILDB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In–out flag is set, a user generally cannot reference the interrupt locations to set them up.

Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- Use interrupt instructions in a manner consistent with the special effects and conditions applicable to such instructions as described above.
- No request can be accepted, not even on higher priority levels, while a request is being processed or an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- To prevent a device from hanging up a level, the programmer must be aware of — and satisfy — whatever requirements the device has for dropping the request.
- The interrupt instruction that calls the routine should be an XPCW on an extended KL10, otherwise a JSR. In either case the paging for the instruction must not use indirect page pointers.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. Computations and any other time–consuming activities that can possibly be performed outside the routine should not be included within it.
- Never turn off the interrupt system in a routine unless it is absolutely necessary, and then always turn it back on again as soon as possible. If one or more levels can be turned off in place of the entire system, always do that instead.
- If the routine uses a UUO it must first save the contents of the locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type (§2.16).
- The routine must dismiss the interrupt (with an XJEN or JEN) when returning to the interrupted program. Flags and UUO locations should be restored.

4.1.2 Cache Management

For the user, the cache is transparent: any program simply gets information from memory and stores information in memory. But use of a cache as part of the memory subsystem reduces program time, since the cache is faster than the storage modules, and also reduces storage use by the program, making a larger percentage of total storage cycles available to other parts of the system. As explained in §1.2.2, transfers between processor and memory are in four–word groups: storage references are to four locations at a time.[9] The cache contains representations of a selection of such location groups. One may view the cache as 2048 general purpose registers, organized in sets of four, which substitute temporarily for the most frequently referenced physical storage location groups. The cache serves this function not only for the program, but for all microcode references, including those for handling interrupts, traps, page refills, and other automatic operations. The way the hardware handles the cache depends upon whether the initial processor reference to a location in a particular group is read or write.

---

[9] Of course memory control does not blindly request four storage cycles for every group even when it is known that some are unnecessary. Fewer references are made when some locations in a group already have valid representations in the cache, or the first or last transfer in a channel block is for part of a group.

When the first processor reference to a group is to read the contents of one of its locations, memory control retrieves the entire four–word group containing the referenced location. The single word requested is supplied to the program, but all four are placed in the cache and are validated, i.e. they are tagged as words that do represent the true contents of memory. Subsequent references, read or write, to the same group are made to the cache, not to storage. If the processor modifies the contents of a location in the group, the new word supplied is substituted for the one in the cache location, which is tagged as written. Thus the cache word is different from storage but still valid — i.e., it represents what the storage location should contain.

When the first reference to a group is for writing, there is no call to storage at all. Instead the hardware sets aside a location group in the cache, with the one word in it tagged as both valid and written. Further reads or writes of the same location are handled solely with the cache, and subsequent writes to other locations in the same group are handled just like the first. But a read to a location that has not been written produces a storage reference. The requested word is given to the processor, and all words in the group that do not already have written representations in the cache are inserted into the group entry.

When storage is being updated or a group entry that is not in use is replaced by another, words just valid can be thrown away. But written words must eventually be sent to a storage module.

Cache Structure. The 2048 locations in the cache are contained in 128 lines of sixteen each. The lines are identified by the possible group numbers in a single page, 0–177. Each line contains four group entries for the given number. Each group entry in turn comprises the number of the physical page[10] containing the storage group corresponding to the entry and representations of the four locations in the group, each with valid, written and parity bits.

The hardware also includes a mechanism for keeping track of the use of the various group entries. Whenever the processor references a group whose corresponding line in the cache already contains valid entries from four other pages, the hardware puts the new group representation in place of the least recently used entry in the line. But in doing so it also updates from any representations tagged as written in the displaced group entry.

Internal Channels. The channels are expected in general to deal with the storage modules, but if the cache contains any valid words for a page being handled through the channels, the hardware acts as follows:

> In an output operation, any valid representations at locations addressed by a channel are taken from the cache instead of storage.
>
> In an input operation, all data is sent to storage. However any entries that are in the cache for locations addressed by the channel are invalidated.

The reasons for this behavior are apparent. For output any valid words left in the cache might as well be taken since that is faster than going to storage. Furthermore some valid entries may have been written, and it is assumed that storage will certainly not be more up to date than the cache. Anything brought in via a channel is assumed to be the correct copy, and it should therefore go to storage as the page cannot be in use at the same time it is being loaded. Any valid entries left over in the cache must be from some previous operation, and they should therefore be invalidated, so any

---

[10] The list of all page numbers makes up the cache "directory." For many hardware functions the cache is organized in four quadrants. A quadrant contains 128 group entries, one from each line.

future references to those locations will go to storage for the correct copy. Should any of the valid leftovers be tagged as written, it is assumed the Monitor would have swapped out the modified page before bringing in the new. Of course a page used as temporary storage, or to hold counters and control words, albeit modified, can just be thrown away.

Cache Programming

The operations the program can perform on or for the cache are three: to invalidate, to validate, and to unload. Any of these operations may be carried out for all entries in the cache or for all entries of a single page. To invalidate a location is simply to clear its valid and written bits so it no longer represents anything. To validate or unload means to update storage, i.e. to write a cached word into storage if it is tagged as written, and to clear the written bit. Otherwise validating storage leaves the validity of the cache entries unchanged, whereas unloading invalidates all entries, written or not, in the groups being processed (all those in a single page or the entire cache).

Following power turnon in any system, the cache use tables must be initialized and the cache invalidated, as its initial state is indeterminate. Beyond this, a system with a single central processor and internal channels requires no cache programming, as everything is handled adequately by the hardware. However if a system contains facilities that bypass the processor to deal directly with external memory, whether such facility be an external channel or another central processor, then the Monitor must actually manage the relationship between storage modules and cache.

As an example of such management and to illustrate the difference in use between validation and unloading, consider the situation in which a program is through with the data in a particular (modified) page and it is to be swapped via an external channel with new data brought into the same physical page for later use. The page must be unloaded into storage so that subsequently the program will go there for the new data. On the other hand suppose a program has created some code in a page, and the system is both to go ahead and execute it immediately and place it in a library. Now validation is the proper procedure: while the storage copy is being filed, the program can continue execution from the cache.

For initialization and management, there is one instruction that initializes the use tables and six that sweep the cache to perform the above three operations for a single page or all pages. Note that a sweep of the entire cache is always necessary, even for handling a single page, as there is no prior way of knowing whether any given line contains a group from any given page. Sweeping for a single page does however take less time than sweeping for all pages. In the latter case the sweeper must check all 512 group entries, whereas the former requires checking only every line to see if it contains an entry for the specified page, and there can be at most one such entry. Moreover sweeping for all pages can usually be expected to require more storage references than sweeping for a single page. In this light it should be noted that the sweep instructions simply initiate operations which are then carried forward by the cache sweeper. The program can continue while the sweep is going on, but this can be expected to slow down the sweep as the cache and program would then compete for storage references. That a sweep is in progress is indicated by the Sweep Busy flag being on, and at completion the sweeper clears Busy and sets Sweep Done. The program can check both of these flags among what are otherwise the processor error conditions, and it can enable the latter to request an interrupt on the level assigned to the processor (§4.1.8).

These are I/O instructions wherein the cache sweeper has device code 014, mnemonic CCA. But the instructions have their own mnemonics since they bear no relation to the standard I/O operations. Six of the eight are used: the BLKI and CONO also sweep, doing nothing but wasting cache cycle time. The single instruction that initializes the use tables is discussed at the end of the section.

SWPIA Sweep Cache, Invalidate All Pages (DATAI CCA,)

70144	*I*	*X*	*Y*
0 12	13	14 17	18 35

E is not used.[11]

Set Sweep Busy, and clear the valid and written bits in all cache entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPIO Sweep Cache, Invalidate One Page (CONI CCA,)

70164	*I*	*X*	*Y*
0 12	13	14 17	18 35

Set Sweep Busy, and clear the valid and written bits in all cache entries for the physical page specified by bits 23–35 of *E*. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPVA Sweep Cache, Validate All Pages (BLKO CCA,)

70150	*I*	*X*	*Y*
0 12	13	14 17	18 35

E is not used.[11]

Set Sweep Busy, and write into storage all cached words whose written bits are set. Clear all written bits but do not change the validity of any entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPVO Sweep Cache, Validate One Page (CONSZ CCA,)

70170	*I*	*X*	*Y*
0 12	13	14 17	18 35

Set Sweep Busy, and write into storage all cached words whose written bits are set and which are found in entries for the physical page specified by bits 23–35 of *E*. Clear the written bits associated with those words sent to storage, but do not change the validity of any entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPUA Sweep Cache, Unload All Pages (DATAO CCA,)

70154	*I*	*X*	*Y*
0 12	13	14 17	18 35

E is not used.[11]

---

[11] *I*, *X* and *Y* are reserved and should be zero.

Set Sweep Busy, and write into storage all cached words whose written bits are set. Invalidate the entire cache, i.e., clear all valid and written bits. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPUO Sweep Cache, Unload One Page (CONSO CCA,)

70174	*I*	*X*	*Y*
0 … 12	13	14 … 17	18 … 35

Set Sweep Busy, and write into storage all cached words whose written bits are set and which are found in entries for the physical page specified by bits 23–35 of E. Invalidate all entries for the specified page, i.e., clear both their valid and written bits. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor

Timing. Simple invalidation takes little time, and it interferes minimally with the program since it requires no storage references. Otherwise an average sweep requires on the order of several hundred microseconds, but varies widely depending on the number of references required. Allowing the program to run simultaneously slows down the sweep because of competition for storage cycles, but program time is saved nonetheless.

Management of the cache is relatively straightforward. With external channels the program must simply be sure always to update storage pages before having them sent out, and to invalidate the cache entries for pages being brought in so processor references will go to storage for the new data.

The same procedures are used for a multiprocessor system, but here a problem arises when different processors are allowed to reference the same page at the same time, if either is allowed also to modify the page. Without modification the cache copies in both processors will remain valid; but if a processor modifies the page, the other cannot expect to get up–to–date data from cached words. To handle this situation, the pager includes mechanisms for bypassing the cache. Each page mapping[12] contains a cache bit for determining whether cache use is allowed for the given page. This cache bit applies only to an individual page, and has no effect at all unless cache use is enabled by the cache look bit. Analogous to the mapping cache bit is a load bit that applies to all unpaged references (such as pager references to the process tables). The look and load bits are among the conditions the Monitor provides to the pager. The way these "cache strategy" conditions govern cache use is as follows.

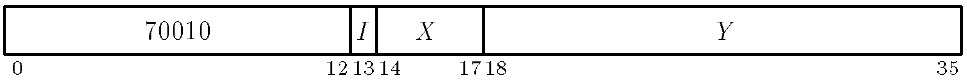
Look	
0	The cache is disabled — go to storage for all references.
1	Look in the cache for all references. This means always use the cache (reading or writing) for any locations that already have valid representations. Furthermore, when there is no valid representation for a reference, load the cache (reading or writing) if either the reference is unpaged and the *Load* bit is 1, or the reference is paged and the cache bit in the mapping for the page is 1.

---

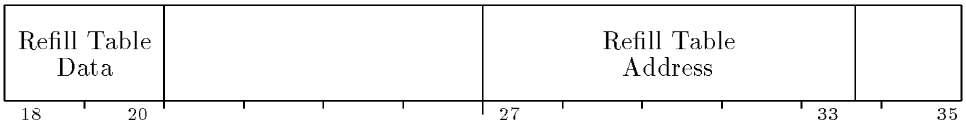
[12] For information on page mapping refer to §4.1.3 or §4.1.4 depending on whether the system uses respectively the TOPS–10 or TOPS–20 Monitor. Instructions for handling the pager are discussed in §4.1.5.

Initializing the Cache. The cache use logic contains two tables each with 128 entries. Each entry in the use table identifies the use history — from most to least recently used — of the group entries in the corresponding cache line. With each reference, the use entry for the line must be updated. But instead of containing complex computational logic, the hardware has a refill table that supplies new use entries as a function of the previous use history of a given line and the group entry currently being accessed in the line. Following power up the program must initialize the cache use logic by giving this instruction 128 times to load every 3–bit location in the refill table.

WRFIL Write Refill Table (BLKO APR,)



Load the refill data given by bits 18–20 of E into the refill table location specified by bits 27–33.[13]



After filling the refill table by stepping through locations 0–177 (values of E that are multiples of 4 from 0 to 774), the program should give an SWPIA to invalidate the indeterminate initial contents of the cache. During the sweep the normal monitoring of cache access by the use logic initializes the use table from the refill table. The way the use table gets set up depends on the data pattern — the "refill algorithm" — loaded into the refill table, and the pattern selected depends on the use strategy desired for the cache. To limit cache use to a single quadrant, simply load the quadrant number (0–3) into the entire refill table. The usual use strategy is to allow equal use of all quadrants and to start with a presumed use history of most to least recently used corresponding to the numerical order of the quadrants. To implement this strategy[14] load the following data pattern.

---

[13] The refill locations are selected by bits 27-33 to make use of the same lines that supply group numbers to address entries in the use table.

[14] For information on refill algorithms for other use strategies, refer to the writeup of MAINDEC 10–DDQDA–L–D(SUBRTN).

	0	1	2	3	4	5	6	7
000	0	1	2	3	4	5	6	7
010	3	1	2	3	2	1	2	3
020	7	1	2	7	1	1	2	7
030	6	5	6	7	5	5	6	7
040	0	3	2	3	0	2	2	3
050	0	1	2	3	4	5	6	7
060	0	7	7	7	0	0	0	7
070	4	6	6	6	4	4	6	4
100	3	1	3	3	1	1	1	3
110	0	7	7	7	0	0	0	7
120	0	1	2	3	4	5	6	7
130	4	5	5	7	4	5	4	7
140	0	1	2	2	0	1	2	1
150	0	5	6	6	0	5	6	0
160	4	5	6	5	4	5	6	4
170	0	1	2	3	4	5	6	7

4.1.3 TOPS–10 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.4. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses TOPS–10 paging, i. e. a Single–section KL10 running a TOPS–10 Monitor (microcode version earlier than 271). The next section presents equivalent information for the TOPS–20 paging. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–10 or TOPS–20, and are described in §4.1.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt functions and instructions reference executive virtual address space except in special cases where a function specifically calls for physical references. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the pager–microcode to carry out the mapping procedure, and also microcode references to retrieve interrupt instructions, handle traps and UUOs, and service the meters and front end.

Paging

All of memory both virtual and physical is divided into pages of 512 words each. In TOPS–10, the extended addressing capabilities of the KL–10 are not used. Hence, the virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18–bit addresses, where the left nine bits (18–26) specify the page number and the right nine (27–35) the location within the page. Physical memory can contain 8192 pages and requires 22–bit addresses, where the left thirteen bits (14–26) specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18–bit addresses into

22–bit addresses.[15] In this mapping the right nine bits of the virtual address are not altered; in other words, a given location in a virtual page is the same location in the corresponding physical page. The transformation maps a virtual page into a physical page by substituting a 13–bit physical page number for the 9–bit virtual page number. The mapping procedure is carried out automatically by the hardware, but the page map that supplies the necessary substitutions is set up by the kernel mode program. Each word in the map provides information for mapping two consecutive pages with the substitution for the even numbered page in the left half, the odd numbered page in the right half.

The pager contains two 13–bit registers that the Monitor loads to specify the physical page numbers of the user and executive process tables (UPT and EPT). To retrieve a map word from a process table, the pager uses the appropriate base page number as the left thirteen bits of the physical address and some function of the virtual page number as the right nine bits. For example, the entire user space of 512 virtual pages at two mappings per word requires a page map of just half a page, and this is the first half page in the user process table. Thus locations 0–377 in the table hold the mappings for pages 0 and 1 to 776 and 777. To find the desired substitution from the 9–bit virtual page number, the hardware uses the left eight bits to address the location and the right bit to select the half word (0 for left, 1 for right)

The executive virtual address space is also 256K, but the page map for it is in three parts. The map for the first 112K (pages 0–337) is in executive process table locations 600–757. The map for the second half of the virtual address space uses the same locations in the executive process table as are used in the user process table for the user map (locations 200–377 for pages 400–777). The map for the remaining 16K in the first half of the executive virtual address space is in the user process table, the mappings for pages 340-377 being in locations 400–417. This means the Monitor can assign a different set of thirty–two physical pages (the per–process area) for its own use relative to each user. Hence when switching from one user to another, the Monitor need change only the user process table, this single substitution making whatever change is necessary in the executive address space for a particular user.

Two figures are provided to show the organization of the virtual address spaces, the process tables and the maps for both user and executive. Figure 4.1 gives the correspondence between the various parts of the address spaces and the corresponding parts of the page maps. Figure 4.2 lists the detailed configuration of the process tables as determined by the hardware. Any table locations not used are reserved for future use by the hardware or for use by the Monitor for software functions. Note that the numbers in the half locations in the page map are the virtual pages for which the half words give the physical substitutions. Hence location 217 in the user page map contains the physical page numbers for virtual pages 436 and 437

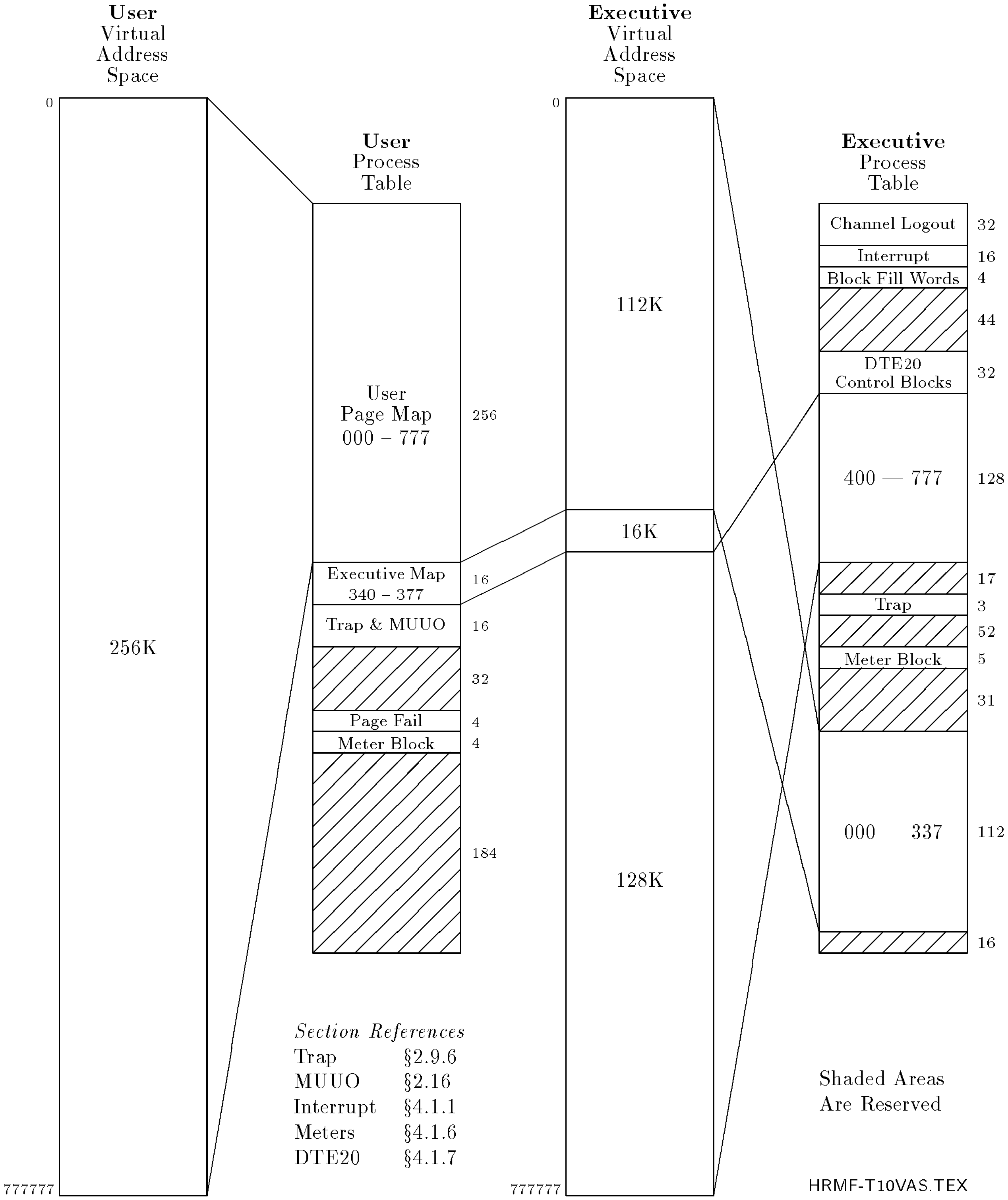
Although the virtual space is always 256K by virtue of the addressing capability of the instruction format, the Monitor usually limits the actual address space for a given program by defining only certain pages as accessible.[16] The Monitor also specifies whether each page is public or not, writable or not, and cacheable or not. The cache bit has an effect only if cache use is enabled as the current cache strategy (§4.1.2); in this case a 1 in the cache bit allows loading the cache for the physical

---

[15] For paging purposes page 0 has only 496 locations using addresses 20–777, as addresses 0–17 reference fast memory, which is unrestricted and available to all programs. (In general a user cannot reference the first sixteen storage module locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to storage.

[16] There is no requirement that the accessible space be continuous — it can be scattered pages. The convention however is for the accessible space to be in two continuous virtual areas, low and high, beginning respectively at locations 0 and 400000. The low part is generally unique to a given user and can be used in any way he wishes. The (perhaps null) high part is a reentrant area, which is shared by several users and is therefore write-protected.

Figure 4.1: KL10 TOPS–10 Virtual Address Space and Process Tables



Section References	
Trap	§2.9.6
MUUO	§2.16
Interrupt	§4.1.1
Meters	§4.1.6
DTE20	§4.1.7

Shaded Areas
Are Reserved

HRMF-T10VAS.TEX

Figure 4.2: TOPS–10 Process Table Configuration (KL10)

User Process Table

0	User Page 0	User Page 1
377	User Page 776	User Page 777
400	Executive Page 340	Executive Page 341
417	Executive Page 376	Executive Page 377
420	Reserved	
421	User Arithmetic Overflow Trap Instruction	
422	User Pushdown Overflow Trap Instruction	
423	User Trap 3 Trap Instruction	
424	MUUO Stored Here	
425	MUUO Old PC Word	
426	MUUO Process Context Word	
427	Reserved	
430	Kernel No Trap MUUO New PC Word	
431	Kernel Trap MUUO New PC Word	
432	Supervisor No Trap MUUO New PC Word	
433	Supervisor Trap MUUO New PC Word	
434	Concealed No Trap MUUO New PC Word	
435	Concealed Trap MUUO New PC Word	
436	Public No Trap MUUO New PC Word	
437	Public Trap MUUO New PC Word	
440	Reserved	
477		
500	Page Fail Word	
501	Page Fail Old PC Word	
502	Page Fail New PC Word	
503	Reserved	
504	User Process Execution Time	
505		
506	User Memory Reference Count	
507		
510	Reserved	
777		

Executive Process Table

0	Eight Channel Logout Areas Each: 0 Initial Channel Command 1 Gets Channel Status Word 2 Gets Last Updated Command 3 Reserved	
37		
40	Reserved	
41		
42	Standard Priority Interrupt Instructions	
57		
60	Four Channel Block Fill Words	
63		
64	Reserved	
137		
140	Four DTE20 Control Blocks	
177		
200	Executive Page 400	Executive Page 401
377	Executive Page 776	Executive Page 777
400	Reserved	
420		
421	Executive Arithmetic Overflow Trap Instruction	
422	Executive Pushdown Overflow Trap Instruction	
423	Executive Trap 3 Trap Instruction	
424	Reserved	
507		
510	Time Base	
511		
512	Performance Analysis Count	
513		
514	Interval Counter Interrupt Instruction	
515	Reserved	
577		
600	Executive Page 0	Executive Page 1
757	Executive Page 336	Executive Page 337
760	Reserved	
777		

page when referenced as this particular virtual page, whereas a 0 limits cache use to look but do not load. Each word in the page map has this format to supply the necessary information for two virtual pages.

Data for Even Virtual Page						Data for Odd Virtual Page					
A	*P*	*W*	*S*	*C*	*Physical Page Address Bits 14–26*	*A*	*P*	*W*	*S*	*C*	*Physical Page Address Bits 14–26*
0	1	2	3	4	5–17	18	19	20	21	22	23–35

Bits 5–17 and 23–35 contain the physical page numbers for the even and odd numbered virtual pages corresponding to the map location that holds the word. The properties represented by 1s in the remaining "page use" bits are as follows.

Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Public
W	Writable (not write–protected)
S	Software (not interpreted by the hardware)
C	Cacheable

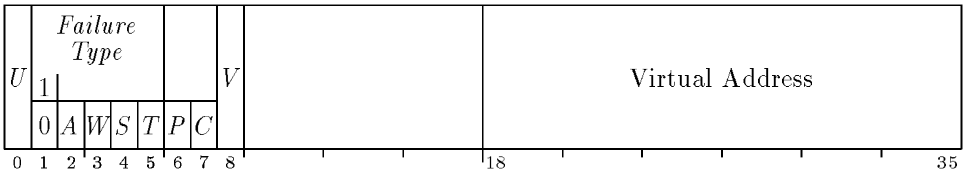
Page Table. If the complete mapping procedure described above were actually carried out in every instance, the processor would require two memory references for every reference by the program. To avoid this, the pager contains a page table, in which it keeps a large assortment of mappings for both the executive and the current user. In a manner analogous to the way the cache is organized to handle word groups of four, the pager handles mappings in sets of eight. A page set is eight consecutively numbered pages beginning with one whose number is a multiple of 10_8. Each page set consists of those pages whose mappings are contained in a single word group in the page map. The 512 locations in the page table are contained in sixty–four lines, each of eight locations holding the mappings for the eight pages of a set. The lines are identified by the possible page–set numbers in an address space, 0–77, and the individual locations are accessed by means of the virtual page numbers, 0–777. Each location has a parity bit and the complete mapping (i.e. map half word) for the virtual page that identifies it, including the physical page number and the five page use bits. Associated with each line are a bit that indicates whether the specified page set is in the user or executive address space, and a bit that indicates whether the set of mappings is valid or not (it is not suitable to clear a line as zero is a perfectly valid mapping, albeit for an inaccessible page). The user and validity bits for all lines collectively constitute the page table directory

When the program references a page contained in a page set whose mapping entry is tagged as valid and in the program address space, the 13–bit physical number from the mapping location for the virtual page is used as the left thirteen bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *A*, *P*, and *W* bits). If however the mapping set is invalid or is not for the correct address space, the pager makes a memory reference (referred to as a "page refill cycle") to get the word group containing the mapping for the specified virtual page from the page map. Even when there is no cache, all eight mappings from the word group are entered into the page table, filling and validating the line for the page set. This means the mappings will also be in the table for subsequent references to pages in the same set, although some may require a trap to the Monitor to make them accessible.

Note that all the mappings in an entire line of the page table are for a single space, user or executive. Since most programs are written beginning at page 0 (and often page 400 for a pure part), a mechanism is built into the table to avoid excessive refills due to switching between user and executive.. In the numbers actually used to select lines in the table, the value of address bit 19 is inverted in user address space. For a given page number, this causes a difference of 200 in the line selection number for user space as against executive space. Suppose the executive uses pages 0–37 and 400437, and also uses the per–process area, pages 340-377. Then if the user is limited to pages 0–137, 240–577 and 640–777, no conflict will ever occur between them in the page table.

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the pager terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap.[17] The trap operation makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current PC word in location 501, and sets up the flags and PC according to a new PC word in location 502. The processor then resumes operation in the new state at the location now addressed by PC. The page fail word supplies this information.



Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a virtual address was given for the reference. If bit 1 is 1, bits 6 and 7 are indeterminate, and the number in bits 1–5 ($\geq$ 20) indicates the type of "hard" failure as follows.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page, or a public program has attempted to fetch an instruction from a concealed page at an illegal entry point (one not containing a PORTAL). The failure for an illegal entry (which forces bit 8 to 0) occurs at the next reference, after the instruction is decoded, so the fail address is meaningless.

22 Page refill failure — this is a hardware malfunction. The pager found no mapping for the virtual page in the page table, so it refilled the line from the page map but still could not find it.

23 Address failure — this is caused by the satisfaction of an address condition selected by the

---

[17] A page failure that occurs during an interrupt instruction does not act this way. Instead it places a page fail word in AC 2, block 7, and sets the In–Out Page Failure flag (CONI APR, bit 26), requesting an interrupt on the level assigned to the processor.

program. It is used for debugging purposes, such as to find an instruction that is maliciously wiping out a memory location, and is explained in §4.1.5 with the description of the DATAO APR, instruction that sets it up. Bit 8 is forced to 0 by this failure.

25 Page table parity error — the pager has encountered a page table mapping with incorrect parity.

36 AR parity error — the processor has detected incorrect parity in a word read into AR (arithmetic register) from a storage module, the cache, or the E bus, and has saved the word (with correct parity) in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

37 ARX parity error — the processor has detected incorrect parity in a word read into ARX (arithmetic register extension) from a storage module or the cache, and has saved the word (with correct parity) in AC 1, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

If the failure is not one of these, then bits 1–7 have the format shown above, where A, W, S, P, and C are simply the corresponding bits taken from the mapping for the page specified by bits 18–26, and T indicates the type of reference in which the failure occurred — 0 for a read–only reference, 1 for any reference involving writing. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course T being 1 in conjunction with W being 0 certainly implies the cause of failure.

For a page fail trap, the new PC word is set up by the Monitor to transfer control to kernel mode. After rectifying the situation, the Monitor returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two–part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §4.1.1.

Note that a soft failure[18] seldom implies that anything is "wrong" — unless a program has attempted to write in a truly write–protected area. Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in core; these would be the virtual pages indicated as accessible. When the user attempts to gain access to a page that is not there (a virtual page indicated in its mapping as inaccessible), the Monitor would respond to the page failure by bringing in the needed page from the disk, either adding to the user space or swapping out a page the user no longer needs.

The same situation exists for writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered, distinguishing those that must be revised on the disk at the end from those that can be thrown away by setting the software bit. Then in response to a write failure, the Monitor makes the page writable and sets the software bit to indicate to itself that that page has in fact been altered and must be saved. When the user is done, the Monitor need write back onto the disk only those pages for which both W and S are set.

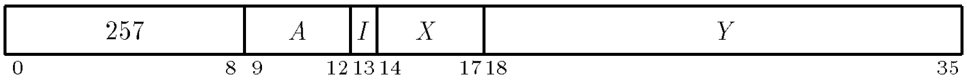
---

[18] In a soft page failure or page table parity error, the line containing the mapping for the page is invalidated on the assumption the Monitor will change it. When the instruction is restarted, the pager must go to the page map to get new information for the table.

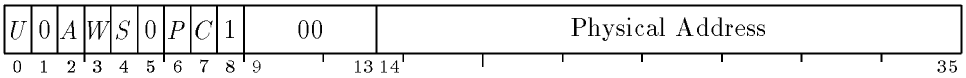
The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory, e.g. to set up a channel command list. For such purposes the processor has this instruction, which unlike all other instructions described in this chapter, is not an I/O instruction even though it is subject to the same restrictions.

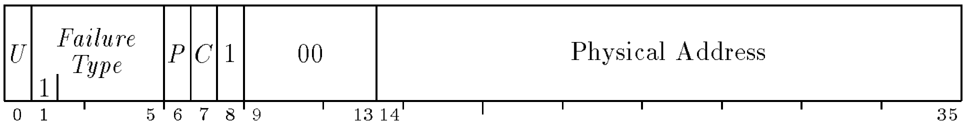
MAP **Map an Address**



If the pager is on and the processor is in kernel or user I/O mode, map the page number of the virtual effective address E and place the resulting physical address and other map data in AC. The information loaded into AC for a true mapping is of the form



where bits 14–26 are the physical page number the pager supplies for E, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and A, W, S, P, and C are the page use bits from the mapping as explained above. If however there is a parity error in the page table entry, or the paging is done in user mode public but the page, while accessible, is private, AC receives



The failure code can be only 21 or 25 for a proprietary or parity error, where in the latter case those bits supplied by the mapping, 6, 7 and 14–35, are meaningless.

This instruction cannot be performed in a user program unless User In–Out is set, nor in a supervisor program. Instead of mapping the address, it executes as an MUUO. If the pager is off, the result is undefined.

Notes. The instruction itself cannot fail because it does not actually reference memory: it just translates the address and gets other mapping data. However the effective address calculation could fail, and getting the mapping may require a refill, in which a hard failure could occur.

4.1.4 TOPS–20 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.4. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses TOPS–20 paging,[19] i. e. any KL10 running the TOPS–20 Monitor, or an Extended KL10 running the TOPS–10 Monitor (microcode version 271 or greater). The previous section presents equivalent information for the TOPS–10 Monitor. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–20 or TOPS–10, and are described in §4.1.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt functions and instructions reference executive virtual address space except in special cases where a function specifically calls for physical references. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the pager–microcode to carry out the mapping procedure, and also microcode references to retrieve interrupt instructions, handle traps and UUOs, and service the meters and front end.

NOTE

> Hardware paging operations are inextricably intertwined with the activities of the Monitor. The reader must be familiar with both to be able to understand either fully.

Paging

All of memory both physical and virtual is divided into pages of 512 words each. Physical memory can contain 8192 pages; its locations are specified by 22–bit addresses, where the left thirteen bits (14–26) specify the page and the right nine (27–35) the location within the page. The virtual memory space addressable by a program is 16,384 pages and requires 23–bit addresses, where the left fourteen bits (13–26) are the extended page number. However the virtual space is usually regarded as composed of thirty–two sections, each of 512 pages. With this view, the extended page number has two parts: the left five bits (13–17) specify the section, and the right nine (18–26) specify the page.[20] Thus within each virtual section, locations are specified by 18–bit addresses, where the left nine bits (18–26) are the page number. The hardware maps each section of the virtual address space into a part of the physical address space by transforming the 18–bit addresses into 22–bit addresses.[21] In this transformation the right nine bits of the virtual address are not altered; in other

---

[19] For additional information on the kind of paging employed in a TOPS–2O system, refer to "Storage organization and management in TENEX", by Daniel L. Murphy, AFIPS — Conference Proceedings, Vol. 41, page 23, AFIPS Press, Montvale, NJ.

[20] The reasons for holding to the section–page view are two. First, the page mapping procedures are actually set up that way. Second, although large data structures can arbitrarily cross section boundaries, the program cannot. For the program to get from one section to another requires an explicit transfer of program control. PC has twenty–three bits, but it counts in only the right eighteen: when going beyond the end of a section, it simply wraps around to the beginning of the same section (from location 777777 to 0).

[21] The mapping procedure is of course applied only to storage module references, whether cached or not. AC references, which can be made by any program, even when virtual page 0 is accessible, are made directly to fast

words a given location in a virtual page is the same location in the corresponding physical page. The translation maps a virtual page into a physical page by substituting a 13–bit physical page number for the 9–bit virtual page number. The mappings are different for each section by virtue of each section having a separate page map. The procedure is carried out automatically by the pager, but the maps that supply the necessary substitutions are set up by the kernel program.

Pointers to the page maps for the various user and executive virtual sections are contained in section tables that begin at location 540 in the user and executive process tables (UPT and EPT). The pager contains two 13–bit registers that the Monitor loads to specify the physical page numbers of these tables. To retrieve a section pointer from a process table, the pager uses the appropriate base page number as the left thirteen bits of the physical address and 540 plus the virtual section number as the right nine bits.[22] The section pointer must identify — either directly or indirectly — a physical page that contains the page map for the section. Every pointer and mapping takes one word, and since there are 512 pages in a section and 512 words in a page, a page map for a section requires exactly one page.

Figures are provided to show the organization of the virtual address spaces, the process tables and the section tables for both user and executive. Figure 4.3 gives the general layout of the process tables and shows the relation between the virtual address spaces and section tables. Figure 4.4 lists the detailed configuration of the process tables for the extended version of the processor and Figure 4.5 repeats this information for the single–section version of the processor.[23] Any table locations not used are reserved for future use by the hardware or use by the Monitor for software functions.

Although the virtual space is always thirty–two sections of 256K by virtue of the addressing capability of the instruction and indirect word formats, the Monitor usually limits the actual address space for a given program by defining only certain sections or pages as accessible. There is no requirement that the accessible space be continuous — it can be scattered pages. The Monitor also specifies whether each section or page is public or not, writable or not, and cacheable or not. To determine the mapping for a given virtual page, the microcode carries out a pointer evaluation procedure that starts at the appropriate entry in the section table. If it is discovered during this procedure that the section or page is inaccessible, the page map or the referenced page is not in memory, or the program is attempting to write in a write–protected page, the microcode traps to the Monitor, which must handle the situation. A trap to the Monitor for a reason of this sort is produced by generating a "soft page failure." But if nothing is amiss, the procedure is carried out entirely by the microcode — with no need to call the software — and it generates the mapping for the specified virtual page. The procedure requires access to both the section table and page map, to a memory status table in which the microcode keeps track of the use made of the page map and the program–referenced page, and perhaps to other predefined or software–defined tables as well. If the complete procedure were carried out in every instance, the processor would require at least five memory references for every one by the program. To avoid this, each mapping generated by the procedure is placed in a page table, and the pager makes its virtual–to–physical translations from the mappings held in the table. Hence it is necessary to go through the evaluation procedure only when the mapping is not

---

memory and require no mapping.

[22] In a single–section KL10 paging procedures are still as given here, but all addresses have zero section numbers.

[23] For release 1 or 2 of the TOPS–20 Monitor, the information given in Figure 4.5 is incorrect for User Process Table locations 424–427 and 500–503, which should read as follows:

424	MUUO Stored Here	500	Page Fail Word
425	MUUO Old PC Word	501	Page Fail Old PC Word
426	MUUO Process Context Word	502	Page Fail New PC Word
427	Reserved	503	Reserved

Moreover, the section tables are at User and Executive Process Table locations 440–477, rather than 540–577.

Figure 4.3: TOPS–20 Virtual Address Space and Process Table Layout

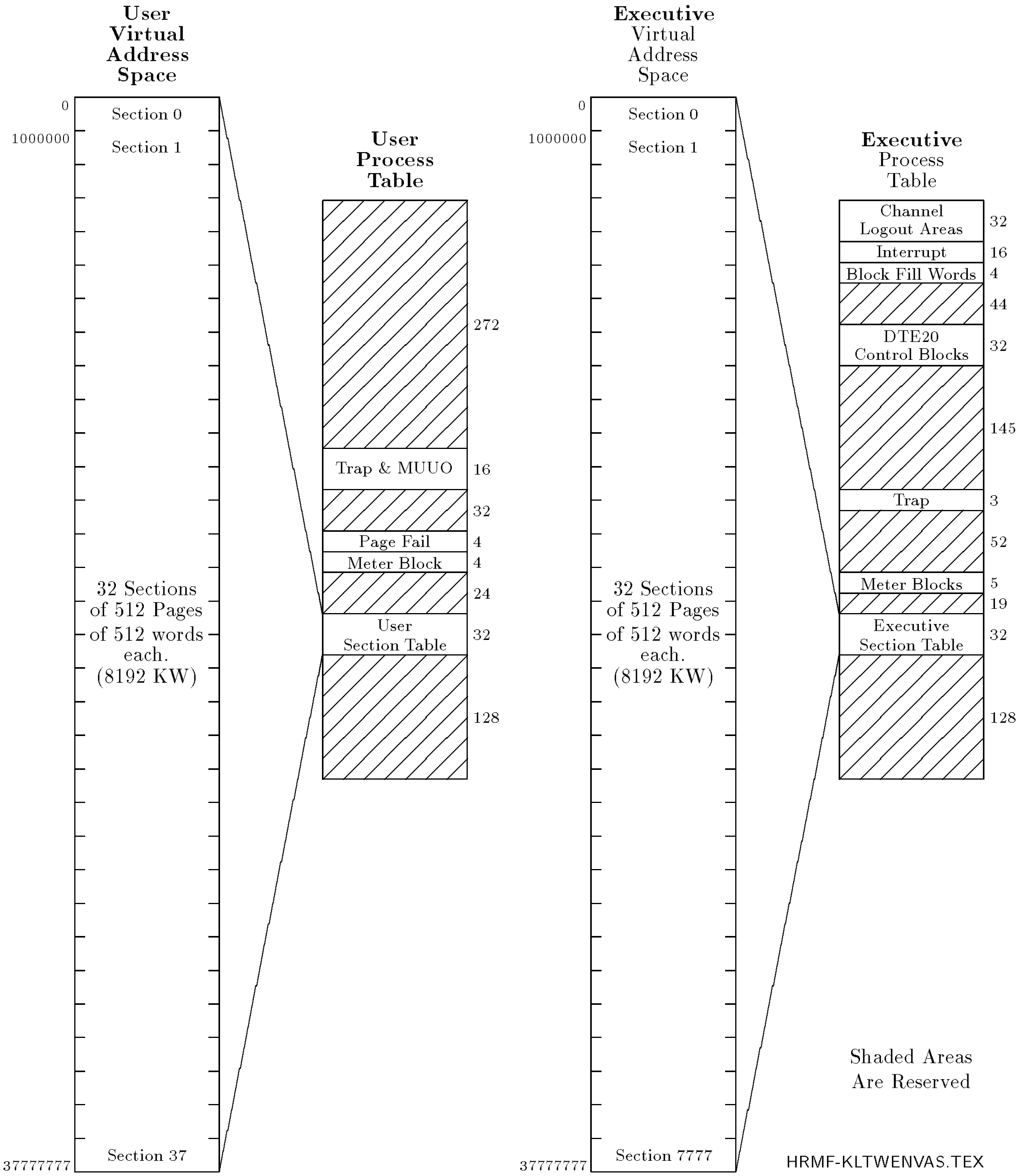


Figure 4.4: Extended TOPS–20 Process Table Configuration

User Process Table

0	Reserved	
420	Address of LUUO Block	*
421	User Arithmetic Overflow Trap Instruction	
422	User Pushdown Overflow Trap Instruction	
423	User Trap 3 Trap Instruction	
424	MUUO Flags MUUO Op Code, A	*
425	MUUO Old PC	*
426	E of MUUO	*
427	MUUO Process Context Word	
430	Kernel No Trap MUUO New PC	*
431	Kernel Trap MUUO New PC	*
432	Supervisor No Trap MUUO New PC	*
433	Supervisor Trap MUUO New PC	*
434	Concealed No Trap MUUO New PC	*
435	Concealed Trap MUUO New PC	*
436	Public No Trap MUUO New PC	*
437	Public Trap MUUO New PC	*
440 477	Reserved	
500	Page Fail Word	*
501	Page Fail Flags	*
502	Page Fail Old PC	*
503	Page Fail New PC	*
504 505	User Process Execution Time	
506 507	User Memory Reference Count	
510 537	Reserved	
540 577	User Section 0 User Section 37	
600 777	Reserved	

Note:
Asterisks indicate locations whose use differs froms those in the single–section process table, listed in the next figure.

Executive Process Table

0 37	Eight Channel Logout Areas Each: 0 Initial Channel Command 1 Gets Channel Status Word 2 Gets Last Updated Command 3 Reserved
40 41	Reserved
42 57	Standard Priority Interrupt Instructions
60 63	Four Channel Block Fill Words
64 137	Reserved
140 177	Four DTE20 Control Blocks
200 420	Reserved
421	Executive Arithmetic Overflow Trap Instruction
422	Executive Pushdown Overflow Trap Instruction
423	Executive Trap 3 Trap Instruction
424 507	Reserved
510 511	Time Base
512 513	Performance Analysis Count
514	Interval Counter Interrupt Instruction
515 537	Reserved
540 577	Executive Section 0 Executive Section 37
600 777	Reserved

Figure 4.5: Single–Section TOPS–20 Process Table Configuration

User Process Table

0	Reserved Note: Asterisks indicate locations whose use differs froms those in the extended process table, listed in the preceding figure.	
420	Reserved	*
421	User Arithmetic Overflow Trap Instruction	
422	User Pushdown Overflow Trap Instruction	
423	User Trap 3 Trap Instruction	
424	Reserved	*
425	MUUO Stored Here	*
426	MUUO Old PC Word	*
427	MUUO Process Context Word	
430	Kernel No Trap MUUO New PC Word	*
431	Kernel Trap MUUO New PC Word	*
432	Supervisor No Trap MUUO New PC Word	*
433	Supervisor Trap MUUO New PC Word	*
434	Concealed No Trap MUUO New PC Word	*
435	Concealed Trap MUUO New PC Word	*
436	Public No Trap MUUO New PC Word	*
437	Public Trap MUUO New PC Word	*
440 477	Reserved	
500	Reserved	*
501	Page Fail Word	*
502	Page Fail Old PC Word	*
503	Page Fail New PC Word	*
504 505	User Process Execution Time	
506 507	User Memory Reference Count	
510 537	Reserved	
540 577	User Section 0 User Section 37	
600 777	Reserved	

Executive Process Table

0 37	Eight Channel Logout Areas Each: 0 Initial Channel Command 1 Gets Channel Status Word 2 Gets Last Updated Command 3 Reserved
40 41	Reserved
42 57	Standard Priority Interrupt Instructions
60 63	Four Channel Block Fill Words
64 137	Reserved
140 177	Four DTE20 Control Blocks
200 420	Reserved
421	Executive Arithmetic Overflow Trap Instruction
422	Executive Pushdown Overflow Trap Instruction
423	Executive Trap 3 Trap Instruction
424 507	Reserved
510 511	Time Base
512 513	Performance Analysis Count
514	Interval Counter Interrupt Instruction
515 537	Reserved
540 577	Executive Section 0 Executive Section 37
600 777	Reserved

available in the page table. Since the objective of the procedure is to place a mapping in the table, it is referred to as a "page refill."

Page Table. A location in the page table contains a mapping entry in this format.[24]

A	*P*	*M*	*W*	*C*	Physical Page Address Bits 14–26

Each entry is identified as providing the physical page number for the translation for a particular virtual page in a particular section and address space (user or executive). A 1 in the *A* bit means the location contains a valid mapping, and the page is therefore immediately accessible without requiring further action by the pager. Otherwise the rest of the entry is meaningless,[25] as *A* being 0 does not necessarily mean the page is inaccessible — only that a refill is required to determine its accessibility. The properties represented by 1s in the remaining "page use" bits are as follows.

Bit	*Meaning of a 1 in the Bit*
P	Public. A 0 means the page is private.
M	Modified — and therefore writable without further ado. A refill produces a 1 in this bit if the page has already been modified or the reference that caused the refill is for write and the page is writable. A 0 does not imply that the page is write-protected, but simply that if a write reference occurs, the pager must find out if it can be written. Throughout this discussion, "write reference" means any reference involving writing; "read reference" means read only.
W	Writable. A refill sets this bit if the page is writable (i.e. not write-protected).
C	Cacheable. This bit has an effect only if cache use is enabled as the current cache strategy (§4.1.2). In this case a 1 in the cache bit allows loading of the cache for the physical page when referenced as this particular virtual page, whereas a 0 limits cache use to look but do not load.

The page table is organized for page groups in a manner somewhat analogous to the way the cache handles word groups. A page group is four consecutively numbered pages beginning with one whose number is a multiple of 4. Each page group consists of those pages whose mappings are contained in a single word group in the page map. The 512 locations in the page table are contained in 128 lines, each of four locations for holding the mappings for the four pages of a group. The lines are identified by the possible page group numbers in a section, 0–177, and the individual locations are accessed by means of the virtual page numbers, 0–177. Each location has a parity bit and the complete mapping resulting from a refill, including the physical page number and the five page use bits. Associated with each line is a bit that indicates whether or not the line is valid, a bit that indicates whether the specified page group is in user or executive address space, and five bits that identify the section

---

[24] In the engineering drawings and even in some Monitor documents, the *M* bit is labeled "writeable" and the *W* bit is labeled "software". which names are consistent with their use with the TOPS–10 Monitor.

[25] The microcode invalidates a mapping entry by clearing it, but clearing would not be sufficient were there no access bit, as zero is a legitimate mapping.

containing the page group.[26]

When the program references a page, the 13–bit physical number from the mapping for that page is used as the left thirteen bits in the physical address for the reference provided all necessary conditions are satisfied. When the directory indicates the appropriate line is invalid or contains mappings for a different section or address space, the pager changes and validates the directory entry to match the desired reference but invalidates the four locations in the line by clearing their access bits. It then executes a refill to get the needed mapping into the table and tries the reference again. If there is already an appropriate directory entry, but the individual mapping is invalid or the reference is for writing and M is 0, the pager does a refill to get a valid mapping or checks whether it can be revised to allow the desired reference.

Note that all the mappings in a line of the page table are for a single space, user or executive, and for a single section. Since most programs are written beginning at page 0, a mechanism is built into the table to avoid excessive refills due to switching between user and executive and among sections. In the numbers actually used to select lines in the table, the value of address bit 19 is inverted in user address space, and the value of address bit 20 is inverted in an odd numbered section. For a given page number this causes a difference of 200 in the line selection number for user space as against executive space, and a difference of 100 for an odd section as against an even one. Suppose the executive uses pages 0–77 and 400–744 in section 1. Then if the user is limited to pages 0–277 and 400–677 in any even section, no conflict will ever occur between them in the page table. In general a program should be organized so that it runs in a single section or in nonconflicting parts of different sections for some significant amount of time. Considerable yet unavoidable switching among sections can occur however in handling large data structures, as when it is necessary to handle the elements of a very large array in a number of different orders.

Page Refill

The refill of a mapping into the page table is accomplished by evaluating various types of pointers found in several kinds of tables. At some point in the procedure the microcode must encounter a "page address" that identifies the page map for the section, and it must end with a page address that identifies the physical page corresponding to the referenced virtual page. A page address has this format.

Storage Medium	Reserved	Page Number
12 17	18 22	23 35

If bits 12–17 are zero, the storage medium is memory: i.e. bits 23–35 supply the number of a page that is in memory. If bits 12–17 are nonzero, the page exists but is stored on some other medium — perhaps the disk — and the microcode traps to the Monitor. A page address may be contained in a pointer, in which case some of the bits at its left have defined uses. But when the page address stands alone, bits 0–11 of the word containing it can be used arbitrarily by the software.

Special Tables. Besides the section tables in the process tables, a refill makes use of two predefined

---

[26] The user bits, validity bits, and section numbers for all lines collectively constitute the page table directory. The Monitor invalidates the contents of the entire table by setting all the validity bits in the directory.

tables: the special page–address table (SPT) and the (core) memory status table (CST). These are software–determined tables in memory, but their base addresses are held in reserved fast memory locations, rather than in hardware registers like those of the process tables.[27]

The special page–address table contains page addresses that specify shared pages or special pages (e.g. those used as page maps or other software–defined tables). The microcode accesses specific entries in the SPT by indexing on a physical base address (bits 14–35) contained in AC 3, block 6. The pointer format provides for an index of eighteen bits, so the SPT can actually be as large as 256K (and it need not start on a page boundary).

Information about the use made by programs of the various physical pages is kept in the memory status table. In every refill, unless the base address is zero the microcode updates CST entries for both the page containing the page map and the page referenced by the program. The entry for a page is a full word, and is accessed by adding the page number to a nonzero base address contained in AC 2, block 6. If memory is fully implemented at 8192 pages, the CST occupies sixteen of them, but need not begin on a page boundary. Note that the microcode does not manipulate CST entries for the process tables, the SPT, nor the CST itself, unless they are actually referenced by the program — in other words, unless the refill is being performed for a program reference to one of the tables.

The status of a physical page in memory is indicated by a CST entry in this format.

State Code	Reserved	*M*
0 8	9 34	35

The Monitor keeps a state code in bits 0–8 of the entry; within the code, bits 0–5 represent the page age, which must be nonzero for the page to be usable, whether it is the program–referenced page or the page map. Bits 0–5 being zero causes an age trap to the Monitor.[28] The microcode updates the entry by anding a CST mask word into it and oring a CST data word into that result. These two words are held respectively in AC 0 and AC 1, block 6. Bits 32–35 in them must be all is or all 0s as illustrated in order to preserve hardware information.

Mask	1 1 1 1
0 31	32 35

CST Mask Word

Data	0 0 0 0
0 31	32 35

CST Data Word

A 1 in the *M* bit indicates the page has been modified since being brought into memory.[29] The

---

[27] Remember that all memory tables defined by the pager are in physical address space, i.e. they have physical base addresses. Of course, to load or access a table, the Monitor must use paged virtual addresses. Note that if the base address is limited to a page number (bits 14–26), the table must begin at a page boundary.

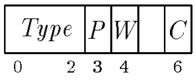
[28] Zero age usually means the page is being swapped in and is not yet available for reference. The Monitor can use part of a CST entry to record which processes use the page.

[29] At the completion of a process, the Monitor checks the CST to determine which pages have been modified and

microcode sets this bit in the entry for the referenced page — not that for the page map — if the reference is write and the page is writable.

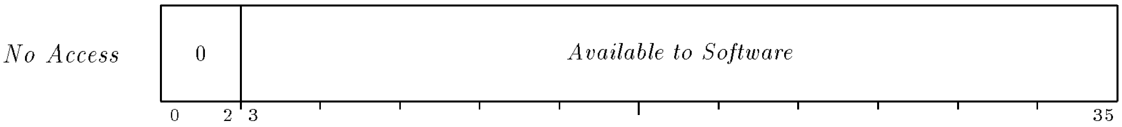
Indirect pointers make use of tables whose locations are defined entirely by the Monitor. In a single refill, these may include one or more secondary section tables or page maps. Each such table or map is determined by a page address and a 9–bit index, and is therefore a single page. Memory status is kept only for the page maps.

Pointers. The microcode evaluates two kinds of pointers: section pointers and map pointers. The former are used in section tables and the latter in page maps. Members of these two classes are identical in form but differ enough in function so they must be treated separately. There are four types of section and map pointers distinguished by a type code in bits 0–2; of these, three are access pointers, i.e. they allow access to the given section or page. An access pointer has this format in its left seven bits.

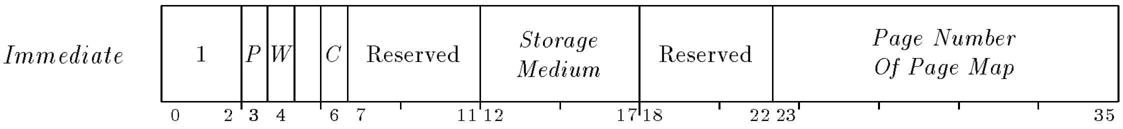


Every access pointer must have use bits for the section or page it represents. These bits, P, W and C, indicate whether the section or page is public, writable or cacheable. Throughout the evaluation procedure the microcode effectively ands these bits from one pointer to the next, so the final result requires that the given characteristics be specified at every step. In other words if P is 1 in the final pointer for the mapping, the page is public provided the entire section was also specified as public by the original section pointer, and "publicness" has been specified by every other pointer encountered along the way. Every access pointer must also either contain a page address or point to an SPT location that contains a page address.

Section Pointers. Entries in a section table are of these four types.[30]



The section is inaccessible.



If bits 12–17 are zero, the page map is in the page specified by bits 23–35. Otherwise the page map is not in memory

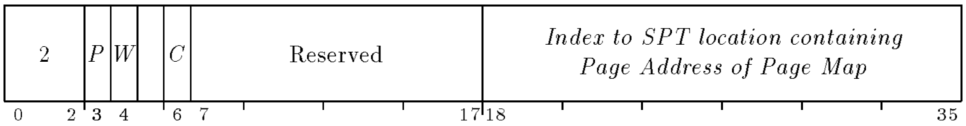
---

must be rewritten on the disk.

[30] Codes 4–7 are undefined.

An immediate pointer contains the page address of the page map.

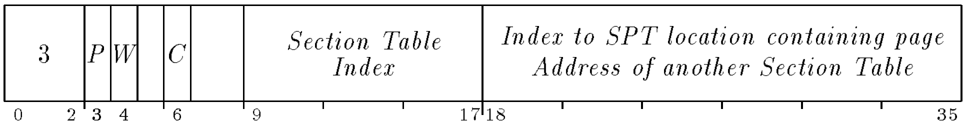
Shared



The page address of the page map is in the SPT at the location specified by bits 18-35.

This pointer is used for a page map shared by a number of processes. Switching to another map requires changing only the common SPT entry.

Indirect



In the SPT location specified by bits 18–35 is the page address of a secondary section table. The next section pointer to be evaluated is in that table at the location specified by bits 9–17

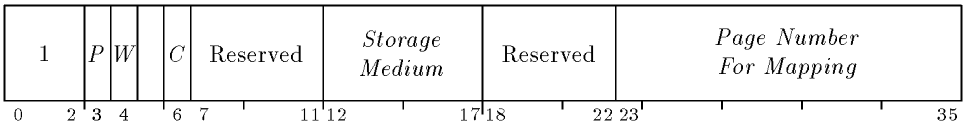
Indirect pointers are used for Monitor reference to per–job and per–process areas. The pointers remain while the second section table is swapped with the job or process, or the SPT entry is changed.

Map Pointers. Entries in a page map are of these four types.[30]

No Access

The page is inaccessible.

Immediate

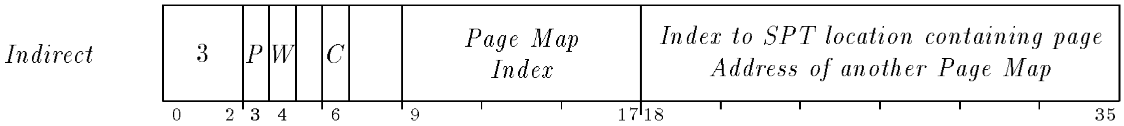


If bits 12–17 are zero, the physical page specified by bits 23–35 corresponds to the referenced virtual page. Otherwise the referenced page is not in memory.

An immediate pointer contains the page address for the mapping.

The page address for the mapping for the referenced virtual page is in the SPT at the location specified by bits 18–35.

This pointer is used for a physical page referenced as different virtual pages by different programs. The Monitor can move the page simply by changing the SPT entry.



In the SPT location specified by bits 18–35 is the page address of a secondary page map. The next map pointer to be evaluated is in that map at the location specified by bits 9–17.

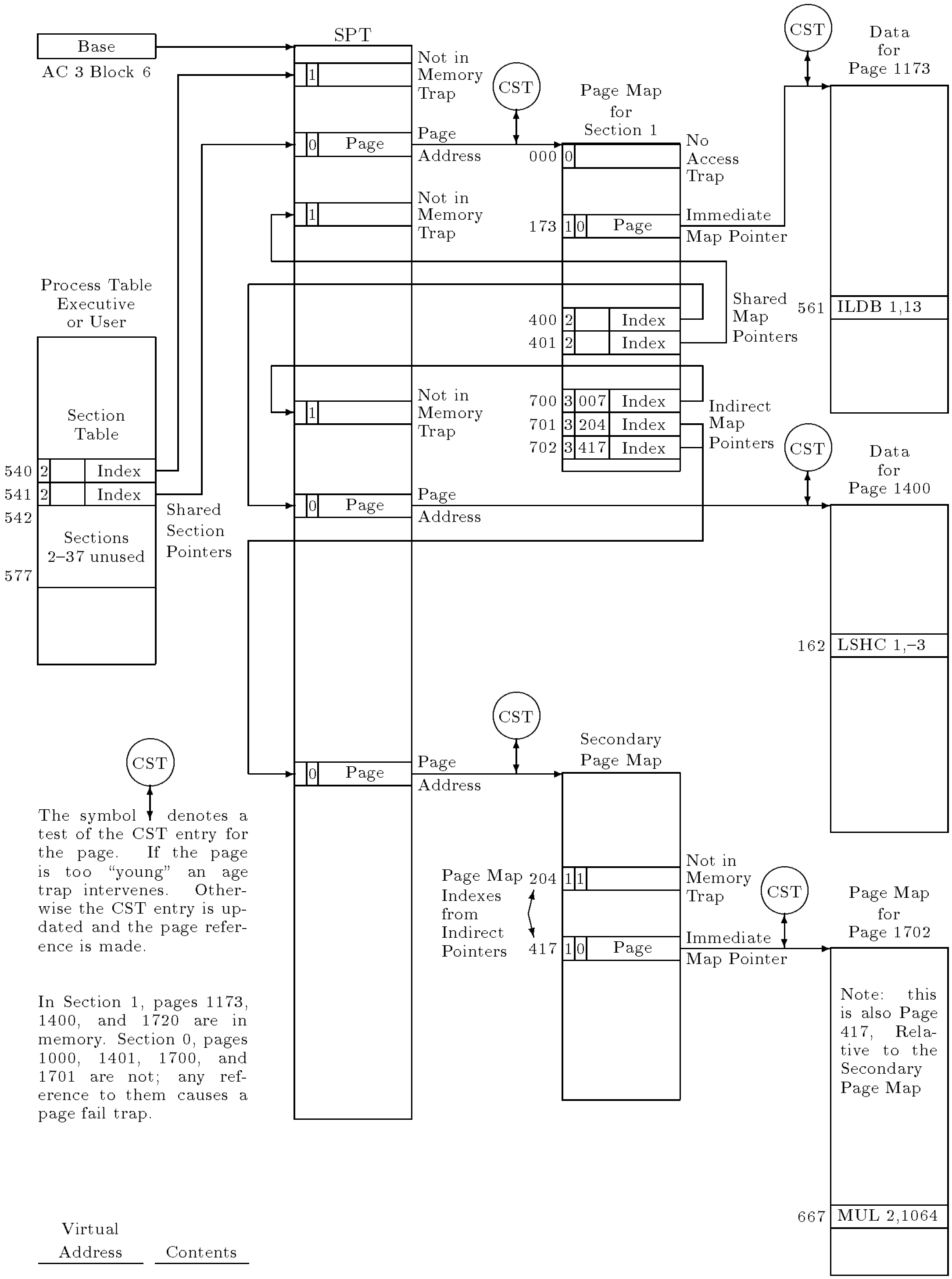
CAUTION

Indirect page pointers cannot be used for references made by interrupt instructions.

Refill Procedure. If the page table lacks a valid mapping for a reference, the pager must evaluate section and map pointers to get the desired mapping. The procedure begins with the pointer for the section from the process table, and the pager follows the trail laid by the various pointers, as illustrated in Figure 4.6. At any step the microcode traps to the Monitor if it encounters a no–access pointer or a page address that indicates the page is not in memory. The first part of the procedure, which may go to the SPT or indirectly through it to other section tables, evaluates section pointers to arrive at the page address of the page map. Using this physical page number as the left thirteen bits of an address and the number of the referenced virtual page as the right nine bits, the second part of the procedure retrieves a map pointer and evaluates it. This part may also go to the SPT or indirectly through it to other page maps to arrive at a page address for the mapping. Unless an age trap intervenes, or the CST base register is zero, memory status is updated along the way for any page maps used. If the reference can be made and there is no age trap for the referenced page, its status is updated including setting the M bit if the program is writing. The microcode then constructs the desired mapping, places it in the page table, and returns to the waiting reference.

The mapping data is constructed from the result of the pointer evaluation, including the running evaluation of the use bits, and has the format illustrated in the discussion of the page table. The microcode always places a 1 in the A bit to indicate that the virtual page is accessible and this is a valid mapping for it. P and C are simply the result of anding the P and C bits of the various pointers. M however is not. A refill sets up M and W according to the type of reference and the characteristics of the referenced page.

Figure 4.6: TOPS–20 Paging Pointer Evaluation (Extended KL10)



The symbol denotes a test of the CST entry for the page. If the page is too "young" an age trap intervenes. Otherwise the CST entry is updated and the page reference is made.

In Section 1, pages 1173, 1400, and 1720 are in memory. Section 0, pages 1000, 1401, 1700, and 1701 are not; any reference to them causes a page fail trap.

Virtual Address	Contents
1173561	ILDB 1,13
1400162	LSHC 1,–3
1702667	MUL 2,1064

Circumstances[31]	*MW*	*Effect*
Read reference, page not writable.	00	An attempt to write will fail.
Read reference, page writable but not yet modified (according to CST).	01	An attempt to write will succeed, after the mapping is revised.
Page writable, write reference or page already modified.	11	Sets M in CST entry; an attempt to write will succeed.

Page Failure

When for any reason the pager is unable to make a desired memory reference, or an extended effective address calculation encounters an incorrectly formatted indirect word, an event known as a "page failure" occurs. For this the microcode terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap.[32] The trap operation makes use of certain locations in the user process table depending on whether the KL10 is extended.

Extended KL10

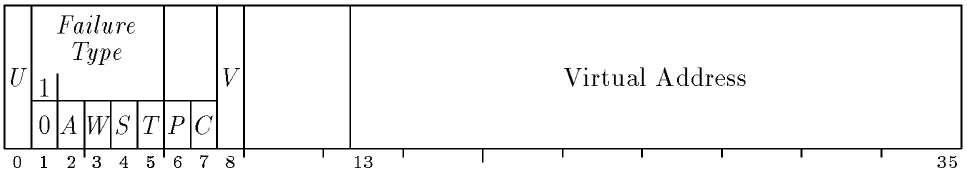
The trap places a page fail word in location 500, identifies the failed state of the processor by placing the current flag–PC doubleword in locations 501 and 502 (this includes the previous context section if the failure is in an executive program), sets up PC according to a new value in location 503, and clears the flags (placing the processor in kernel mode)

Single–section KL10[33]

The trap places a page fail word in location 501, identifies the failed state of the processor by placing the current PC word in location 502, and sets up the flags and PC according to a new PC word in location 503.

The processor then resumes operation in the new state at the location now addressed by PC.

The page fail word supplies this information.



Whether the violation occurred in user or executive virtual address space is indicated, respectively, by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a virtual address was given for the

---

[31] The missing circumstance produces a page failure.

[32] A page failure that occurs during an interrupt instruction does not act this way. Instead it places a page fail word in AC 2, block 7, and sets the In-out Page Failure flag (CONI APR, bit 26), requesting an interrupt on the level assigned to the processor.

[33] The process table locations given are as of Release 3 of the TOPS–20 Monitor. With Release 1 or 2 the trap uses locations 500–502 instead.

reference. If bit 1 is 1, bits 6 and 7 are indeterminate, and the number in bits 1–5 (≥ 20) indicates the type of "hard" failure as follows.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page, or a public program has attempted to fetch an instruction from a concealed page at an illegal entry point (one not containing a PORTAL). The failure for an illegal entry (which forces bit 8 to 0) occurs at the next reference, after the instruction is decoded, so the fail address is meaningless.

23 Address failure — this is caused by the satisfaction of an address condition selected by the program. It is used for debugging purposes, such as to find an instruction that is maliciously wiping out a memory location, and is explained in §4.1.5 with the description of the DATAO APR, instruction that sets it up. Bit 8 is forced to 0 by this failure.

24 Illegal indirect — an extended effective address calculation has encountered an indirect word with 11 in bits 0 and 1.

25 Page table parity error — the pager has encountered a page table mapping with incorrect parity

27 Illegal address — a memory reference has supplied an address whose section number is greater than 37. Bit 8 is forced to 0 by this failure.

36 AR parity error — the processor has detected incorrect parity in a word read into AR from a storage module, the cache, or the E bus, and has saved the word with correct parity in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

37 ARX parity error — the processor has detected incorrect parity in a word read into ARX from a storage module or the cache, and has saved the word with correct parity in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

If the failure is not one of these, then bits 1–7 (if meaningful) have the format shown above, where A, M, W, P, and C are simply the corresponding bits taken from the mapping for the page specified by bits 13–26, and T indicates the type of reference in which the failure occurred — 0 for a read–only reference, 1 for any reference involving writing. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Moreover the possible configurations for these bits are quite limited. A soft page failure can result only from actions taken in a refill or writability check. A valid page table mapping can require action by the pager only if M is 0 in a write reference. Hence in a soft failure resulting from a valid mapping, bits 0–8 of the page fail word are of the form

U	0	1	0	0	1	P	C	1
0	1	2	3	4	5	6	7	8

for a write failure. When no valid mapping is found, the page fail bits have the form

U	0	0	0	0	*T*	0	0	1
0	1	2	3	4	5	6	7	8

where for a write failure, *T* must be 1.

For a page fail trap, the extended KL10 automatically switches to kernel mode, and in the unextended version the Monitor should set up the new PC word for that action. After rectifying the situation, the Monitor eventually returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two–part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §4.1.1. Before returning to the failed instruction, the Monitor must invalidate the mapping for the page and revise the pointers for the new situation. Then when the instruction is restarted, the pager will do a refill to get the new, correct mapping.

A no–access pointer may well imply that the section or page simply does not exist. Otherwise a soft failure seldom implies that anything is "wrong." Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in memory. When the user attempts to gain access to a page that is not there (i.e. for which the refill encounters a not–in–memory page address), the Monitor would respond to the failure by bringing in the needed page from the disk, either adding to the user space, or swapping out a page the user no longer needs or has not used recently. Similarly a process using several sections may have only one in core at a time. While swapping is in progress, the Monitor runs some other user, returning to the interrupted job when the requested page is available.

The same situation exists for writability. Keeping track of modified pages is handled by the refill procedure using the memory status table. But a page may be write–protected because is it shared by a number of processes, wherein a change made by one might not be wanted by the others. Thus in response to a write failure, the Monitor might make a separate writable copy of the page for the sole use of the process that wishes to modify it.

The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory, e.g. to set up a channel command list. For such purposes the processor has this instruction, which unlike all other instructions described in this chapter, is not an I/O instruction even though it is subject to the same restrictions.

MAP **Map an Address**

257	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the pager is on and the processor is in kernel or user I/O mode, map the (extended) page number of the virtual effective address E and place the resulting physical address and other map data in AC. The information loaded into AC for a true mapping is of the form

U	0	1	M	W	0	P	C	1	00	Physical Address
0	1	2	3	4	5	6	7	8	9 13	14 35

where bits 14–26 are the physical page number the pager supplies for E, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and M, W, P, and C are page use bits from the mapping as explained above. Failure of the instruction to generate a valid mapping is indicated by AC receiving

U	*Failure Type*		00	Undefined
0	1 5	6 8	9 13	14 35

where bits 6–8 are undefined, and the failure code can be 21, 25, 27, 36 or 00 (refer to the preceding discussion of page failures). Of these, 25 and 36 represent what are effectively real failures: a parity error in the page table entry or in a word retrieved from memory in a refill. The others represent failures that would occur were the instruction actually to reference memory rather than simply requesting a mapping: 21, an attempt by a public program to reference a private page; 27, an illegal address; and 00, an age, no–access or not–in–memory trap in a refill.

This instruction cannot be performed in a user program unless User In–out is set, nor in a supervisor program. Instead of mapping the address, it executes as an MUUO. If the pager is off, the result is undefined.

Notes. The instruction cannot actually fail, because regardless of what happens, the refill or page fail microcode returns to it instead of trapping to the Monitor. The effective address calculation done for it could fail however.

4.1.5 Memory Management

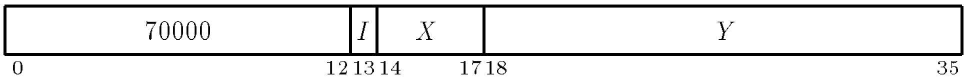
In order properly to manage memory, the kernel program must select the kind of paging and the cache strategy, set up process tables and page maps for itself and the various users, oversee the operation of the page table, and select the fast memory block to be used by each program (usually block 0 for itself). At any given time, accumulator, index register and fast memory references are made to that AC block that is assigned as "current." Given a particular processor mode (user or executive, public or private) and an appropriate process table and page map, the Monitor effectively defines the address space for a process (which may be itself) by specifying the base address for the process table and selecting the current AC block.

When a user program calls the Monitor it is usually to request some activity, which may often require the executive to gain access to the user address space. To facilitate the crossover from one address space to another, the same instruction through which the Monitor assigns its own current AC block also allows assignment of an AC block and section for the "previous–context" — i.e. the context of the process that made the call. These quantities, together with flags that indicate the mode of the caller, allow execution of instructions in the previous context (more about this subject later). At any point in time, the previous–context is essentially the circumstances in which the previous process was running. Note that the previous–context need not be the user; the same techniques can be exploited following a call from one level of the Monitor to another.

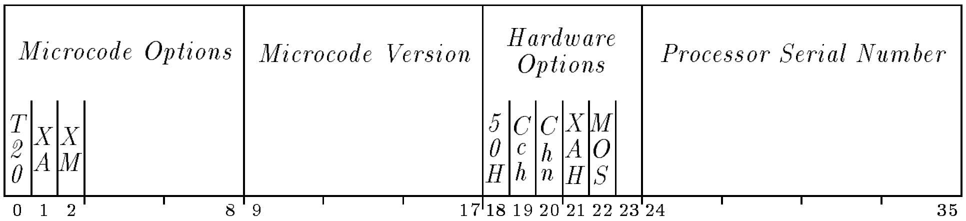
For initial setup, the kernel program must be cognizant of certain fundamental characteristics that can vary from one system to another. For this purpose the instructions for basic management include not only those that address the pager, but also one that addresses the processor to discover what those characteristics are.

The device code for the pager is 010, mnemonic PAG.[34]

APRID Arithmetic Processor Identification



Read the microcode version number, the processor serial number, and a listing of the fundamental characteristics of the system into location E as shown.



0 (*T20*) The microcode implements paging for the TOPS–20 Monitor; 0 indicates TOPS–10 paging.

1 (*XA*) The microcode handles extended addresses.

2 (*XM*, exotic microcode) The microcode differs in some way from the standard version.

18 (*50H*) Line power frequency is 50 Hz; 0 indicates the standard 60 Hz.

19 (*Cch*) Cache is present in this processor if this bit is 1; 0 indicates that the cache is absent, e.g., 2040 systems.

20 (*Chn*) RH20 internal channels are present in the system if this bit is 1; 0 indicates that external RH10 are used. External channels are used in 1080 configurations.

21 (*XAH*) The processor is an extended KL10; 0 indicates a single–section KL10. The microcode options must of course be consistent with the processor type.

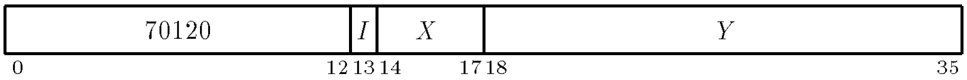
22 (*MOS*) The system has a master oscillator, which is available as an external clock source. In

---

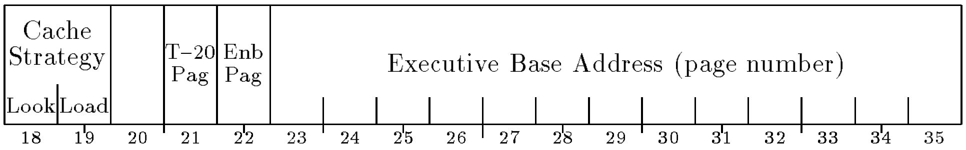
[34] BLKI PAG, is unassigned and executes as an MUUO.

a system containing MOS memory, the software must select this source (CPU clock source 2) from the PDP–11.

CONO PAG, Conditions Out, Pager



Set up the system–oriented characteristics of the pager according to the effective conditions E as shown.



Load bits 23–35 into the executive base register to select the executive process table. If bit 22 is 1 enable overflow trapping and enable the pager for the type of paging selected by bit 21: 1 for TOPS–20, or 0 for TOPS–10. The paging selected *must* be the same as that implemented by the microcode as indicated by APRID bit 0. A 0 in bit 22 prevents traps and disables paging so all memory references are to physical locations unpaged.[35]

CAUTION

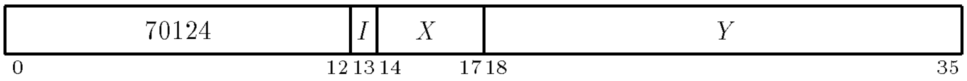
> Paging can be disabled only for executive mode. A user mode program will not run correctly unless the pager is turned on.

Select the cache strategy according to bits 0 and 1 as follows:

0x	Disable the cache.
10	Look for all references, but do not load physical references; for virtual references act as directed by the cache bit in the mapping for the page.
11	Make complete use of the cache for physical references; for virtual references act as directed by the cache bit in the mapping for the page.

Invalidate the entire page table by setting the invalid bits in all lines.

CONI PAG, Conditions In, Pager

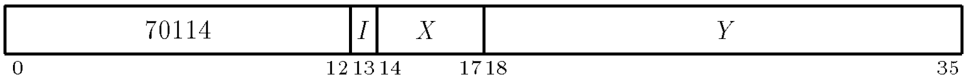


---

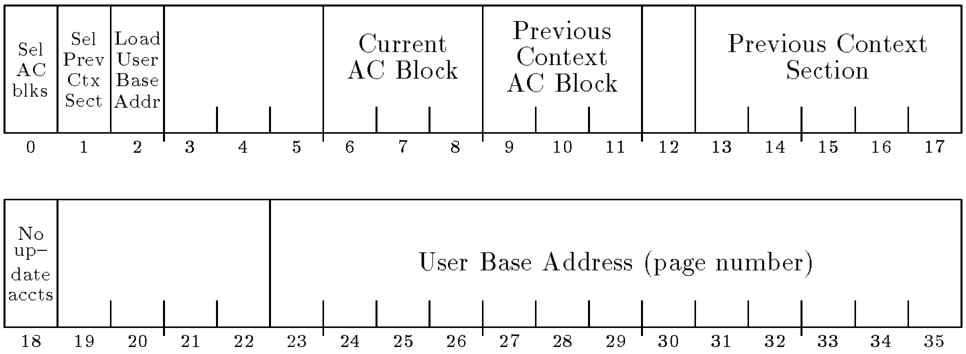
[35] Note that disabling the pager does not mean there can be no page failures, as these can be caused by conditions having nothing to do with paging, i.e. with translating virtual to physical addresses.

Read the system status of the pager into the right half of location E. The information read is the same as that supplied by a CONO.

DATAO PAG, Data Out, Pager



Set up the process–oriented elements of the pager according to the contents of location E as shown.

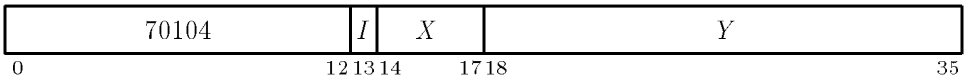


Bits 0–2 are change indicators for parts of the data word: when a bit is 0, the corresponding part of the word is ignored, and the equivalent value supplied by a previous DATAO remains in effect.

If bit 0 is 1, select as the current and previous–context AC blocks those specified by bits 6–8 and 9–11, respectively. If bit 1 is 1, select as the previous–context section that specified by bits 13–17 (which must be zero in a single section processor). If bit 2 is 1, perform these functions:

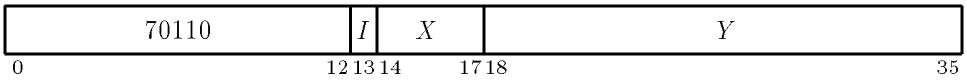
> If bit 18 is 0, update the user accounts as explained in §4.1.6.
>
> Load bits 23–35 into the user base register to select the user process table.
>
> Invalidate the entire page table by setting the invalid bits in all lines.

DATAI PAG, Data In, Pager



Read the process status of the pager into location E. The information read is in the same format as that supplied by a DATAO (bits 0–2 are 1s and bit 18 is 0). Note however that only the AC block designations and user base address are necessarily the same information supplied by a previous DATAO. When an MUUO stores its own context as given by the DATAO that set up the process containing it, it changes the designation of the previous–context section to that in which the program is currently running. Hence following a call by an MUUO, a DATAI PAG, in the called program will see as the previous–context section that specified by PC at the time the MUUO was performed.

CLRPT Clear Page Table Entry



TOPS–20	*TOPS–10*
Invalidate the page table mapping entry for the page referenced by E.	Invalidate the page table line (eight entries) containing the mapping for the page referenced by E.

At power turn–on the contents of the cache and page table are indeterminate, the processor is in kernel mode, paging is disabled, the cache is off, and the current AC block is 0 by default. After the front end loads the microcode, it then loads the initializing kernel program. This program, running unpaged in physical memory, should give an APRID to determine system characteristics and an SWPIA to invalidate the cache. The unpaged program ends with a CONO PAG, that selects the cache strategy, selects and enables paging, specifies the executive base address, and invalidates the page table. From this point the kernel program runs paged and must set up the first user or users, loading the user process tables and page maps, bringing in whatever parts of user programs and data that are consistent with good working-set management, and setting up the timing and accounting meters. Finally the Monitor gives a DATAO PAG, to assign the base address and current AC block for the first user, and then transfers control to the user program via an XJRSTF or JRSTF. The initial DATAO PAG, should have a 1 in bit 18 to inhibit updating accounts before any user has run.

On a call from the user via an MUUO, give a DATAI PAG, to determine the context of the user, i.e. his AC block and section. Then give a DATAO PAG, that assigns block 0 as current for the Monitor, assigns the user AC block and section as previous–context for accessing user space, but leaves the base address alone so the right paging is still available for such access. To return to the same user, reassign the AC block without changing the base address. Leaving the base address alone also avoids unnecessary updating of user accounts. Note that on the transfer to a user program no previous context values need be given as the user cannot employ PXCTs. For switching from one user to another, give a DATAO PAG, that updates the first user's accounts in his process table, as specified by the old base address, and then loads a base address for the new user. The transfer to a user is done with a JRSTF or XJRSTF; the latter also restores the previous–context section when used to return from a higher to a lower level within the executive.

The usual procedure for administering AC blocks is to assign block 1 to all users and assign two or three blocks for the sole use of interrupt routines. Suppose the assignments are: block 0 for the Monitor, block 1 for all users, block 2 for the highest priority interrupt level, block 3 for the second highest level, and block 4 for all other levels. Then in no circumstances is it necessary to determine which block to save, and interrupt routines on the highest, second highest and lowest levels need not save any. Moreover, the Monitor need not even store block 1 when it takes control from a user temporarily. When switching from one ordinary user to another, the Monitor usually stores the first user's accumulators in his process table or shadow area — this is locations 0–17 in user virtual page 0, an area not generally accessible to the user at all — and loads the new user's accumulators from his process table or shadow area, where they were stored after the last time the new user ran.

On a change from one process to another the entire page table must be invalidated, but this is done automatically by the instruction that assigns the new user base address. If the system uses shared or indirect pointers, or several virtual page numbers point to the same physical page, then the table must be invalidated whenever a page is removed from memory or a pointer is removed from a user section table or page map. On the other hand deletion of a page with a unique mapping requires only that a CLRPT be given to invalidate the line containing it. In multiprocessor operation all page tables must be cleared whenever one is. CST entries can be used to communicate paging information from one processor to another.

Previous–Context Execute

Ordinarily an instruction in a user program is performed entirely in user address space, and an instruction in the executive program is performed entirely in executive address space. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous–context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero A bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous–context. A PXCT can be given only in executive mode, but the previous–context may be the user, as following a call to the Monitor by the user. The previous–context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. (Note: it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.)

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory and index register references are made. To work as a PXCT, an XCT must be given in executive mode, and the bits in its A field (9–12) must not all be 0 (in user mode A is ignored). But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction, all accumulator references (specified by bits 9-12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a memory location.) If the instruction makes no memory operand references, as in a shift or immediate mode instruction, and it has no indexing or indirection (i.e. the instruction word gives E directly), then its execution differs in no way from the normal case. The only difference is in memory and index register references.

The previous–context is specified by four quantities. Following a call by an MUUO, the section in which the calling program was running (its PC section) and the fast memory block assigned to it appear as the previous–context section and current context AC block in the word read by a DATAI PAG,. For the called program, these two quantities can then be assigned as the previous–context by a DATAO PAG,. The current AC block of the calling program also appears in the process context word supplied by the MUUO. Various levels of the Monitor may all use fast memory block 0; or a separate block may be assigned to that part of the Monitor that uses PXCTs in handling MUUO calls from other parts of the Monitor.

Just as the current mode is indicated by the User and Public flags, the mode in which the calling program was running is indicated by Previous Context User and Previous Context Public.[36] At a

---

[36] Previous Context User and Previous Context Public are in the same flag bits that are used for User In-out and Overflow in user mode. The former has no meaning in executive mode, and the latter is not really necessary as the executive program is not ordinarily interested in performing extensive mathematical procedures.

call these flags may be set up automatically or they may be set up by a flag–PC doubleword or a PC word. Note that the restrictions on references made in the previous–context are those of the previous–context — not those of the context in which the PXCT is given — with the single exception that if the current program is running in section 0, the previous–context is also limited to section 0. Suppose the executive executes an instruction that references the concealed user area. Such a reference would fail if Previous Context Public were set.

Which references in the executed instruction are made in the previous–context is determined by 1s in the A portion of the PXCT instruction word as follows.

Bit	*References Made in Previous–Context if Bit is 1*
9	Effective address calculation of instruction, including both instruction words in EXTEND (index registers, address words by indirection); also EXTEND effective address calculation of source pointer if bit 11 is 1 and of destination pointer if bit 12 is 1.
10	Memory operands specified by E, whether fetch or store (e.g. PUSH source, POP or BLT destination); byte pointer; second instruction word in EXTEND.[37]
11	Effective address calculation of byte pointer; source in EXTEND; effective address calculation of EXTEND source pointer if bit 9 is 1.
12	Byte data; stack in PUSH or POP; source in BLT; destination in EXTEND; effective address calculation of EXTEND destination pointer if bit 9 is 1

Previous–context referencing is useful and reasonable in some instructions but inapplicable to others. There is no trap of any kind, and the effect of using the feature with an instruction to which it does not apply is simply undefined.

---

[37] Caution: if the current program is running in a non–zero section and the previous–context section is zero, and bits 9 and 10 of the PXCT are 0 and 1 respectively, the KL10 avoids global indexing in an attempt to apply section zero semantics to the effective address calculation of the instruction executed by PXCT. This works properly, unless the instruction uses indirect addressing. The instructions executed by PXCT must be carefully matched to the capabilities of the implementation.

Applicable

Move, XMOVEI
EXCH, BLT, XBLT[38]
Half word, XHLLI
Arithmetic
Boolean
Double move
CAM, CAI
SKIP, SOS, AOS
Logical Test
PUSH, POP, ADJSP
Byte
MOVSLJ (extended KL10 only)
MAP

Inapplicable

LUUO, MUUO
AOBJN, AOBJP
JUMP, AOJ, SOJ
JSR, JSP, JSA,JSP,JRST
PUSHJ, POPJ
XCT, PXCT
Shift–Rotate
String, except MOVSLJ
I/O

Note that no jumps can use previous–context referencing. Even among the instructions to which such referencing is applicable, only a limited number of the sixteen possible bit combinations is useful or meaningful. Doing an effective address calculation in the previous context (selected by bit 9 or 11) makes sense only if the corresponding data access is also in the previous–context (as selected by bit 10 or 12, except 11 or 12 in EXTEND). Only the combinations listed in Table 4.1 are permitted.

Execution of a BLT by a PXCT is limited to these three cases:

> Where all operations, regardless of context, are in section 0.
>
> Where the previous–context fast memory block is being saved in or restored from the current context, provided all addresses are local and thus in the same section. (Remember that regardless of context a BLT–given in–section address in the range 0–17 always refers to fast memory. Hence an AC block can never be saved in or restored from the first sixteen storage locations in any section.)
>
> Where all operations are confined to a single section in the previous context, as would be the case when clearing a user page.

In all other circumstances XBLT must be used instead.

Address Debugging

The address failure, or address break, feature of the pager implements the traditional program debugging technique of catching a particular type of memory reference to a selected location (it does not catch fast memory references). It may be used to determine whether a given program is modifying a particular location, is executing a particular piece of code, or is simply using a particular block of data. This instruction uses the processor device code to specify the circumstances in which a break shall occur.

---

[38] As of KL10 microcode 2.1[442], there are problems with the implementation of XBLT under PXCT.

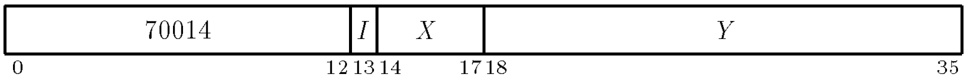
Table 4.1: KL10 Permissible PXCT Addressing Modes

Instructions	*9*	*10*	*11*	*12*	*References in Previous–Context*
General	0	1	0	0	Data
	1	1	0	0	E, Data
Immediate*	1	–	0	0	E
BLT	0	0	0	1	Source
	0	1	0	0	Destination
	0	1	0	1	Source, Destination
	1	1	0	0	E, Destination
	1	1	0	1	E, Source, Destination
XBLT	0	0	1	0	Source
	0	0	0	1	Destination
	0	0	1	1	Source, Destination
Stack	0	0	0	1	Stack
	0	1	0	0	Memory Data
	0	1	0	1	Memory Data, Stack
	1	1	0	0	E, Memory Data
	1	1	0	1	E, Memory Data, Stack
Byte	0	0	0	1	Data
	0	0	1	1	Pointer E, Data
	0	1	1	1	Pointer, Pointer E, Data
	1	1	1	1	E, Pointer, Pointer E, Data
MOVSLJ	0	0	0	1	Destination
(Extended KL10 only)	1	0	0	1	$E(=Y)$, Destination Pointer, Destination
	0	0	1	0	Source
	1	0	1	0	$E(=Y)$, Source Pointer, Source
	0	0	1	1	Source, Destination
	1	0	1	1	$E(=Y)$, Pointers, Source, Destination

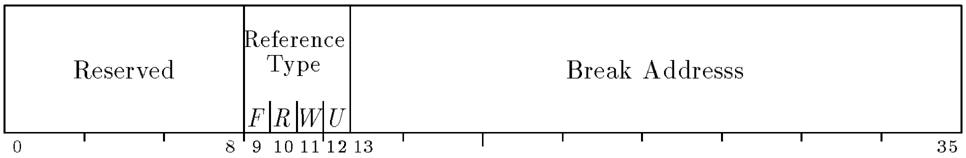
NOTE

* An A of 1000 is the "correct" configuration for a PXCT of an immediate mode instruction, but the KL10 inadvertently uses the current context section rather than the previous–context as would be desired in say the PXCT of an XHLLI. To get the previous–context section in the extended KL10, use 1100 instead.

DATAO APR, Data Out, Arithmetic Processor



Select the break address and the break conditions according to bits 9–35 of location E as shown (a 1 in a condition bit selects the condition indicated, a 0 makes no reference selection or selects the opposite address space).



The break conditions selected by 1s in bits 9–12 are as follows.

9 F: fetch. A normal fetch of an instruction in the program under control of PC.

10 R: read. Any reference that reads except the normal fetch of an instruction. This includes retrieval of operands, address words in an effective address calculation, or an instruction to be executed by an XCT or user LUUO.

11 W: write. Any reference that writes.

12 U: user. A reference made in user virtual address space (0 selects executive virtual address space).

The break mechanism operates only for virtual address space. It does not catch microcode physical references, such as to the process tables.

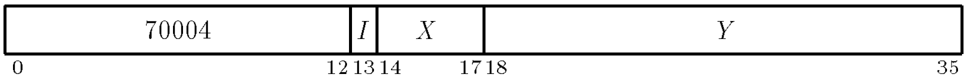
Whenever the processor attempts one of the selected types of reference to the location specified by the break address in the selected virtual address space, a page failure results[39] unless the Address Failure Inhibit flag is set. This flag, which is bit 8 of the program flags and can be set only by an instruction that restores them, prevents an address failure during the next instruction — the completion of the next instruction automatically clears it. If an interrupt or trap intervenes, the flag has no effect and is saved and cleared if the flags are saved with PC. If it is not saved, it affects the instruction following the interrupt or trap. Otherwise it affects the instruction following a return in which it is restored with PC. Using the inhibit flag, the Monitor can return to a user instruction that caused an address failure and "get by it."

Since this feature is entirely under the control of the above I/O instruction, it can be used quite flexibly for the executive to debug its own routines, or to debug a single user program without bothering either the executive or other users. The break conditions in effect at any time can be ascertained by giving this instruction.

---

[39] Executive conditions also catch virtual references in interrupt functions, but the page failure sets the In-out Page Failure flag instead of resulting in a trap for an address failure.

DATAI APR, Data In, Arithmetic Processor

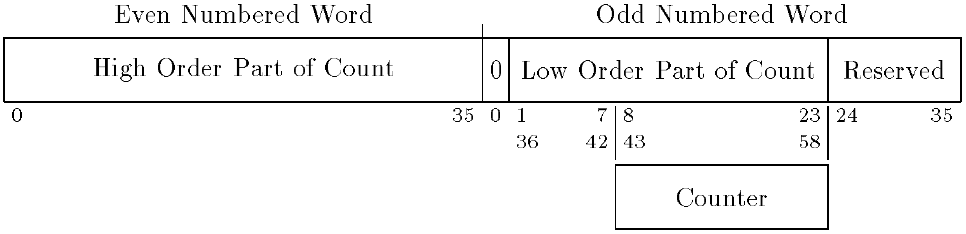


Read the current break conditions into bits 9–12 of location E. The information read is the same as that supplied by the last DATAO. (Note that the break address cannot be read.)

4.1.6 Timing and Accounting

The processor includes a subsystem with elements for keeping track of time, use of system facilities, and use of individual system features. One element is a standard 12–bit interval counter that is set up by the program to interrupt when the count reaches a preset value. The others are meters for keeping a 59–bit count, wherein only the low order sixteen bits are implemented in hardware. In each case the actual counting is done in a 16–bit hardware counter, while the overall count is kept in a doubleword in a process table. A count is updated from its counter by a procedure that is performed periodically by the microcode and whenever appropriate to an operation requested by the software. In the update procedure the contents of a counter are added into the corresponding count and the counter is cleared. Whenever the microcode checks for interrupt requests it updates any count whose counter is more than half full, i.e. whose MSB is 1. The current user accounts are generally updated when the Monitor switches to a new user.

A doubleword count is a 59–bit unsigned quantity whose format and relationship to the hardware counter are as shown here:



The entire first word comprises the high order thirty–six bits, and the low order twenty–three are in bits 1–23 of the second word.[40] Reserving bits for expansion at the low order end guarantees format compatibility with future machines that may be much faster (and therefore require bits for counting smaller time units). Altogether there are four meters that use this counter–doubleword format. One is a straightforward time base that counts at 1 MHz. Two keep track of process execution time and number of memory references for purposes for user accounting. Last is a mechanism for analyzing system performance by investigating the use of individual system features, either by counting the number of times particular events occur or measuring the duration of time particular procedures are in progress.

---

[40] Remember, it is a property of twos complement arithmetic that the sign can be used as an extra magnitude bit in an unsigned number. But since the hardware is set up for signed arithmetic, bit 0 of any lower order word must be skipped.

The program controls the various subsystem elements through two sets of I/O instructions using device codes 20 and 24, mnemonics TIM and MTR.[41] In general the meter code is for handling the accounting meters and the timer code is for the other elements, but the MTR conditions are for both. Data instructions read updated doubleword counts, but affect neither the counts nor the counters. Condition bits (in a CONO) directly affect only the 16–bit hardware counters. Of course a counter being enabled does mean updating of the doubleword count will probably occur. But to reset a count, the program must not only clear the hardware counter but separately clear the corresponding pair of locations in the process table.

System Timing

For regular system use, the processor provides a time base and an interval counter. The time base is a doubleword count (of the type described above) kept in locations 510 and 511 of the executive process table. It counts elapsed time in microseconds (a rate of 1 MHz). Drift is guaranteed to be less than 5 seconds per day for at least the first six years of use. To maintain day–to–day accuracy, the Monitor can reset the time base once each day from the line frequency clock in the front end processor (although a line frequency clock has quite low resolution, it has very high long–term accuracy.)

The interval counter is a 12–bit hardware counter that counts in 10μs increments (100 kHz). It can therefore count, and signal completion of, any interval from 10μs to 40.95 ms; and it can also be read at any time to determine how long some particular operation or procedure has taken. The counter can be used for any purpose by the software, but it is employed principally to signal the Monitor should a user tie up the system too long. Associated with the counter are two flags, Interval Done and Interval Overflow. Done sets when the counter reaches the value the program specifies as its period or reaches its maximum (all 1s); Overflow sets only if the counter reaches its maximum without ever matching its period.[42] Setting Done requests an interrupt on the level assigned to the counter, and the processor responds by executing the instruction in location 514 of the executive process table.

WRTIME **Conditions Out, Meters (CONO MTR,)**

70260	I	X	Y
0 12	13	14 17	18 35

Assign the interrupt level specified by bits 33–35 of the effective conditions E and perform the functions specified by bits 18–26 as shown.

		Accounting			Time Base				
Set up Accts		Exec PI	Exec Non–PI	Turn On	Turn Off	Turn On	Clr		Priority Interrupt Assignment
18	19 20	21	22	23	24	25	26	27 28 29 30 31 32	33 34 35

---

[41] Unassigned instructions using these codes are DATAO TIM,, BLKO MTR,, and DATAI MTR,. They execute as MUUOs.

[42] Overflow can occur only if at some time during the count, the program changes the period to a value less than the current counter value.

Only bits 24–26 and 33–35 are for the system timing features under discussion (time base, interval counter); bits 18–23 are for the accounting meters discussed in a later part of this section.

The interrupt level assignment is solely for the interval counter. Bits 24–26 control the hardware counter for the time base, wherein is clear it and turn it on or off (Os have no effect). The result of putting is in both bits 24 and 25 is indeterminate.

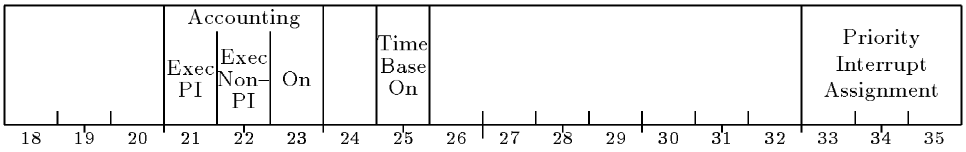
Bit 18 is a change bit for the accounting setup. If it is 0, bits 21–23 are ignored. But if it is 1, the way in which the meters are enabled is adjusted according to the configuration of those bits, where a 1 produces the indicated function and a 0 has the opposite effect. A 1 in bit 23 turns on the meters, and while on they automatically keep an account of user activity In addition the meters are enabled during interrupt routines, during noninterrupt executive time, or both (i.e. all executive time) as selected by bits 21 and 22.

Notes. The accounting bits affect only the circumstances in which the accounts are kept. Whenever the accounting meters are enabled, they automatically count both execution time and memory references.

CONI MTR, Conditions In, Meters

70264	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the accounting meters and time base, and the interrupt level assigned to the interval counter into the right half of location *E* as shown.



RDTIME Read Time Base (DATAI TIM,)

70204	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the time base doubleword count from locations 510 and 511 in the executive process table, add the current contents of the time base hardware counter to the doubleword read, and place the result in location E, $E+1$.

CONO TIM, Conditions Out, Interval Counter

70220	*I*	*X*	*Y*
0 12	13	14 17	18 35

Set up the interval counter according to the effective conditions *E* as shown.

Clr Intvl Cnt		Turn Intvl Cnt On	Clr Intvl Flag		Interval Period
18	19 20	21	22	23	24 25 26 27 28 29 30 31 32 33 34 35

A 1 in bit 18 clears the counter, and can be given simultaneously with a 1 or 0 in bit 21 to turn the counter on or off. A 1 in bit 22 clears both Interval Done and Interval Overflow. If the counter is on, Interval Done will set when the count reaches the value specified by bits 24–35.

CONI TIM, Conditions In, Interval Counter

70224	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the interval counter into location E as shown. The single bit that can cause an interrupt is bit 22, Interval Done.

	Interval Count
0 1 2 3 4 5	6 7 8 9 10 11 12 13 14 15 16 17

	* Interval Counter			Interval Period
	On	Done	Ovfl	
18 19 20	21	22	23	24 25 26 27 28 29 30 31 32 33 34 35

Bits 22 and 23 are the counter flags; note that Done can be set alone, but a 1 in bit 23 implies a 1 in bit 22 as well. Bits 24–35 are the period supplied by the CONO, and bits 6–17 are the current contents of the counter.

User Accounts

Two doubleword counts are kept for every user process. These are under the control of the accounting bits in a CONO MTR, as described above, and they always work together — i.e. the bits that select the circumstances for accounting do so for both of them. When the accounting meters are enabled, the execution meter counts at half the system clock rate while the processor is actually executing instruction operations, in other words except while waiting for memory (note that fast memory references are handled during execution — there is no wait). The memory meter counts memory references by or for instructions, not including fast memory references. Each individual instruction reference is regarded as a single reference even if it requires a page refill, and even if in one case memory control might handle four words whereas in the next three cases the references might be to the cache.

While the accounting meters are on, they are always enabled in user mode, except in certain special procedures discussed at the end of this paragraph. Additional enabling circumstances are selected by bits 21 and 22 of a CONO MTR,. Bit 21 enables while interrupts are actually being held, in other words during the execution of interrupt routines. Bit 22 enables in executive mode except while interrupts are being held. Programming 1s in both bits causes selection throughout executive mode. Note that interrupt routines executed in user mode are always included regardless of the selected circumstances by virtue of their being in user mode. Lastly there are two circumstances that automatically disable the meters regardless of any selection made and whatever mode the processor is in. These are the execution of interrupt functions (PI cycles) (§4.1.1) and special exempt microcode procedures: updating the meters, handling a page failure, and handling a TOPS–20 page refill.[43]

When a DATAO PAG, assigns a new user base address (§4.1.5), the accounts for the preceding user are updated in this process table unless such action is inhibited by a 1 in bit 18. The program can read the current user accounts by these two instructions.

RDEACT Read Execution Account (DATAI MTR,)

70244	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the process execution time doubleword count from locations 504 and 505 in the user process table, add the current contents of the execution time hardware counter to the doubleword read, and place the result in location E, $E + 1$.

RDMACT Read Memory Account (BLKI MTR,)

70240	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the memory reference doubleword count from locations 506 and 507 in the user process table, add the current contents of the memory reference hardware counter to the doubleword read, and place the result on location E, $E + 1$.

The accounting meters provide an accurate and reproducible measure of the resources used by a given process. Even though one model processor may differ in speed from another, the execution time count should be the same for a given program run on either of them (the unit of time counted will of course be different). Billing of charges to a user can be based on the execution time and the memory reference count taken separately, or a time equivalent can be assigned to a memory reference and the two accounts combined in a single quantity.

Performance Analysis

The performance analysis meter is a tool for studying the performance of the hardware and software

---

[43] A TOPS–10 page refill is excluded from accounting by virtue of being done by memory control while the execution meter is waiting.

of the system. With it, the analysis software can find bottlenecks, such as overuse of a particular system facility. Information of this sort should help the system administrator decide what new equipment to add or how to expand the system, and should help Digital decide how to modify existing software or what new hardware or software to design.

The result of an analysis is a doubleword count kept in locations 512 and 513 of the executive process table. Available to the analyzer is a large set of logic signals representing various conditions in the system. Incrementing of the hardware counter is controlled by a subset of these conditions selected by the program. The conditions are treated as a Boolean expression, and are divided into six groups, each corresponding to a term in the expression. Counting is enabled when the expression is true, which requires that all six terms be true. Within each term the conditions are ored, so a given term is true when any chosen condition in it is true. In each term the program must select some condition, or the term will be false by default. Selection of conditions is by means of the bit configuration of a word supplied to the analyzer. The following table lists the categories of conditions for the terms, the bits in the word that make the selection, and the individual conditions available in each category.

Terms	*Bits*	*Conditions*
Mode	27–28	User, executive, ignore.
Memory	12–16	Processor waiting (E box wait), cache miss, writeback for reference (cache writeback), writeback for sweep (sweep write), ignore
Interrupt	18–26	Interrupt on any level 0–7, no interrupt in progress
Channels	0–8	Any channel busy (0–7), ignore
Microcode	9	Microcode enable, ignore
Probe	10–11	Probe high or low, ignore

By setting bits 18–26 to select all available interrupt conditions — interrupts on all levels and no interrupt — the program effectively deletes the interrupt term from the expression. In other words it forces the term true so the state of the interrupt system has no effect on whether analysis counting is enabled. All other categories include a specific provision by which the program can force the term true and thus cause the selected conditions in it to be ignored in evaluating the expression. For example the mode choice is made by bit 27: 1 selects user mode, 0 selects executive. But a 1 in bit 28 causes the selection made by bit 27 to be ignored; thus enabling of the analyzer no longer depends on the mode and is purely a function of the conditions selected in other categories.

Besides selecting conditions for analysis, the program also chooses the counting method used by the analyzer. In the duration method the analyzer counts at half the system clock rate while the expression is true. In the event method the counter advances one step each time the expression changes from false to true. Selection of multiple conditions for the duration method produces a composite picture of performance. Suppose we select interrupts on levels 4 and 6 as our interrupt conditions. The analyzer will then give a count of the total time spent handling interrupts on those levels, and the nesting of an interrupt on level 4 within one on level 6 will not affect the result.

Event counting however can vary considerably depending upon the order in which events occur. If we choose only interrupts on level 6, each return to an interrupt routine at level 6 from some higher level that interrupted it will be counted as separate event; hence a single interrupt on the level of interest may be counted several times. On the other hand selecting interrupts on say levels 2 and 6 may mean that a level 6 interrupt plus half a dozen level 2 interrupts will be seen as only one event. This would happen if all of the level 2 interrupts occurred during the level 6 interrupt routine.

There are two instructions for the performance analyzer: one to set it up and one to read it.

WRPAE Write Performance Analysis Enables (BLKO TIM,)

70210	*I*	*X*	*Y*
0 — 12	13	14 — 17	18 — 35

Select the counting method and conditions for performance analysis according to the contents of location *E* as shown. (A dagger indicates a bit in which a 0 makes the selection indicated; otherwise 1 makes the selection indicated).

												†	†	†	†		
Select Channels									Ignr μc	Select Probe		Select Memory Conditions					
0	1	2	3	4	5	6	7	None		Low	Ignr	EBox Wait	Miss	Wrt Back	Swp Wrt	Ign	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Select Interrupt Levels									Select Mode		Evnt Dur	Clr Cnt					
0	1	2	3	4	5	6	7	None	Usr	Ignr							
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Bit groups corresponding to the terms in the enabling expression and the individual conditions that constitute the groups are as follows.

0–8	Channel conditions. Bits 0–7 select channels 0–7 busy. A channel is busy when it is waiting for a device to respond or a transfer is in progress. A 1 in bit 8 deletes the term from the expression.
9	Microcode condition. A 1 in this bit deletes the term from the expression. If the bit is 0, the counter can run only when specifically enabled by the microcode, which is the standard case.
10–11	Probe conditions. The probe is simply an available input at pin CA1 on the meter board, so the program must generally give a 1 in bit 11 to delete this term from the expression. Should a signal under investigation be connected to the pin, then a 0 in bit 11 enables bit 10 to select the input level that satisfies the condition: 0 high, 1 low.

CAUTION

> Connecting a signal line to the probe input may produce ringing in that line, which depending on its length, may seriously degrade signal quality and cause machine malfunction.

12–16 Memory conditions.[44] A 1 in bit 16 deletes this term from the expression. Otherwise 0s

---

[44] Note: M box references initiated by the E box include those for instructions, operands, interrupts, and special

(not 1s) in bits 12–15 select enabling conditions as follows.

12 The E box is waiting for the M box in a memory reference This is only for a reference made by the E box. Its duration may however encompass a writeback to free a cache group entry or a TOPS–10 page refill.

13 Because of an E box reference, the M box is fetching data from storage or filling the cache (a cache miss). This includes only a fetch and load stemming from an E box reference made because the cache does not contain the desired word or is not in use.

14 The M box is writing in storage because of an E box reference. This would usually be a writeback to free a cache entry

15 The M box is performing a writeback for a cache sweep.

18–26 Interrupt conditions. Bits 18–35 select interrupts on levels 0–7. An interrupt condition includes both the execution of an interrupt function and the subsequent interrupt routine, if any; in other words it includes both PI cycles and an interrupt held for the level. A 1 in bit 26 selects the condition that no interrupt is currently in progress. If bits 18–26 all contain 1s, the interrupt term is always true and thus ignored. Similarly all 0s holds it false.

27–28 Mode conditions. A 1 or 0 in bit 27 enables the counter during user or executive mode respectively: a 1 in bit 28 deletes this term from the expression.

29 This bit selects the method of counting when the expression corresponding to the set of conditions selected by bits 0–28 is true. A 1 selects the event method wherein there is one count for each time the expression becomes true; and a 0 selects the duration method wherein the counter increments at half the system clock rate while the expression is true.

Notes. There is no specific provision for turning the counter on and off. It functions automatically whenever the selected expression is satisfied, but it can easily be stalled by selecting an impossible combination. In particular, giving a WRPAE [40] clears the counter and disables it.

RDPERF Read Performance Analysis Count (BLKI TIM,)

70200	I	X	Y
0 12	13	14 17	18 35

Read the performance analysis count doubleword count from locations 512 and 513 in the exec process table, add the current contents of the performance analysis counter to the doubleword read, and place the result in location $E, E+1$.

Applications. The event method allows software to collect counts of the number of times specific events occur over a period. Examples are calls to the executive, interrupts on a particular level or disjoint interrupts to all levels, cache misses, cache misses in user mode, traffic on the channels. There are also more esoteric analyses, such as counting the number of times a particular instruction or set of instructions is used (this would require modifying the microcode to enable) or how often a particular piece of software is called (this would require a patch in the Monitor). But the event method is subject to the limitations discussed above. A low priority interrupt routine could easily

---

microcode procedures (meter update, page failure, TOPS–20 page refill). References for writebacks, cache sweeping, TOPS–10 page refills, and the channels are initiated by the M box.

be recognized several times, and with the selection of multiple conditions, events can be lost due to overlap. The memory conditions especially overlap one another, and channel events are very likely to be lost if combined with memory or interrupt conditions.

These limitations do not affect the duration method. Suppose we wish to determine the total time spent doing interrupts and waiting for memory references. Overlap here is of no significance: the fact that sometimes the system is doing both does not matter. Typical uses are measuring the duration spent in user mode, or in executive mode, handling interrupts, handling interrupts at a particular level, doing DTE20 console functions or byte transfers (interrupt level 0), doing writebacks, and so forth. With an enable inserted in the microcode, one could measure the time spent manipulating strings.

4.1.7 Front End Functions

Every system contains one or more PDP–11 front end processors. But from the point of view of the KL10, a front end is a DTE20 interface — it is only the DTE20 that the KL10 hardware, microcode and program see on the E bus, and it is only the relationship between KL10 and DTE20 that concerns us here (there is nothing in this section about the PDP–11 per se). A DTE20 handles communication between the central processor and a front end processor by way of the KL10 interrupt system. The program can assign a level for standard or vector interrupts, but the interface can also perform special interrupt functions — examine, deposit, byte transfer — on level 0. In general all but one of the DTE20s are restricted: this means that a unit can request special interrupt functions only if interrupt level 0 is enabled in it, and examine and deposit are restricted to communication areas defined by the Monitor.

Among the DTE20s, one is master and is thus unrestricted. It gains this privileged status by means of a switch setting on the unit. The master can perform diagnostic operations[45] (included among these are the console functions start, stop, execute, and continue), can perform the special interrupt functions even when level 0 is disabled, and can override the restrictions on examine and deposit so as to gain access to all PDP–10 memory in either physical or executive virtual address space or the executive process table. Removal of the restrictions by placing a 0 in the Q bit of the interrupt function word must be done individually for each transfer.

For each DTE20 the executive process table contains an 8–word control block. These blocks contain the following information for byte transfer, vector, examine and deposit interrupt functions.

Locations in Executive Process Table				
Unit 0	*Unit 1*	*Unit 2*	*Unit 3*	*Contents*
140	150	160	170	Output byte pointer (to 11)
141	151	161	171	Input byte pointer (to 10)
142	152	162	172	Vector interrupt instruction
143	153	163	173	Reserved
144	154	164	174	Size of communication area for examine
145	155	165	175	Relocation address for examine area
146	156	166	176	Size of communication area for deposit
147	157	167	177	Relocation address for deposit area

---

[45] Except for stopping, diagnostic operations should be performed only when the processor is halted or when something has actually gone wrong. Otherwise, they would interfere with normal traffic on the E bus.

A byte pointer is limited to a single word; it must therefore have a 0 in bit 12, and its address is interpreted in executive virtual address space, section 0. The programmer must also refrain from using any indexing or indirection (bits 13–17 must be zero). After the microcode increments the byte pointer selected by Q (0 out, 1 in) and calculates its effective address, an input byte is inserted at the appropriate position in a memory location, or an output byte from memory is sent to the DTE20 right–justified with the rest of the output word filled with 0s. An output byte transfer is essentially an ILDB–DATAO combination; input is a DATAI–IDPB. Output bytes larger than sixteen bits can produce spurious E bus parity errors in the DTE20.

In a DTE20 vector interrupt, the address part of the function word is ignored, and the microcode executes the instruction supplied by the control block. This should be a call to an interrupt routine.

Communication areas are defined separately for examine and deposit. Thus the Monitor might divide the overall communication area into separate parts for deposits by several units, but allow all of them to examine the entire area. The size of an area is given as a number of locations, and the relocation address is the physical address of the first location in the area. Suppose we wish to assign a deposit area of sixteen words beginning at location 22660 for DTE20 number 2. In locations 166 and 167 of the executive process table we would put respectively 20 and 22660. In its deposit function words the DTE20 would then use addresses 0–17, and these would be relocated to 22660–22677.

4.1.8 Error and Diagnostic Instructions

The first part of this section explains the instructions through which the software handles the error flags and identifies the source of a hardware error. The second part discusses a special instruction the Monitor uses to set up the memory system and to get diagnostic and configuration information directly from individual memory controllers. The objective of this treatment is to complete the definition of all KL10 instructions and to give the programmer what he needs to identify sources of hardware error for purposes of software recovery. For information on diagnosing equipment ills, the reader must turn to maintenance documents. Note that this section does not touch on diagnostic functions the front end can execute in the KL10 without the KL10 microcode running; that subject is treated in the maintenance documentation.

Error Monitoring and Investigation

A few hardware errors — specifically a parity error in the page table or in a word brought into AR or ARX from memory — are detected by the pager and produce a page failure. Other hardware errors detected in the processor or on the S bus are indicated by flags that can request an interrupt on a level assigned to the processor. Several of these flags also lock information about the bad reference into the error address register ERA. The program can read this register, and it continues to hold the same information, even should subsequent errors occur, until the flag that locked it is cleared.

The error conditions are generally regarded as important enough to be assigned to the highest priority level. However for conditions that may be associated with user instructions (a parity error or unanswered memory reference), the common practice is for the error interrupt to switch over to the lowest priority level by means of a program–set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

Error flags are handled by two condition I/O instructions that address the processor, which has device code 000, mnemonic APR.[46] These instructions also handle the sweep flags for the cache (§4.1.2). The instruction that reads ERA uses the interrupt device code.

CONO APR, Conditions Out, Processor Flags

70020	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 19–31 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

	Clr All I/O Dev	Enb	Dis	Clr	Set	Select Flags for Bits 20–23									Priority Interrupt Assignment		
		Selected Flags				SBus Err	No Mem	I/O Page Fail	MB Par	Cch Dir	Addr Par	Pwr Fail	Swp Done				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

A 1 in bit 19 generates the I/O reset signal, which clears the control logic in all of the peripheral equipment (but affects none of the internal devices, such as the pager or the processor flags).

Bits 20–23 select flag functions: is in these bits produce the indicated effects on the processor flags selected by is in bits 24–31. A 1 in bit 20 enables the setting of any selected flag to request an interrupt on the level assigned to the processor; a 1 in bit 21 disables the selected flags from requesting interrupts. Similarly a 1 in bit 22 or 23 clears or sets the selected flags. The result of putting is in both bits 20 and 21 or 22 and 23 is indeterminate.

Notes. Setting flags has of course no relation to what the flags represent; the function is used only to check out the flag logic.

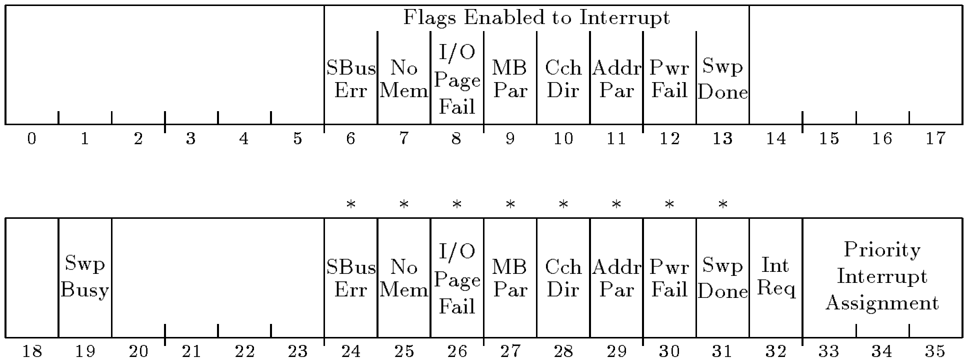
CONI APR, Conditions In, Processor Flags

70024	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the processor error and sweep flags into location *E* as shown (asterisks indicate bits that can cause interrupts).

---

[46] The processor device code is also used in several instructions for the pager and the cache.



6–13 A 1 in any of these bits indicates that setting the listed flag will request an interrupt on the level assigned to the processor by bits 33–35 of the CONO.

19 The cache is currently undergoing a sweep.

24 A storage controller has signaled the processor that it has detected an error in its own operation or in information it has received over the S bus or from one of its storage modules. If the type of error is not identified by there also being a 1 in bit 25, 27 or 29, then the condition is either an incomplete cycle or a parity error in data sent to the memory (all data received by memory is written, even if bad). Controller flags for some of these conditions can be read by the diagnostic instruction discussed in the second part of this section.

25 The processor attempted to access a memory that did not respond within a preset time. This time is 68μs on an extended KL10, 82μs on a single–section KL10. The setting of this flag locks information about the attempted reference into ERA. Since a nonexistent memory supplies zero data, on read this error should be accompanied by a 1 in bit 27.

26 A page failure has occurred in an interrupt instruction, or a word with even parity has been received at AR from the E bus (the latter can be recognized only if the transmitting device generates a parity bit). An interrupt failure caused by an address break sets this flag instead of producing an address failure (§4.1.5).

NOTE

A page failure in an interrupt instruction is regarded as a fatal error, and causes an interrupt instead of a page failure trap. The kernel program is expected to set up the interrupt instructions so that a software page failure simply cannot occur.

27 The buffer (MB) in memory control has received a word with even parity. The setting of this flag locks information about the reference into ERA.

28 A physical page number with even parity has been encountered in the cache directory. The setting of this bit turns off the cache, and it remains off until the flag is cleared by giving a CONO APR, with 1s in bits 22 and 28.

29 A storage controller has signaled that it has received an address with even parity from the processor. The parity check actually encompasses both the address and the control

signals that accompany it on the S bus. The setting of this bit locks information about the attempted reference into ERA.

30 Ac power has failed. The program should save PC, the flags, mode information and fast memory in storage, update the accounting meters, validate the entire cache, and halt the processor. Note that PC may point to an interrupt routine rather than the main program. After power is restored the front end must reboot the system, and the Monitor must reestablish the operating environment (§4.1.5).

31 A cache sweep has been completed.

32 Some processor flag is currently requesting an interrupt, i.e., some flag in bits 24–31 is set and has been enabled to interrupt as indicated by a 1 in the corresponding position in bits 6–13.

RDERA Read Error Address Register (BLKI PI,)

70040	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the error address register into location *E*. If No Memory, MB Parity Error or Address Parity Error is set, ERA contains information about the reference corresponding to the first of those flags to be set as shown.

Word Number	Reference Identification					Inde–termi–nate	0	High Order Address Bits
	Swp	Chn	Data	Src	Wrt			
0 1	2	3	4	5	6	7 8	9 10 11 12 13	14 15 16 17

Physical Address of First Word of Transfer
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

Bits 0–1 and 14–35 identify the physical location of the reference in which the error occurred. Bits 14–35 are the address of the specific memory reference made by the program or whatever. If the reference required only a single transfer, that address is the error address. But if the reference triggered a group transfer, bits 14–35 are the address of the first reference chronologically in the group, and bits 0 and 1 give the number of the word on which the error actually occurred. Note that word numbers are in physical, not chronological, order.

Information given in bits 2–6 identifies the reference. A 1 in bit 2 or 3 respectively means the reference was made for a cache sweep or a channel transfer. Bit 6 indicates the memory function being performed for the reference, where the read and write parts of a read–pause–write are separately indicated by 0 and 1. Bits 4, 5 and 6 together identify the source of the data for the transfer or attempted transfer (on write the word is always going to storage).

Bits 4–5	*Source with 0 in bit 6*	*Source with 1 in bit 6*
00	Storage for any read or read–pause–write	Channel status
01		Channel data
10		AR
11	Cache for channel read or TOPS–10 page refill	Cache writeback

ERA retains the same information until the program clears the locking flags by giving a CONO APR,22600+P. Of course only flags that are set actually need be cleared, and the routine that responds to errors should consider and clear all set flags. To facilitate diagnosis from the front end, the master reset does not clear ERA. Hence if need be, the front end can give diagnostic functions that reset the KL10 and then read ERA.

The processor includes provision for forcing bad parity to check the error detection logic. Bits 18–20 of a CONO PI, (§4.1.1) respectively cause even parity to be generated for an address sent to memory, a data word available from AR, and a page number entered into the cache directory. Where the data error shows up depends on where the word is sent from AR. Which errors are being forced can be seen by checking the flags in the same bits of a CONI PI,.

Programming Cautions. When handling parity error or nonexistent memory interrupts, the programmer should beware of the following.

- An incorrect word from memory to AR or ARX can result in both a page failure and an interrupt. In general the page fail trap to the Monitor can be expected to occur slightly ahead of the interrupt.
- Should an error flag be set while another interrupt request is being processed, the system would handle the lower priority interrupt before getting to the processor interrupt. This means PC may be pointing to a lower level interrupt routine rather than the program level at which the error occurred. Remember that during request processing, the interrupt system is otherwise static and the program continues.
- Even without inadvertent interference from another level, it is quite likely the processor will perform one or perhaps two more instructions between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may well be pointing to the first or second instruction following the one in which the error occurred.
- A processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second processor interrupt before the error–handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another — consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what went wrong.

S Bus Diagnostic Cycle

Ordinarily the S bus is used for the processor to reference memory. But the S bus also has a diagnostic cycle that allows the processor to communicate with the memory controllers rather than to access a particular location. The diagnostic cycle is initiated by the processor giving a special instruction that sends a function word to a controller and receives a word of error and diagnostic information back from it.

SBDIAG S Bus Diagnostic Function (BLKO PI,)

70050	*I*	*X*	*Y*
0 12	13	14 17	18 35

Send the contents of location E as a function word over the S bus to the controller specified by bits 0–4, and read the return word for the function from that controller into location $E + 1$. Which function a word represents is indicated by its code in bits 31–35.

4.2 KS10 System Operations

The information presented in this section is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own I/O, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

WARNING

> KS10 functions are implemented in microcode, which can be changed much more easily than hardware. Although user operations, described in Chapter 2, are deliberately kept as compatible as possible from one machine to the next, Digital will change the KS10 system microcode whenever such change will result in greater speed, efficiency or effectiveness. Therefore anyone writing system software should make sure to use the most recently updated version of this documentation, and before embarking on any project as enormous and critical as an operating system, to check with Large Systems Engineering for any changes not yet documented.

Programming for the system as a whole is programming in executive mode. Only the executive program is without instruction restrictions, and only it can, if needed, access physical memory unpaged. The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working sets, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of a user. The executive program accomplishes these objectives by handling all in–out for the system, setting up page maps, trap locations, interrupt locations and the like for both itself and the users, handling user accounts, and so forth. In other words, except for handling in–out, the activities of an operating system are the topics covered in this chapter. Of course the system programmer must also be quite familiar with all of the material presented in Chapters 1 and 2. In particular he must fully understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the JRST instruction and MUUOs (§2.9.4, §2.16).

System information for other Digital Equipment Corporation processors is given in the other sections of this chapter. The present section is devoted solely to the KS10; it contains two sections on paging, only one of which is applicable to a given system. §4.2.3 describes the paging used with the TOPS–10 Monitor; this paging is similar to that of the KI10. §4.2.4 treats the paging associated with the TOPS–20 Monitor. Both kinds of paging employ the same hardware — the difference lies in the microcode. All instructions discussed in this chapter are for system operations and are thus subject to the same restrictions as I/O instructions: namely, they can be performed only when the processor is in executive mode or is in user mode with User In–out set.

Some of the material presented here is related to the Unibus adapters. The chapter describes only the activities of the microcode undertaken for the adapters; it does not describe the adapters themselves or their programming.

4.2.1 Priority Interrupt

Most in–out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often result in loss of information and certainly results in operating the device below its maximum speed. The priority interrupt is designed with these considerations in mind, i.e., the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices through the Unibus adapters. The hardware also allows system flags (representing the console and conditions internal to the processor) to signal the program by requesting an interrupt. To avoid confusion with Unibus peripheral devices, let us regard the entities with which the interrupt system deals as "units". The system flags together constitute a unit.

Interrupt requests are handled through seven levels arranged in a priority chain, with assignment of units to levels entirely at the discretion of the programmer. To assign a unit to a level, the program sends the number of the level to the unit control register as part of its operating conditions. Levels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the unit from the interrupt levels altogether. Any number of units can be connected to a single level, and an adapter can be connected to two levels.

When a unit requires service it sends an interrupt request signal over the request line corresponding to its assigned level in the processor. The processor recognizes the request if the level is active (on). The request signal remains on the line until turned off by an appropriate response from the processor, either given by the program or generated automatically by the hardware. Thus if a request is not recognized or accepted when made, it will be when the appropriate conditions are satisfied. A single level will shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request.

In a Unibus system the I/O devices receive and send information via the adapter, and they signal the adapter to indicate their needs. To transfer data for high speed devices, the adapter can make direct access to memory over the KS10 bus. But to transfer data for slower devices and to handle control situations for all devices, the adapter uses the KS10 interrupt. For individual devices to signal the adapter, the Unibus has its own interrupt system of four levels, BR4–BR7, with the last having highest priority. Requests for interrupts on BR6 and BR7 are translated into requests on the KS10 interrupt level specified by the so–called "high" assignment, and those on BR4 and BR5 are translated into KS10 requests on the "low" level. Of course complete control over the adapter and the Unibus devices, including assignment of levels for KS10 and Unibus interrupts, is entirely in the hands of the KS10 program.

The request signal is generally derived from a flag that is set by various conditions in the device. Often associated with these flags are enabling flags, where the setting of some device condition flag can request an interrupt on the assigned level only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device. For example, a device may have half a dozen flags to indicate various internal conditions that may require service by an interrupt; by setting up the associated enabling flags, the program can determine which conditions shall actually request interrupts in any given circumstances.

Having recognized a request, the processor will do nothing further with it unless the priority interrupt system is on. But even with the system off, the processor will continue to recognize requests on other levels; and when the system is finally turned on, it will respond as though all requests had just been recognized, handling the highest priority one first.

Processing an Interrupt

The processor handles only one request at a time. When it is ready, it accepts the highest priority request currently recognized, provided that request is on a level higher than the current program (all levels are higher than a noninterrupt program). To process a request the microcode stops the program, turns off the interrupt system to prevent interference from other requests, and executes a "who are you?" cycle on the KS10 bus to determine which adapters are currently requesting interrupts on the accepted level. Note that at this point the processor is accepting not an individual request, but rather a class of requests: namely all those being made on the same level. In this cycle the microcode sends out the number of the level, and the individual adapters 0–3 indicate whether they are requesting interrupts on that level by placing 1s on bus lines 18–21 respectively. (Hence only lines 19 and 21 are used, for adapters 1 and 3.)

If no adapter responds, the request is assumed to be internal, originating either from the system flags or the program itself. In this case the microcode starts the interrupt by executing the instruction at location $40 + 2N$ in the executive process table, where N is the level number. Level 1 uses location 42, level 2 uses 44, and so on to level 7 which uses 56.

If the response on lines 18–21 is nonzero, the processor gives priority to the lowest-numbered adapter that has a request on the accepted level[47] by sending out the number of that adapter[48] in a vector request cycle on the bus. The vector address returned from a device is divided by 4, and the result[49] is used as an index into a table of interrupt instructions for that adapter. The table address is taken from executive process table location $100 + N$, where N is the adapter number (i.e., locations 101 and 103 are used). The processor then starts the interrupt by executing the instruction contained in the location specified by the table address plus the vector address divided by 4. The table pointer must be nonzero — otherwise an illegal interrupt halt occurs (§4.2.7).

Interrupt Instructions. An interrupt instruction is one executed in the interrupt location for a level, in direct response by the hardware (rather than by the program) to a request on that level. An interrupt location is either executive process table location $40 + 2N$ specifically for level N; or the adapter table location derived from the interrupt vector and the table pointer corresponding to the adapter having priority among those on the accepted level. Only two instructions can be used as interrupt instructions: JSR and XPCW. For either, the processor holds an interrupt on the level, turns the interrupt system back on, and takes the next instruction from the location specified by the jump (as indicated by the newly changed PC). For a JSR the processor automatically enters executive mode. For an XPCW it enters the mode specified by the new flag word. Either instruction is a jump to a service routine handled by the Monitor. Use of any other instruction results in an illegal interrupt instruction halt (§4.2.7).

The most important point of which the programmer must be aware is that even while User is set, the interrupt instructions are not part of the user program. They are executed in executive mode and are therefore subject only to executive restrictions. As an interrupt instruction, JSR automatically clears User to jump to an executive service routine. An XPCW should be set up to produce the same result.

---

[47] There are therefore two orders of priority associated with an interrupt: first the level, and then for all adapters requesting interrupts simultaneously on the same level, adapter number.

[48] Note that these are the adapter numbers (1 and 3), not the controller numbers used in I/O addresses (0 and 1).

[49] A vector address is a multiple of 4 because it specifies a pair of word locations in the byte-oriented Unibus addressing scheme.

Interrupt Programming

The program can control the priority interrupt system by means of these two instructions.

WRPI **Write Priority Interrupt Conditions**

70060	*I*	*X*	*Y*
0 12	13	14 17	18 35

Perform the functions specified by the effective conditions E as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

				Drop Prgm Req On Lvls	Clear PI Sys–tem	Make Prgm Req On	Turn On	Turn Off	Turn Off	Turn On	Select Levels for Bits 22,24,25,26						
						Selected Levels			PI System		1	2	3	4	5	6	7
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

22 On levels selected by 1s in bits 29–35, turn off any interrupt requests made previously by the program (via bit 24).

23 Turn off the priority interrupt system, turn off all levels, drop all program–set requests, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by 1s in bits 29–35, and force the processor to recognize them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a WRPI that selects the same channel and has a 1 in bit 22.

When this bit forces recognition of a request on the highest priority level, at most one additional program instruction may be performed before the interrupt.

25 Turn on the levels selected by 1s in bits 29–35 so interrupt requests can be recognized on them.

26 Turn off the levels by 1s in bits 29–35, so interrupt requests cannot be recognized on them unless made by a WRPI with a 1 in bit 24.

27 Turn off the interrupt system so no requests can be accepted.

28 Turn on the interrupt system so the hardware can process requests.

RDPI **Read Priority Interrupt Status**

70064	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the priority interrupt into location E as shown.

	Program Requests on Levels										Interrupt Holding on Levels							PI On	Levels On						
	1	2	3	4	5	6	7				1	2	3	4	5	6	7		1	2	3	4	5	6	7
0	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; and 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a WRPI with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on, and 1s in bits 29–35 therefore indicate active levels.

Dismissing an Interrupt. The processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt request can be accepted on that level or any of lower priority.

A routine dismisses the interrupt by using an instruction that restores the level on which the interrupt is being held at the same time it returns to the interrupted program. The proper instruction is XJEN (JRST 7,) or JEN (JRST 12,). Once the level is restored, the hardware can again accept requests and start interrupts on it and lower priority levels. These instructions also restore the flags: XJEN from the flag–PC doubleword if the routine was called by an XPCW; JEN from the left half of the PC word if the routine was called by a JSR.

CAUTION

> An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its level and all levels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Timing. The maximum time a device may wait for an interrupt to start depends on how many active devices are of higher priority and how long their service routines are. When a given request is of highest priority, its device need never wait longer than 40μs.

Special Considerations. When an interrupt occurs, PC points to the interrupted instruction (or to an XCT that executed it), unless the interrupt occurred in an overflow trap instruction, in which case PC points to the instruction that overflowed. After taking care of the interrupt, the processor can always return to the interrupted instruction. Either a) the instruction did not change anything; b) the interrupt was in the second part of a two–part instruction, where First Part Done being set prevents the processor from repeating any unwanted operations in the first part; or c) the interrupt occurred at some point in a multipart instruction where the microcode rigged the various pointers and other quantities so the processor actually restarts the instruction where it stopped, rather than from the beginning. However, in a BLT and in byte manipulation, the very mechanism that facilitates the return results in special properties of which the programmer must be aware.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two–part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a flag word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an ILDB or IDPB would skip a byte. And if the routine restored the flag, the interrupted ILDB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In–out flag is set, a user generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- No request can be accepted, not even on higher priority levels, while a request is being processed or an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- To prevent a device from hanging up a level, the programmer must be aware of — and satisfy — whatever requirements the device has for dropping the request.
- The interrupt instruction that calls the routine must be an XPCW or a JSR.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. Computations and any other time–consuming activities that can possibly be performed outside the routine should not be included within it.
- Never turn off the interrupt system in a routine unless it is absolutely necessary, and then always turn it back on again as soon as possible. If one or more levels can be turned off in place of the entire system, always do that instead.
- If the routine uses a UUO it must first save the contents of the locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type (§2.16).
- The routine must dismiss the interrupt (with an XJEN or JEN) when returning to the interrupted program. Flags and UUO locations should be restored.

4.2.2 Cache

For the user, the cache is transparent: any program simply gets information from memory and stores information in memory. But use of a cache as part of the memory subsystem reduces program time, since the cache is faster than the storage modules, and also reduces storage use by the program, making a larger percentage of total storage cycles available to other parts of the system. The cache is essentially 512 registers that duplicate the contents of frequently referenced storage locations in the virtual address space. its only use is for reading information from it instead of taking the time to go to storage, but this can result in a considerable saving for the program.

Each register in the cache corresponds to a unique position within a page. Associated with the cache is a directory that labels each register by the virtual page containing the word that the register duplicates. A directory entry also has a parity bit and other bits that identify certain characteristics of the reference that caused the word to be written in the cache. A cache hit can occur only when the circumstances of a read reference for a particular location match those of the last time the location was written. These requirements are a virtual reference[50] to the same page in the same address space (user or executive). Given a match, it is also required that paging be enabled by the Monitor, that the page map indicate the individual page is cacheable, and that the directory entry have correct parity. Moreover the cache can be disabled altogether from the console, and the microcode can inhibit its use in individual references.

There is no real programming for the cache except that the Monitor must decide, and so indicate in the page map, which pages are cacheable and which are not. Obviously the contents of the cache must be invalidated whenever there is any significant change in the virtual address environment, but the microcode handles this automatically. A sweep of the entire cache takes about 80μs.

4.2.3 TOPS–10 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.4. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This subsection covers these topics relative to a machine that uses the TOPS–10 Monitor. The next subsection presents equivalent information for the TOPS–20 Monitor. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–10 or TOPS–20, and are described in §4.2.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt instructions reference executive virtual address space. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the microcode to carry out the mapping procedure, retrieve interrupt instructions, and handle traps, halts and UUOs.

Paging

All of memory both virtual and physical is divided into pages of 512 words each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18–bit addresses, where the left nine bits (18–26) specify the page number and the right nine (27–35) the location within the page. Physical memory can contain 1024 pages and requires 19–bit addresses, where the left ten bits (17–26) specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18–bit addresses into 19–bit addresses.[51] In this mapping the right nine bits of the virtual address are not altered; in other

---

[50] The cache is also written on a physical reference, but the word cannot later be used as the directory entry is invalid (i.e., not virtual).

[51] For paging purposes page 0 has only 496 locations using addresses 20–777, as addresses 0–17 reference fast memory,

words, a given location in a virtual page is the same location in the corresponding physical page. The transformation maps a virtual page into a physical page by substituting a 10–bit physical page number for the 9–bit virtual page number. The mapping procedure is carried out automatically by the pager, but the page map that supplies the necessary substitutions is set up by the executive program. Each word in the map provides information for mapping two consecutive pages with the substitution for the even numbered page in the left half, the odd numbered page in the right half.

Two locations in the register file are used by the Monitor to specify the physical page numbers of the user and executive process tables. To retrieve a map word from a process table, the pager uses the appropriate base page number as the left ten bits of the physical address and some function of the virtual page number as the right nine bits. For example, the entire user space of 512 virtual pages at two mappings per word requires a page map of just half a page, and this is the first half page in the user process table. Thus locations 0–377 in the table hold the mappings for pages 0 and 1 to 776 and 777. To find the desired substitution from the 9–bit virtual page number, the hardware uses the left eight bits to address the location and the right bit to select the half word (0 for left, 1 for right).

The executive virtual address space is also 256K, but the page map for it is in three parts. The map for the first 112K (pages 0–337) is in executive process table locations 600–757. The map for the second half of the virtual address space uses the same locations in the executive process table as are used in the user process table for the user map (locations 200–377 for pages 400–777). The map for the remaining 16K in the first half of the executive virtual address space is in the user process table, the mappings for pages 340–377 being in locations 400–417. This means the Monitor can assign a different set of thirty–two physical pages (the per–process area) for its own use relative to each user. Hence when switching from one user to another, the Monitor need change only the user process table, this single substitution making whatever change is necessary in the executive address space for a particular user.

Figure 4.7 and Figure 4.8 show the organization of the virtual address spaces, the process tables and the maps for both user and executive. The first illustration gives the correspondence between the various parts of the address spaces and the corresponding parts of the page maps. The second illustration lists the detailed configuration of the process tables as determined by the hardware. Any table locations not used are reserved for future use by the hardware or for use by the Monitor for software functions. Note that the numbers in the half locations in the page map are the virtual pages for which the half words give the physical substitutions. Hence location 217 in the user page map contains the physical page numbers for virtual pages 436 and 437

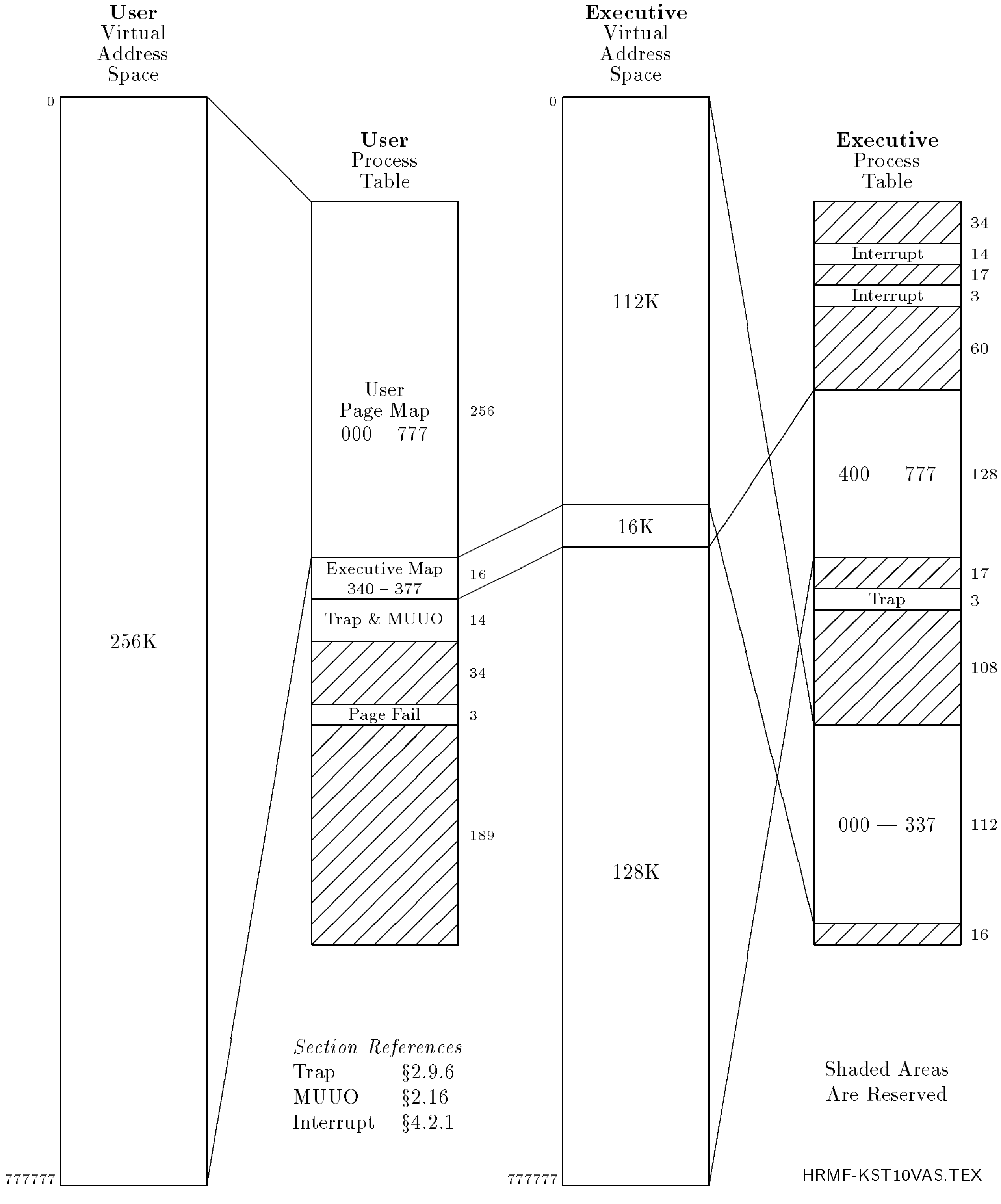
Although the virtual space is always 256K by virtue of the addressing capability of the instruction format, the Monitor usually limits the actual address space for a given program by defining only certain pages as accessible.[52] The Monitor also specifies whether each page is writable or not and cacheable or not. Each word in the page map has this format to supply the necessary information for two virtual pages.

---

which is unrestricted and available to all programs. (In general a user cannot reference the first sixteen storage module locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to storage.

[52] There is no requirement that the accessible space be continuous — it can be scattered pages. The convention however is for the accessible space to be in two continuous virtual areas, low and high, beginning respectively at locations 0 and 400000. The low part is generally unique to a given user and can be used in any way he wishes. The (perhaps null) high part is a reentrant area, which is shared by several users and is therefore write–protected.

Figure 4.7: KS10 TOPS–10 Virtual Address Space and Process Tables



Section References
Trap §2.9.6
MUUO §2.16
Interrupt §4.2.1

Shaded Areas
Are Reserved

HRMF-KST10VAS.TEX

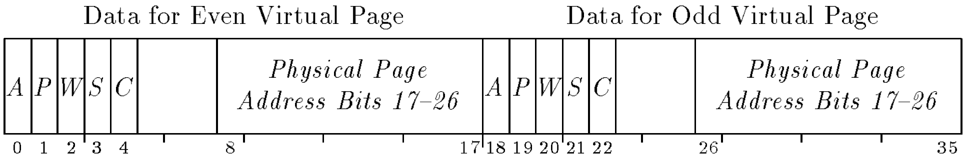
Figure 4.8: KS10 TOPS–10 Process Table Configuration

User Process Table

0	User Page 0	User Page 1
377	User Page 776	User Page 777
400	Executive Page 340	Executive Page 341
417	Executive Page 376	Executive Page 377
420	Reserved	
421	User Arithmetic Overflow Trap Instruction	
422	User Pushdown Overflow Trap Instruction	
423	User Trap 3 Trap Instruction	
424	MUUO Stored Here	
425	MUUO Old PC Word	
426	MUUO Process Context Word	
427	Reserved	
430	Executive No Trap MUUO New PC Word	
431	Executive Trap MUUO New PC Word	
432	Reserved	
433	Reserved	
434	User No Trap MUUO New PC Word	
435	User Trap MUUO New PC Word	
436	Reserved	
477		
500	Page Fail Word	
501	Page Fail Old PC Word	
502	Page Fail New PC Word	
503	Reserved	
777		

Executive Process Table

0	Reserved	
41		
42	Priority Interrupt Instructions	
57		
60	Reserved	
100		
101	Adapter 1 Interrupt Table Pointer	
102	Reserved	
103	Adapter 3 Interrupt Table Pointer	
104	Reserved	
177		
200	Executive Page 400	Executive Page 401
377	Executive Page 776	Executive Page 777
400	Reserved	
420		
421	Executive Arithmetic Overflow Trap Instruction	
422	Executive Pushdown Overflow Trap Instruction	
423	Executive Trap 3 Trap Instruction	
424	Reserved	
577		
600	Executive Page 0	Executive Page 1
757	Executive Page 336	Executive Page 337
760	Reserved	
777		



Bits 8–17 and 26–35 contain the physical page numbers for the even and odd numbered virtual pages corresponding to the map location that holds the word. The properties represented by 1s in the remaining "page use" bits are as follows.

Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Not used (public in other processors)
W	Writable (not write–protected)
S	Software (not interpreted by the hardware)
C	Cacheable

Page Table. If the complete mapping procedure described above were actually carried out in every instance, the processor would require two memory references for every reference by the program. To avoid this, the pager contains a page table, in which it keeps a large assortment of mappings for both the executive and the current user. The table has 512 locations, one for each virtual page number. Each location contains a mapping (from a map half word) for the virtual page that identifies it, including the physical page number and the W and C bits. Each location also has a parity bit, a bit that indicates whether the mapping is for user or executive address space, and a bit that indicates whether the entry is valid. A zero mapping is perfectly valid, but a location is labeled as containing no valid mapping by clearing it, thus clearing the valid bit. It is not necessary to keep the access bit, as mappings for inaccessible pages are not entered into the table.

When the program references a page whose mapping entry is tagged as valid and in the program address space, the 10–bit physical number[53] from the mapping for the virtual page is used as the left ten bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *W* bit). If however the entry is invalid or is not for the correct address space, or the reference is for writing and *W* is 0, the pager makes a separate memory reference (referred to as a "page refill") to get the mapping for the specified virtual page from the page map. The mapping is placed in the table unless the reference fails because the page is inaccessible or the program is attempting to write in a protected page.

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the page terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap. The trap operation[54] makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current PC word in location

---

[53] Actually table locations have eleven bits for physical numbers, but the most significant is not used.

[54] A page failure that occurs during an interrupt instruction does not act this way. Instead the processor halts (§4.2.7).

501, and sets up the flags and PC according to a new PC word in location 502. The processor then resumes operation in the new state at the location now addressed by PC. The same sequence of events occurs if the processor performs an I/O instruction and the adapter fails to indicate the transfer was accomplished.

There are two kinds of page failures, hard and soft. A hard failure means that something really is amiss, whereas a soft failure generally means only that the program requires some kind of service from the Monitor. A hard failure is indicated by a 1 in bit 1 of the page fail word, and the particular failure is specified by a code (which is therefore ≥ 20) in bits 1–5. There are three such failures of which two are true page failures, i.e., failures involving memory reference, and for these the page fail word has this format.

U	36 or 37	0	*P*	0	Address
0	1 5	6 7	8	9 16	17 35

Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a physical address was given for the reference. The code names the particular failure as follows.

36 Uncorrectable memory error — in a processor reference the memory controller has read an incorrect word from storage and was unable to correct it. The processor has saved the word in AC 0 and AC 1, block 7, and has set the Bad Memory flag (RDAPR bit 28).

37 Nonexistent memory — the processor has called for a storage reference over the bus but the memory controller did not respond. This error also sets the No Memory flag (RDAPR bit 27).

If the failure code is 20, the fail word instead has this format

U	20	0	1	0	1	0	*B*	I/O Address
0	1 5	6 7	8	9	10	11 12	13	14 35

and indicates a nonexistent I/O register, i.e., an I/O instruction gave an I/O address to which there was no response. A 1 in bit 13 indicates a byte operation. (The 1s in bits 8 and 10 mean a physical reference and an I/O function on the bus.) Note that this is not an I/O page failure, which is a true (memory) page failure and causes a halt.

A soft failure — of which there are two, an inaccessible page and an attempt to write in a write-protected page — is indicated by a 0 in bit 1. The fail word still contains the *U* bit and the virtual address, but now bits 1–8 have one of these formats,

Inaccessible

U	0	0	0	0	*T*	0	0	1
0	1	2	3	4	5	6	7	8

Write Violation

U	0	1	0	*S*	*T*	0	0	1
0	1	2	3	4	5	6	7	8

where *S* is simply the software bit taken from the mapping for the page specified by bits 18–26, bit 8 is the inverse of bit 8 in the hard case (1 means virtual), and *T* indicates the type of reference in which the failure occurred: 0 for a read–only reference, 1 for any reference involving writing. It is evident from inspection of the two configurations that bit 2 is actually the A bit from the mapping; and when the page is accessible, the 0 in bit 3 comes from the *W* bit. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course *T* and *A* both being 1 implies a write failure.

For a page fail trap, the new PC word is set up by the Monitor to transfer control to executive mode. After rectifying the situation, the Monitor returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two–part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §4.2.1.

Note that a soft failure seldom implies that anything is "wrong" — unless a program has attempted to write in a truly write–protected area. Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in core; these would be the virtual pages indicated as accessible. When the user attempts to gain access to a page that is not there (a virtual page indicated in its mapping as inaccessible), the Monitor would respond to the page failure by bringing in the needed page from the disk, either adding to the user space or swapping out a page the user no longer needs.

The same situation exists for writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered, distinguishing those that must be revised on the disk at the end from those that can be thrown away by setting the software bit. Then in response to a write failure, the Monitor makes the page writable and sets the software bit to indicate to itself that that page has in fact been altered and must be saved. When the user is done, the Monitor need write back onto the disk only those pages for which both *W* and *S* are set.

The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory. For such purposes the processor has this instruction.

MAP **Map an Address**

257	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

If the pager is on, map the page number of the virtual effective address *E* and place the resulting physical address and other map data in AC. If the page is accessible, the information loaded into AC is of the form

U	0	1	*W*	*S*	0	0	*C*	1	0	Physical Address
0	1	2	3	4	5	6	7	8	9 16	17 35

where bits 17–26 are the physical page number the pager supplies for *E*, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and W, S and C are the page use bits from the mapping as explained above (the 1 in bit 2 represents A). If the page is inaccessible, AC receives the given virtual address in place of a physical address; the word also includes U and a 1 in bit 8, but the remaining bits are all zero.

However, should a memory error occur during access to the page map, AC receives a hard page fail word. If the pager is off, the result is undefined.

Notes. The instruction cannot actually fail, because regardless of what happens, the page fail microcode returns to it instead of trapping to the Monitor. The effective address calculation done for it could fail however.

4.2.4 TOPS–20 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.4. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses the TOPS–20 Monitor.[55] The previous section presents equivalent information for the TOPS–10 Monitor. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–20 or TOPS–10, and are described in §4.2.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt instructions reference executive virtual address space. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the microcode to carry out the mapping procedure, retrieve interrupt instructions, and handle traps, halts and UU0s.

NOTE

> Hardware paging operations are inextricably intertwined with the activities of the Monitor. The reader must be familiar with both to be able to understand either fully.

Paging

All of memory both physical and virtual is divided into pages of 512 words each. Physical memory can contain 1024 pages; its locations are specified by 19–bit addresses, where the left ten bits (17–26) specify the page and the right nine (27–35) the location within the page. The virtual memory space addressable by a program is 512 pages and uses 18–bit addresses, where the left nine bits (18–26) are the page number. However for compatibility with extended processors, the TOPS–20 paging system regards the virtual page as composed of sections, each of 512 pages, even though the KS10 has only one such section, and its virtual addresses have no section number. The hardware maps the one–section virtual address space into a part of the physical address space by transforming the 18–bit

---

[55] For additional information on the kind of paging employed in a TOPS–20 system, refer to "Storage organization and management in TENEX", by Daniel L. Murphy, AFIPS — Conference Proceedings, Vol. 41, page 23, AFIPS Press, Montvale, NJ.

addresses into 19–bit addresses.[56] In this transformation the right nine bits of the virtual address are not altered; in other words a given location in a virtual page is the same location in the corresponding physical page. The translation maps a virtual page into a physical page by substituting a 10–bit physical page number for the 9–bit virtual page number. The mapping procedure is carried out automatically by the pager, but the page map that supplies the necessary substitutions is set up by the executive program.

Pointers to the page maps for the user and executive virtual address spaces are contained in section tables that begin at location 540 in the user and executive process tables. But in the KS10 each section table has only one entry (for section 0) at location 540. Two locations in the register file are used by the Monitor to specify the physical page numbers of the process tables. To retrieve the section pointer from a process table, the pager uses the appropriate base page number as the left ten bits of the physical address and 540 as the right nine bits. The section pointer must identify — either directly or indirectly — a physical page that contains the page map. Every pointer and mapping takes one word, and since there are 512 pages and 512 words in a page, a page map requires exactly one page.

Figure 4.9 shows the detailed organization of the process tables for both user and executive, as determined by the hardware. Any table locations not used are reserved for future use by the hardware or use by the Monitor for software functions.

Although the virtual space is always 256K by virtue of the addressing capability of the instruction and indirect word formats, the Monitor usually limits the actual address space for a given program by defining only certain pages as accessible. There is no requirement that the accessible space be continuous — it can be scattered pages. The Monitor also specifies whether each page is writable or not and cacheable or not.[57] To determine the mapping for a given virtual page, the microcode carries out a pointer evaluation procedure that starts with the section pointer. If it is discovered during this procedure that the page is inaccessible, the page map or the referenced page is not in memory, or the program is attempting to write in a write–protected page, the microcode traps to the Monitor, which must handle the situation. A trap to the Monitor for a reason of this sort is produced by generating a "soft page failure." But if nothing is amiss, the procedure is carried out entirely by the microcode — with no need to call the software — and it generates the mapping for the specified virtual page. The procedure requires access to the page map, to a memory status table in which the microcode keeps track of the use made of the page map and the program–referenced page, and perhaps to other predefined or software–defined tables as well. If the complete procedure were carried out in every instance, the processor would require at least two memory references for every one by the program. To avoid this, each mapping generated by the procedure is placed in a page table, and the pager makes its virtual–to–physical translations from the mappings held in the table.[58] Hence it is necessary to go through the evaluation procedure only when the reference cannot be made from the page table. Since the objective of the procedure is to place a mapping in the table, it is referred to as a "page refill."

Page Table A location in the page table contains a mapping entry in this format.[59]

---

[56] The mapping procedure is of course applied only to storage module references, whether cached or not. AC references, which can be made by any program, even when virtual page 0 is accessible, are made directly to fast memory and require no mapping.

[57] Again for consistency with extended processors, the Monitor can make the section (i.e., the whole virtual space) inaccessible, unwritable or uncacheable, but is rather unlikely to do so.

[58] In the evaluations the microcode does carry out, it generally does not need to access a process table for a section pointer, as it keeps copies of the current pointers in the workspace.

[59] In the engineering drawings and even in some Monitor documents, the M bit is labeled "writeable", which name is consistent with its use with the TOPS–10 Monitor.

Figure 4.9: KS10 TOPS–20 Process Table Configuration

User Process Table

0 420	Reserved	
421	User Arithmetic Overflow Trap Instruction	
422	User Pushdown Overflow Trap Instruction	
423	User Trap 3 Trap Instruction	
424	MUUO Flags	MUUO Op Code, A
425	MUUO Old PC	
426	E of MUUO	
427	MUUO Process Context Word	
430	Executive No Trap MUUO New PC	
431	Executive Trap MUUO New PC	
432	Reserved	
433	Reserved	
434	User No Trap MUUO New PC	
435	User Trap MUUO New PC	
436 477	Reserved	
500	Page Fail Word	
501	Page Fail Flags	
502	Page Fail Old PC	
503	Page Fail New PC	
504 537	Reserved	
540	User Section 0 Pointer	
541 777	Reserved	

Executive Process Table

0 41	Reserved
42 57	Priority Interrupt Instructions
60 100	Reserved
101	Adapter 1 Interrupt Table Pointer
102	Reserved
103	Adapter 3 Interrupt Table Pointer
104 420	Reserved
421	Executive Arithmetic Overflow Trap Instruction
422	Executive Pushdown Overflow Trap Instruction
423	Executive Trap 3 Trap Instruction
424 537	Reserved
540	Executive Section 0 Pointer
541 777	Reserved

M	*C*	Physical Page Address Bits 17–26

Each entry is identified as providing the physical page number for the translation for a particular virtual page. The properties represented by 1s in the two "page use" bits are as follows.

Bit	*Meaning of a 1 in the Bit*
M	Modified — and therefore writable without further ado. A refill produces a 1 in this bit if the page has already been modified or the reference that caused the refill is for write and the page is writable. A 0 does not imply that the page is write–protected, but simply that if a write reference occurs, the pager must find out if it can be written. Throughout this discussion, "write reference" means any reference involving writing; "read reference" means read only.
C	Cacheable.

The page table has 512 locations, one for each virtual page number. Besides a mapping for the virtual page that identifies it, each location has a parity bit, a bit that indicates whether the mapping is for user or executive address space, and a bit that indicates whether the entry is valid. A zero mapping is perfectly valid, but a location is labeled as containing no valid mapping by clearing it, thus clearing the valid bit.

When the program references a page whose mapping entry is tagged as valid and in the program address space, the 10–bit physical number[60] from the mapping for the virtual page is used as the left ten bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *M* bit). If however the entry is invalid or is not in the correct address space, or the reference is for writing and *M* is 0, the pager does a refill to get or revise the mapping for the specified virtual page from the page map. The result of the refill is placed in the table unless the reference fails because the page is inaccessible or the program is attempting to write in a protected page.

Page Refill

The refill of a mapping into the page table is accomplished by evaluating various types of pointers found in several kinds of tables. At some point in the procedure the microcode must encounter a "age address" that identifies the page map for the section, and it must end with a page address that identifies the physical page corresponding to the referenced virtual page. A page address has this format.

Storage Medium	Reserved	Page Number
12 – 17	18 – 22	23 – 35

If bits 12–17 are zero, the storage medium is memory: i.e., bits 23–35 supply the number of a page[61]

---

[60] Actually table locations have eleven bits for physical numbers, but the most significant is not used.

[61] All pointers have provision for 13-bit physical page numbers (as in the KL10), but the microcode uses only the

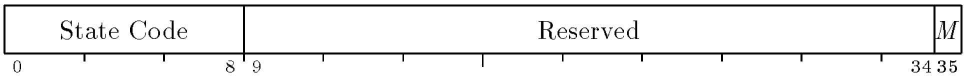
that is in memory. If bits 12–17 are nonzero, the page exists but is stored on some other medium — perhaps the disk — and the microcode traps to the Monitor. A page address may be contained in a pointer, in which case some of the bits at its left have defined uses. But when the page address stands alone, bits 0–11 of the word containing it can be used arbitrarily by the software.

Special Tables. Besides the section tables in the process tables, a refill makes use of two predefined tables: the special page–address table (SPT) and the (core) memory status table (CST). These are software–determined tables in memory, but their base addresses are held in the workspace, rather than in the register file like those of the process tables.[62]

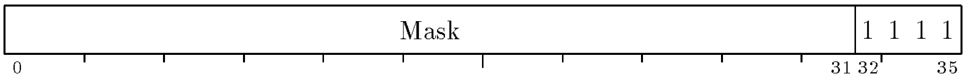
The special page–address table contains page addresses that specify shared pages or special pages (e.g. those used as page maps or other software–defined tables). The microcode accesses specific entries in the SPT by indexing on the physical base address (bits 17–35). The pointer format provides for an index of eighteen bits, so the SPT can actually be as large as 256K (and it need not start on a page boundary).

Information about the use made by programs of the various physical pages is kept in the memory status table. In every refill, the microcode updates CST entries for both the page containing the page map and the page referenced by the program. The entry for a page is a full word, and is accessed by adding the page number to the base address. If memory is fully implemented at 1024 pages, the CST occupies two of them, but need not begin on a page boundary. Note that the microcode does not manipulate CST entries for the process tables, the SPT, nor the CST itself, unless they are actually referenced by the program — in other words, unless the refill is being performed for a program reference to one of the tables.

The status of a physical page in memory is indicated by a CST entry in this format.



The Monitor keeps a state code in bits 0–8 of the entry. State codes smaller than 10 (i.e., bits 0–5 being zero) cause an age trap to the Monitor. Other codes represent the page age, which must be greater than 7 for the page to be usable, whether it is the program–referenced page or the page map.[63] The microcode updates the entry by anding a CST mask word into it and oring a process use word into that result. These two words are also held in the workspace. Bits 32–35 in them must be all 1s or all 0s as illustrated in order to preserve hardware information.



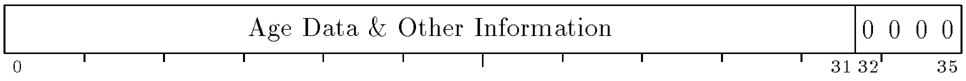
CST Mask Word

---

right ten bits.

[62] Remember that all memory tables defined by the pager are in physical address space. i.e., they have physical base addresses. Of course, to load or access a table, the Monitor must use paged virtual addresses. Note that if the base address is limited to a page number (bits 17–26), the table must begin at a page boundary.

[63] Zero age usually means the page is being swapped in and is not yet available for reference. The Monitor can use part of a CST entry to record which processes use the page.

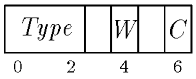


Process Use Word

A 1 in the *M* bit indicates the page has been modified since being brought into memory.[64] The microcode sets this bit in the entry for the referenced page — not that for the page map — if the reference is write and the page is writable.

Indirect pointers make use of tables whose locations are defined entirely by the Monitor. In a single refill, these may include one or more secondary section tables or page maps. Each such table or map is determined by a page address and a 9–bit index, and is therefore a single page. Memory status is kept only for the page maps.

Pointers. The microcode evaluates two kinds of pointers: section pointers and map pointers. The former are used in section tables and the latter in page maps. Members of these two classes are identical in form but differ enough in function so they must be treated separately. There are four types of section and map pointers distinguished by a type code in bits 0–2; of these, three are access pointers, i.e., they allow access to the given section or page. An access pointer has this format in its left seven bits.



Every access pointer must have use bits for the section or page it represents. These bits, *W* and *C*, indicate whether the section or page is writable or cacheable. Throughout the evaluation procedure the microcode effectively ands these bits from one pointer to the next, so the final result requires that the given characteristics be specified at every step. In other words if W is 1 in the final pointer for the mapping, the page is writable provided the entire section was also specified as writable by the original section pointer, and "writability" has been specified by every other pointer encountered along the way. Every access pointer must also either contain a page address or point to an SPT location that contains a page address.

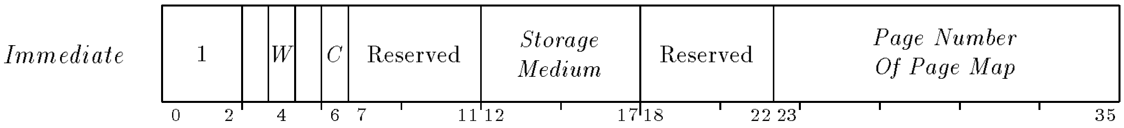
Section Pointers. Entries in a section table are of these four types.[65]

The section is inaccessible.

---

[64] At the completion of a process, the Monitor checks the CST to determine which pages have been modified and must be rewritten on the disk.

[65] Type codes 4–7 are undefined and result in a page failure.

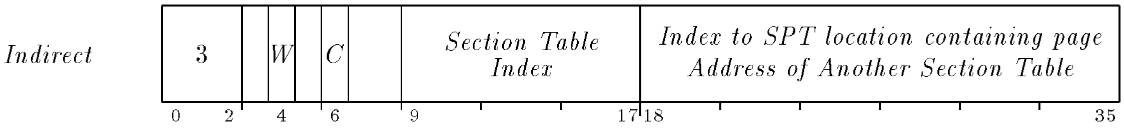


If bits 12–17 are zero, the page map is in the page specified by bits 26–35. Otherwise the page map is not in memory.

An immediate pointer contains the page address of the page map.

The page address of the page map is in the SPT at the location specified by bits 18–35.

This pointer is used for a page map shared by a number of processes. Switching to another map requires changing only the common SPT entry.

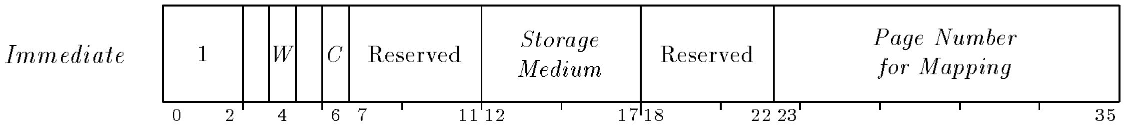


In the SPT location specified by bits 18–35 is the page address of a secondary section table. The next section pointer to be evaluated is in that table at the location specified by bits 9–17.

Indirect pointers are used for Monitor reference to per–job and preprocess areas. The pointers remain while the second section table is swapped with the job or process, or the SPT entry is changed.

Map Pointers. Entries in a page map are of these four types.[65]

The page is inaccessible.

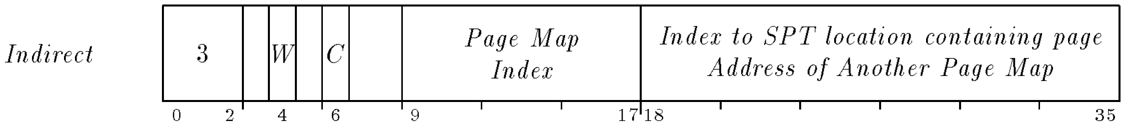


If bits 12–17 are zero, the physical page specified by bits 26–35 corresponds to the referenced virtual page. Otherwise the referenced page is not in memory.

An immediate pointer contains the page address for the mapping.

The page address for the mapping for the referenced virtual page is in the SPT at the location specified by bits 18–35.

This pointer is used for a physical page referenced as different virtual pages by different programs. The Monitor can move the page simply by changing the SPT entry.

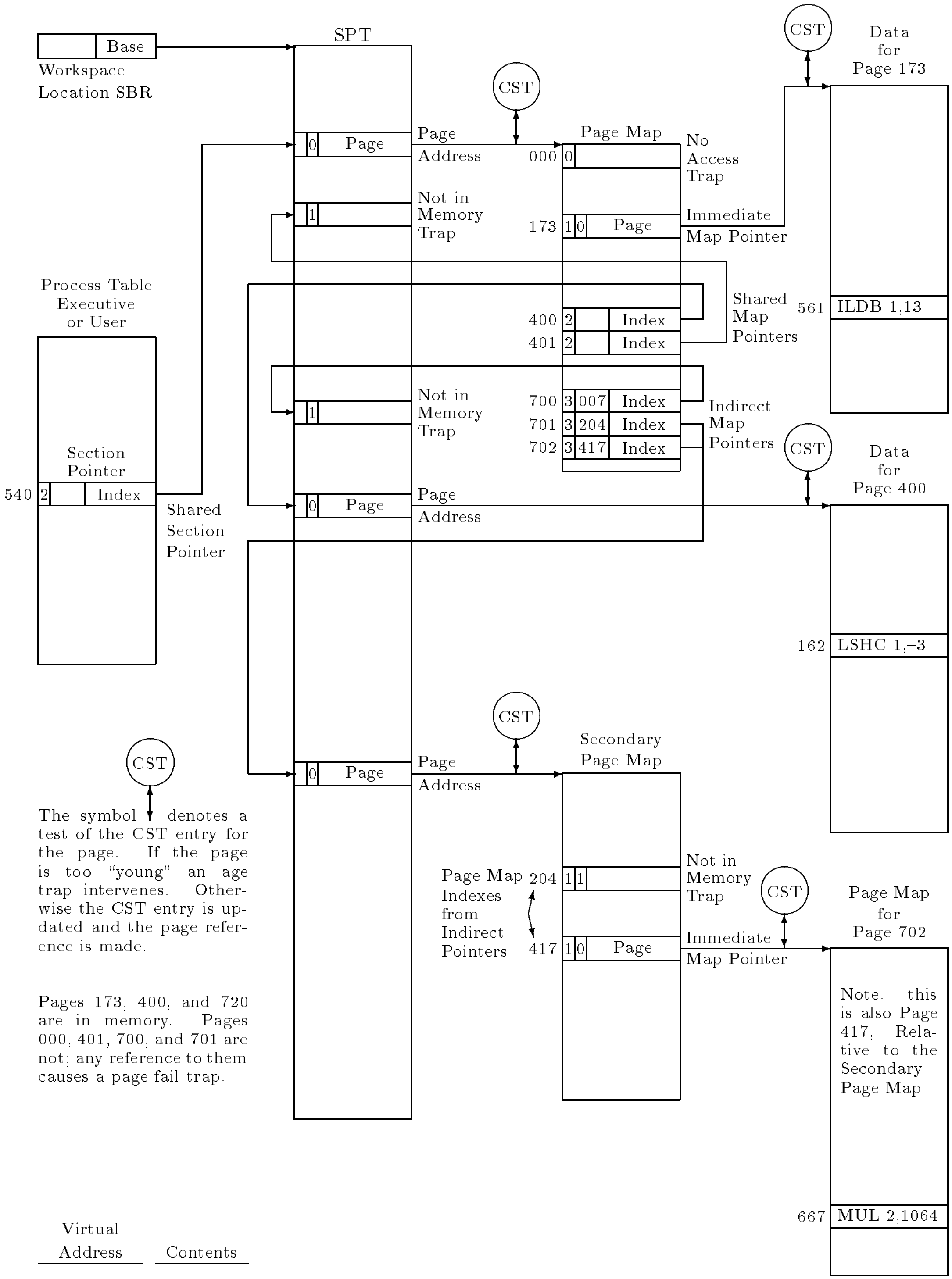


In the SPT location specified by bits 18–35 is the page address of a secondary page map. The next map pointer to be evaluated is in that map at the location specified by bits 9–17.

Refill Procedure. If the page table lacks a valid mapping for a reference, the pager must evaluate section and map pointers to get the desired mapping. The procedure begins with the pointer for the section from the process table, and the pager follows the trail laid by the various pointers, as illustrated in Figure 4.10. At any step the microcode traps to the Monitor if it encounters a no–access pointer or a page address that indicates the page is not in memory. The first part of the procedure, which may go to the SPT or indirectly through it to other section tables, evaluates section pointers to arrive at the page address of the page map. Using this physical page number as the left ten bits of an address and the number of the referenced virtual page as the right nine bits, the second part of the procedure retrieves a map pointer and evaluates it. This part may also go to the SPT or indirectly through it to other page maps to arrive at a page address for the mapping. Unless an age trap intervenes, memory status is updated along the way for any page maps used. If the reference can be made and there is no age trap for the referenced page, its status is updated including setting the M bit if the program is writing. The microcode then constructs the desired mapping, places it in the page table, and returns to the waiting reference.

The mapping data is constructed from the result of the pointer evaluation, including the running evaluation of the use bits, and has the format illustrated in the discussion of the page table. The

Figure 4.10: TOPS–20 Paging Pointer Evaluation (KS10)



The symbol denotes a test of the CST entry for the page. If the page is too "young" an age trap intervenes. Otherwise the CST entry is updated and the page reference is made.

Pages 173, 400, and 720 are in memory. Pages 000, 401, 700, and 701 are not; any reference to them causes a page fail trap.

Virtual Address	Contents
173561	ILDB 1,13
400162	LSHC 1,–3
702667	MUL 2,1064

microcode always places a 1 in the valid bit to indicate that the virtual page is accessible and this is a valid mapping for it. C is simply the result of anding the C bits of the various pointers. M however is not. A refill sets up M according to the type of reference and the characteristics of the referenced page.

Circumstances[66]	M	*Effect*
Read reference, page not writable.	0	An attempt to write will fail.
Read reference, page writable but not yet modified (according to CST).	0	An attempt to write will succeed, after the mapping is revised.
Page writable, write reference or page already modified.	1	Sets M in CST entry: an attempt to write will succeed.

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the microcode terminates the instruction immediately, without disturbing PC or storing any results in memory or the accumulators, and executes a page fail trap.[67] The trap operation makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current flag–PC doubleword in locations 501 and 502, sets up PC according to a new value in location 503, and clears the flags (placing the processor in executive mode). The processor then resumes operation in the new state at the location now addressed by PC. The same sequence of events occurs if the processor performs an I/O instruction and the adapter fails to indicate the transfer was accomplished.

There are two kinds of page failures, hard and soft. A hard failure means that something really is amiss, whereas a soft failure generally means only that the program requires some kind of service from the Monitor. A hard failure is indicated by a 1 in bit 1 of the page fail word, and the particular failure is specified by a code (which is therefore ≥ 20) in bits 1–5 There are three such failures of which two are true page failures, i.e., failures involving memory reference, and for these the page fail word has this format.

U	36 or 37	0	P	0	Address
0	1–5	6–7	8	9–16	17–35

Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a physical address was given for the reference. The code names the particular failure as follows.

36 Uncorrectable memory error — in a processor reference the memory controller has read an incorrect word from storage and was unable to correct it. The processor has saved the word in AC 0 and AC 1, block 7, and has set the Bad Memory flag (RDAPR bit 28).

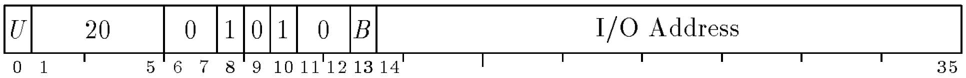
37 Nonexistent memory — the processor has called for a storage reference over the bus but the memory controller did not respond. This error also sets the No Memory flag (RDAPR bit 27).

---

[66] The missing circumstance produces a page failure.

[67] A page failure that occurs during an interrupt instruction does not act this way. Instead the processor halts (§4.2.7).

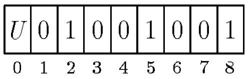
If the failure code is 20, the fail word instead has this format



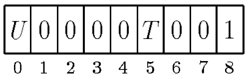
and indicates a nonexistent I/O register, i.e., an I/O instruction gave an I/O address to which there was no response. A 1 in bit 13 indicates a byte operation. (The 1s in bits 8 and 10 mean a physical reference and an I/O function on the bus.) Note that this is not an I/O page failure, which is a true (memory) page failure and causes a halt.

A soft failure can result only from actions taken in a refill or writability check and is indicated by a 0 in bit 1. This means either an attempt to write in a write–protected page, or the evaluation procedure encountered some condition beyond which it could not go — a no-access pointer, an illegal pointer code, some page (not necessarily the program–referenced one) not in memory, or an age trap. The fail word still contains the *U* bit and the virtual address, but now bits 1–8 have one of these formats,

Write Violation



Other Failure



where bit 8 is the inverse of bit 8 in the hard case (1 means virtual), and *T* indicates the type of reference in which the failure occurred: 0 for a read–only reference, 1 for any reference involving writing. A 0 in bit 2 means the evaluation procedure was incomplete. In the write violation configuration, the 1 in bit 2 means the procedure was completed, and the 0 in bit 4 comes from anding the *W* bits in the string of pointers. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course *T* and bit 2 both being 1 implies a write failure.

For a page fail trap, the processor automatically switches to executive mode. After rectifying the situation, the Monitor eventually returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multi part instruction. Even a two–part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §4.2.1. Before returning to the failed instruction, the Monitor must invalidate the mapping for the page and revise the pointers for the new situation. Then when the instruction is restarted, the pager will do a refill to get the new, correct mapping.

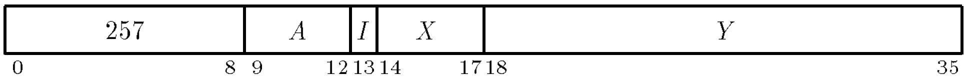
A no–access pointer may imply that the page simply does not exist. Otherwise a soft failure seldom implies that anything is “wrong.” Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in memory. When the user attempts to gain access to a page that is not there (i.e., for which the refill encounters a not–in–memory page address), the Monitor would respond to the failure by bringing in the needed page from the disk, either adding to the user space, or swapping out a page the user no longer needs or has not used recently. Similarly a process using several sections may have only one in core at a time. While swapping is in progress, the Monitor runs some other user. returning to the interrupted job when the requested page is available.

The same situation exists for writability. Keeping track of modified pages is handled by the refill procedure using the memory status table. But a page may be write–protected because is it shared by a number of processes, wherein a change made by one might not be wanted by the others. Thus in response to a write failure, the Monitor might make a separate writable copy of the page for the sole use of the process that wishes to modify it.

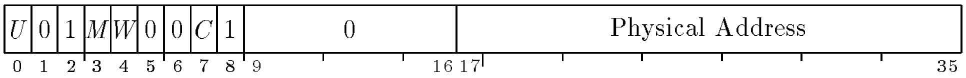
The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory. For such purposes the processor has this instruction.

MAP Map an Address



If the pager is on, map the page number of the virtual effective address E and place the resulting physical address and other map data in AC. If the page is accessible, the information loaded into AC is of the form



where bits 17–26 are the physical page number the pager supplies for E, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and M, W and C are page use bits resulting from the pointer evaluation procedure as explained above. If the page is inaccessible, AC receives the given virtual address in place of a physical address; the word also includes U and a 1 in bit 8, but the remaining bits are all zero.

However, should a memory error occur during the refill, AC receives a hard page fail word. If the pager is off, the result is undefined.

Notes. The instruction cannot actually fail, because regardless of what happens, the page fail microcode returns to it instead of trapping to the Monitor. The effective address calculation done for it could fail however.

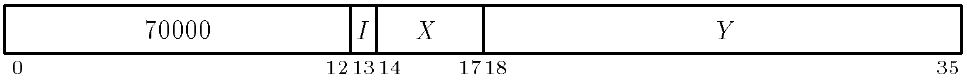
4.2.5 Memory Management

In order properly to manage memory, the executive program must select the kind of paging, set up process tables and page maps for itself and the various users, oversee the operation of the page table, and select the fast memory block to be used by each program (usually block 0 for itself). At any given time, accumulator, index register and fast memory references are made to that AC block that is assigned as "current." Given a particular processor mode and an appropriate process table and page map, the Monitor effectively defines the address space for a process (which may be itself) by specifying the base address for the process table and selecting the current AC block.

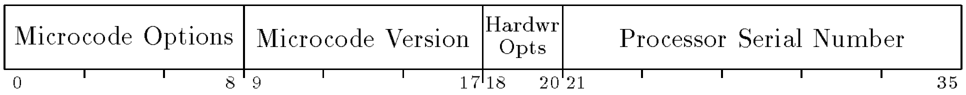
When a user program calls the Monitor it is usually to request some activity, which may often require the executive to gain access to the user address space. To facilitate the crossover from one address space to another, the same instruction through which the Monitor assigns its own current AC block also allows assignment of an AC block for the "previous–context" — i.e., the context of the process that made the call. This, together with a flag that indicates the mode of the caller, allows execution of instructions in the previous–context (more about this subject later). At any point in time, the previous–context is essentially the circumstances in which the previous process was running. Note that the previous–context need not be the user; the same techniques can be exploited following a call from one level of the Monitor to another.

For initial setup, the executive program must be cognizant of certain fundamental characteristics that can vary from one system to another. For this purpose the instructions for basic management include not only those that control the pager, but also one that addresses the processor to discover what those characteristics are. The first five of the following instructions are for either kind of paging; the remaining eight are solely for handling the special registers used in the TOPS–20 pointer evaluation.

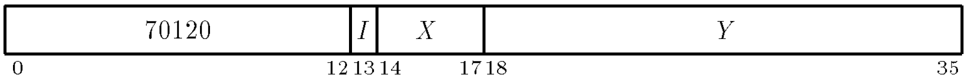
APRID Arithmetic Processor Identification



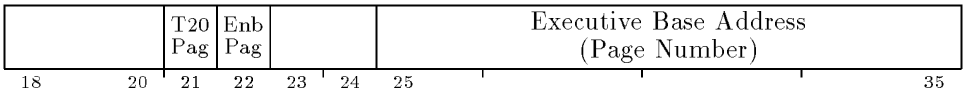
Read the microcode version number, the processor serial number, and a listing of the fundamental characteristics of the system into location E as shown. At present there are no microcode or hardware options.



WREBR Write Executive Base Register



Set up the system–oriented characteristics of the pager according to the effective conditions E as shown.



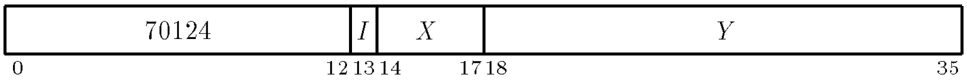
Load bits 25–35 into bits 16–26 in the executive base register (EBR in the register file) to select the executive process table. If bit 22 is 1 enable overflow trapping and enable the pager for the type of paging selected by bit 21: 1 TOPS–20, 0 TOPS–10. A 0 in bit 22 prevents traps and disables paging so all memory references are to physical locations unpaged.[68]

CAUTION

> Paging can be disabled only for executive mode. A user mode program will not run correctly unless the pager is turned on.

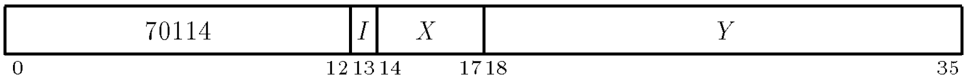
Invalidate the entire cache and page table by clearing the valid bits in all entries.

RDEBR **Read Executive Base Register**

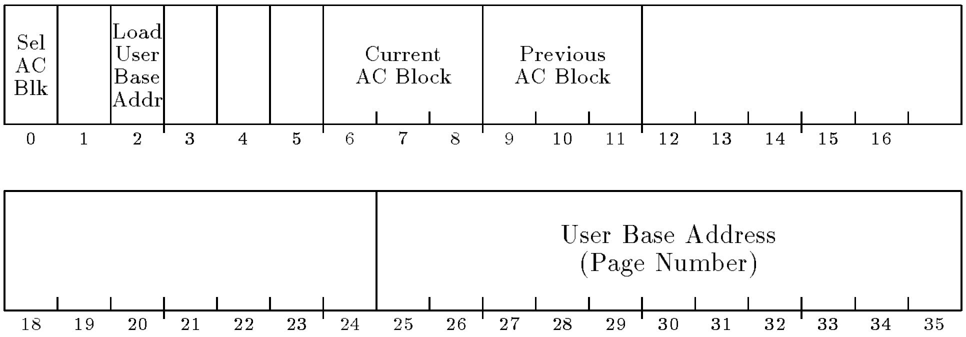


Read the system status of the pager into the right half of location E. The information read is the same as that supplied by WREBR.

WRUBR **Write User Base Register**



Set up the process–oriented elements of the pager according to the contents of location *E* as shown.



Bits 0 and 2 are change indicators for parts of the data word: when a bit is 0, the corresponding part of the word is ignored, and the equivalent value supplied by a previous WRUBR remains in effect.

If bit 0 is 1, select as the current and previous–context AC blocks those specified by bits 6–8 and 9–11, respectively. If bit 2 is 1, load bits 25–35 into bits 16–26 in the user base register (UBR in the register file) to select the user process table, and invalidate the entire cache and page table by clearing the valid bits in all entries.

---

[68] Note that disabling the pager does not mean there can be no page failures, as these can be caused by conditions having nothing to do with paging, i.e., with translating virtual to physical addresses.

RDUBR Read User Base Register

70104	I	X	Y
0 12	13	14 17	18 35

Read the process status of the pager into location E. The information read is the same as that supplied by a WRUBR (bits 0 and 2 are 1s).

CLRPT Clear Page Table Entry

70110	I	X	Y
0 12	13	14 17	18 35

Invalidate the page table mapping entry for the page referenced by E, and invalidate the entire cache.

WRSPB Write SPT Base Address

70240	I	X	Y
0 12	13	14 17	18 35

Load the contents of location E into the SPT base register in the workspace.

RDSPB Read SPT Base Address

70200	I	X	Y
0 12	13	14 17	18 35

Read the contents of the SPT base register into location E.

WRCSB Write CST Base Address

70244	I	X	Y
0 12	13	14 17	18 35

Load the contents of location E into the CST base register in the workspace.

RDCSB Read CST Base Address

70204	I	X	Y
0 12	13	14 17	18 35

Read the contents of the CST base register into location E.

WRCSTM **Write CST Mask**

70254	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location E into the CST mask register in the workspace for use as the mask in CST updating.

RDCSTM **Read CST Mask**

70214	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the CST mask register into location E.

WRPUR **Write Process Use Register**

70250	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location E into the process use register in the workspace for use as the process use word in CST updating.

RDPUR **Read Process Use Register**

70210	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the process use register into location E.

At power turn on the contents of the cache and page table are indeterminate, the processor is in executive mode, paging is disabled, and the current AC block is 0. After the console loads the microcode, it then loads the initializing executive program. This program, running unpaged in physical memory, should give an APRID to determine system characteristics. The unpaged program ends with a WREBR that selects and enables paging, specifies the executive base address, and invalidates the cache and page table. From this point the executive program runs paged and must set up the first user or users, loading the user process tables and page maps, and bringing in whatever parts of user programs and data that are consistent with good working–set management. Finally the Monitor gives a WRUBR to assign the base address and current AC block for the first user, and then transfers control to the user program via an XJRSTF or JRSTF.

On a call from the user via an MUUO, give an RDUBR to determine the context of the user, i.e., his AC block. Then give a WRUBR that assigns block 0 as current for the Monitor, assigns the user AC block as previous–context for accessing user space, but leaves the base address alone so the right paging is still available for such access. To return to the same user, reassign the AC block without changing the base address. Note that on the transfer to a user program no previous–context AC block need be given as the user cannot employ PXCTs.

The usual procedure for administering AC blocks is to assing block 1 to all users and assign two or three blocks for the sole use of interrupt routines. Suppose the assignments are: block 0 for the Monitor, block 1 for all users, block 2 for the highest priority interrupt level, block 3 for the second highest level, and block 4 for all other levels. Then in no circumstances is it necessary to determine which block to save, and interrupt routines on the highest, second highest and lowest levels need not save any. Moreover, the Monitor need not even store block 1 when it takes control from a user temporarily. When switching from one ordinary user to another, the Monitor usually stores the first user's accumulators in his process table or shadow area — this is locations 0–17 in user virtual page 0, an area not generally accessible to the user at all — and loads the new user's accumulators from his process table or shadow area, where they were stored after the last time the new user ran

On a change from one process to another the entire page table must be invalidated, but this is done automatically by the instruction that assigns the new user base address. If the system uses shared or indirect pointers, or several virtual page numbers point to the same physical page, then the table must be invalidated whenever a page is removed from memory or a pointer is removed from a user page map. On the other hand deletion of a page with a unique mapping requires only that a CLRPT be given to invalidate the entry containing it.

Previous–Context Execute

Ordinarily an instruction in a user program is performed entirely in user address space, and an instruction in the executive program is performed entirely in executive address space. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous–context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero *A* bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous–context. A PXCT can be given only in executive mode, but the previous–context may be the user, as following a call to the Monitor by the user. The previous context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. (Note: it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.)

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory and index register references are made. To work as a PXCT, an XCTa must be given in executive mode, and the bits in its *A* field (9–12) must not all be 0 (in user mode *A* is ignored). But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction, all accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a fast memory location.) If the instruction makes no memory operand references, as in a shift or immediate mode instruction, and it has no indexing or indirection (i.e., the instruction word gives *E* directly), then its execution differs in no way from the normal case. The only difference is in memory and index register references.

The previous–context is specified by two quantities. Following a call by an MUUO, the fast memory block assigned to the calling program appears as the current context AC block in the word read by an RDUBR. For the called program, this value can then be assigned as the previous–context by a WRUBR. The current AC block of the calling program also appears in the process context word supplied by the MUUO. Various levels of the Monitor may all use fast memory block 0; or a separate block may be assigned to that part of the Monitor that uses PXCTs in handling MUUO calls from other parts of the Monitor.

Just as the current mode is indicated by the User flag, the mode in which the calling program was running is indicated by Previous Context User.[69] At a call this flag may be set up automatically or it may be set up by a flag–PC doubleword or a PC word. Note that the restrictions on references made in the previous–context are those of the previous context — not those of the context in which the PXCT is given. Suppose the executive executes an instruction that references an inaccessible user area. Such a reference would fail.

Which references in the executed instruction are made in the previous–context is determined by 1s in the A portion of the PXCT instruction word as follows.

Bit	*References Made in Previous–Context if Bit is 1*
9	Effective address calculation of instruction, including both instruction words in EXTEND (index registers, address words by indirection); also EXTEND effective address calculation of source pointer if bit 11 is 1 and of destination pointer if bit 12 is 1.
10	Memory operands specified by E, whether fetch or store (e.g., PUSH source, POP or BLT destination); byte pointer: second instruction word in EXTEND.
11	Effective address calculation of byte pointer; source in EXTEND; effective address calculation of EXTEND source pointer if bit 9 is 1.
12	Byte data; stack in PUSH or POP; source in BLT; destination in EXTEND; effective address calculation of EXTEND destination pointer if bit 9 is 1.

Previous–context referencing is useful and reasonable in some instructions but inapplicable to others. There is no trap of any kind, and the effect of using the feature with an instruction to which it does not apply is simply undefined.

---

[69] Previous Context User is in the same flag bit that is used for User In–out, which has no meaning in executive mode.

Applicable	*Inapplicable*
Move, XMOVEI	LUUO, MUUO
EXCH, BLT, XBLT	AOBJN, AOBJP
Half word, XHLLI	JUMP, AOJ, SOJ
Arithmetic	JSR, JSP, JSA, JRA, JRST
Boolean	PUSHJ, POPJ
Double move	XCT, PXCT
CAI, CAM	Shift–rotate
SKIP, AOS, SOS	String (except MOVSLJ)
Logical test	I/O
PUSH, POP, ADJSP	System (except MAP)
Byte	
MOVSLJ	
MAP	

Note that no jumps can use previous–context referencing. Even among the instructions to which such referencing is applicable, only a limited number of the sixteen possible bit combinations is useful or meaningful. Doing an effective address calculation in the previous–context (selected by bit 9 or 11) makes sense only if the corresponding data access is also in the previous–context (as selected by bit 10 or 12 except 11 or 12 in EXTEND). Only the combinations listed in Table 4.2 are permitted.

The most frequent use of previous–context referencing is simply for the transfer of words between user and executive. For this reason the processor has these two convenient instructions.

UMOVE User Move

704	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the same function as PXCT 4,[MOVE A,E]. However, whereas a PXCT can be performed only in executive mode, UMOVE can also be done in user in–out mode.

UMOVEM User Move to Memory

705	*A*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

Perform the same function as PXCT 4,[MOVEM A,E]. However, whereas a PXCT can be performed only in executive mode, UMOVEM can also be done in user in–out mode.

4.2.6 System Timing

The timer includes a 12–bit hardware millisecond counter, a doubleword time base kept from it, and an interval register for timed interrupts. The millisecond counter runs continuously at 4.1 MHz and represents an elapsed time of just under 1 ms at each overflow. Whenever the counter is read, its two least significant bits are ignored, so its contents effectively represent a count in microseconds

Table 4.2: KS10 Permissible PXCT Addressing Modes

Instructions	9	10	11	12	*References in Previous-Context*
General	0	1	0	0	Data
	1	1	0	0	E, Data
Immediate	1	0	0	0	E
BLT	0	0	0	1	Source
	0	1	0	0	Destination
	0	1	0	1	Source, destination
	1	1	0	0	E, destination
	1	1	0	1	E, source, destination
XBLT	0	0	1	0	Source
	0	1	0	0	Destination
	0	0	1	1	Source, destination
Stack	0	0	0	1	Stack
	0	1	0	0	Memory data
	0	1	0	1	Memory data, stack
	1	1	0	0	E, memory data
	1	1	0	1	E, memory data, stack
Byte	0	0	0	1	Data
	0	0	1	1	Pointer E, data
	0	1	1	1	Pointer, pointer E, data
	1	1	1	1	E, pointer, pointer E, data
MOVSLJ	0	0	0	1	Destination
	1	0	0	1	$E(=Y)$, destination pointer, destination
	0	0	1	0	Source
	1	0	1	0	$E(=Y)$, source pointer, source
	0	0	1	1	Source, destination
	1	0	1	1	$E(=Y)$, pointers, source, destination

(1/1025th ms).

The time base is a double length number kept in a pair of registers in the workspace. It is a 71–bit unsigned quantity in which the entire first word comprises the high order thirty–six bits, and the low order thirty–five are in bits 1–35 of the second word.[70] In this doubleword, the hardware counter corresponds to the right twelve bits of the low order word. The program can initialize the time base as a number of milliseconds (the low order twelve bits are ignored), and every time the counter overflows the microcode adds 2^{12} to the base.

The interval register (in the workspace) holds a period that is specified by the program and corresponds in magnitude to the low order word of the time base. This allows a maximum interval of 2^{23} ms, which is almost 140 minutes. At the end of each interval, the microcode sets Interval Done (RDAPR bit 30), requesting an interrupt on the level assigned to the system flags (§4.2.8). In a separate workspace register, the microcode starts with the given period, decrements it by 2^{12} every time the millisecond counter overflows, and sets the flag when the contents of this "time to go" register reach zero or less. Hence the countdown is by milliseconds, and any nonzero quantity in the low order twelve bits of the given period adds a whole millisecond to the count. (However, following specification of an interval by the program, the first downcount occurs at the first counter overflow regardless of when the register was loaded.)

The processor has these instructions for the program to handle the time base and the interrupt interval.

WRTIM Write Time Base

70260	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of location E, $E{+}1$, clear the right twelve bits of the low order word read (the part corresponding to the hardware millisecond counter), and place the result in the time base registers in the workspace.

RDTIM Read Time Base

70220	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the time base registers, add the current contents of the millisecond counter to the doubleword read, and place the result in location $E, E + 1$.

WRINT Write Interval

70264	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location E into the interval register in the workspace.

---

[70] Remember, it is a property of twos complement arithmetic that the sign can be used as an extra magnitude bit in an unsigned number. But since the hardware is set up for signed arithmetic, bit 0 of any lower order word must be skipped.

RDINT **Read Interval**

70224	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the interval register into location E. The period read is the same as that supplied by WRINT.

4.2.7 Halt Status

Whenever the processor halts, the microcode places a halt code, giving the reason for the halt, in physical (i.e., storage) location 0, and places PC in physical location 1. Except at error–free powerup, it then saves the register file and VMA in a halt status block beginning at a physical location specified by the program, although the program can inhibit storing of halt status altogether. The registers saved in the status block are as follows.

Location	*Register*
0	MAG
1	PC
2	HR
3	AR
4	ARX
5	BR
6	BRX
7	ONE (1)
10	EBR
11	UBR
12	MASK
13	FLG (flags, page fail code)
14	PI
15	XWD1 (1,,1)
16	TO
17	TI
20	VMA (with flags)

Halt codes in the range 0–77 are used for “normal” halts. Codes in the ranges 100–777 and 1000 or greater respectively indicate software and microcode/hardware failures. Codes currently assigned are these.

Code	*Halt Condition*
0	Microcode just started; on this halt no status block is stored
1	Program gave a HALT (AR and PC contain E)
2	Console halted the processor
100	page failure
101	Illegal interrupt instruction If halt occurs on a vector interrupt, status block contains these quantities: TO Vector as read from bus ARX EPT address + 100 + adapter number BR Address of illegal instruction BRX Vector masked and shifted
102	Zero table pointer for vector interrupt (for contents of TO and ARX, see code 101)
1000	Error in BWRITE dispatch on dispatch ROM
1005	In powerup sequence, processor got wrong result when computing table of powers of 10 for use by string microcode (BR and ARX contain high and low words of incorrect 10^{21})

At powerup the microcode assigns an address of 376000 for storing halt status. The program can change the assignment at any time using these instructions.

WRHSB Write Halt Status Block Base Address

70270	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of location E into the halt status block base register in the workspace. If bit 0 of the word in E is 0, bits 17–35 will be used as the physical address for storing halt status. But if bit 0 is 1, no status will be stored.

RDHSB Read Halt Status Block Base Address

70230	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the halt status block base register into location *E*.

4.2.8 System Conditions

This section discusses special logic through which the program controls and receives information about other parts of the system, specifically memory and the console. Any program also has considerable dealings with the peripheral equipment, but that is another subject.

System Flags

Four of these eight flags are set by memory hardware error conditions. Two others are used for communication between processor and console, and one is used by the microcode to signal completion of an interval count. The program can enable any flag to request an interrupt on a level assigned to them all. There are of course other error indications besides the flags. A parity error in the internal data paths of the processor causes the console to shut down the system by turning off the processor clock. Software errors in the handling of interrupts and some processor hardware failures cause the microcode to halt the processor as discussed in §4.2.7. And yet other conditions cause page failures.

The system flags are generally regarded as important enough to be assigned to the highest priority level. However for most conditions the common practice is for the interrupt to switch over to the lowest priority level by means of a program–set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

The flags are handled by these two instructions.

WRAPR Write System Flags

70020	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 20–31 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

		Enb	Dis	Clr	Set	Select Flags for Bits 20–23									Priority Interrupt Assignment		
		Selected Flags				Flag 24	Int Cons	Pwr Fail	No Mem	Bad Mem Data	Corr Mem Data	Intv Done	Cons Int				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Then after 300 ns clear the Interrupt Console flag.

Bits 20–23 select flag functions: 1s in these bits produce the indicated effects on the system flags selected by 1s in bits 24–31. A 1 in bit 20 enables the setting of any selected flag to request an interrupt on the level assigned to the flags; a 1 in bit 21 disables the selected flags from requesting interrupts. Similarly a 1 in bit 22 or 23 clears or sets the selected flags. The result of putting 1s in both bits 20 and 21 or 22 and 23 is indeterminate.

The reason for clearing Interrupt Console is to provide a pulse on the signal line to the console in case the instruction has set the Interrupt Console flag. Pulsing the line triggers an interrupt in the console microprogram.

Notes. Except for Flag 24 (which has no defined meaning) and Interrupt Console, the program setting a flag has no relation to what the flag represents — the function is used only to check out the flag logic.

RDAPR **Read System Flags**

70024	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the system flags into location E as shown (asterisks indicate bits that can cause interrupts)

	Flags Enabled to Interrupt								
	Flag 24	Int Cons	Pwr Fail	No Mem	Bad Mem Data	Corr Mem Data	Intv Done	Cons Int	
0 1 2 3 4 5	6	7	8	9	10	11	12	13	14 15 16 17

	*	*	*	*	*	*	*	*		
	Flag 24	0	Pwr Fail	No Mem	Bad Mem Data	Corr Mem Data	Intv Done	Cons Int	Int Req	Priority Interrupt Assignment
18 19 20 21 22 23	24	25	26	27	28	29	30	31	32	33 34 35

6–13 A 1 in any of these bits indicates that setting the listed flag will request an interrupt on the level assigned to the flags by bits 33–35 of the WRAPR.

24 Spare — available to the program for any purpose.

25 (Interrupt Console.) When read, this flag should always be 0, as any WRAPR that sets the Interrupt Console flag also clears it to provide a pulse on the interrupt line to the console.

26 (Power Fail.) AC power has failed. The program should execute an appropriate shut–down procedure and halt the processor. Note that PC may point to an interrupt routine rather than the main program. After power is restored the console must reboot the system, and the Monitor must reestablish the operating environment (§4.2.5).

27 (No Memory.) The processor was granted the bus for access to memory, but the memory controller did not respond within two bus cycles. This is most likely because the memory subsystem contained no array board corresponding to the address given, or there has been a refresh error. Note that this condition also produces a page failure. Since an nonexistent supplies zero data, on read this error may be accompanied by a 1 in bit 28.

28 (Bad Memory Data.) In a read reference by the processor, the word retrieved (and sent) was wrong and the memory circuits were unable to correct it. Note that this condition also causes a page failure.

29 (Corrected Memory Data.) In a read reference by the processor, the word retrieved was wrong but the memory circuits were able to correct it.

30 (Interval Done.) The microcode has completed a count of the interval specified by the program.

31	(Console Interrupt.) The console is requesting a processor interrupt.
32	(Interrupt Request.) Some system flag is currently requesting an interrupt, i.e., some flag in bits 24–31 is set and has been enabled to interrupt as indicated by a 1 in the corresponding position in bits 6–13.
33–35	The priority interrupt level assigned to the processor by WRAPR.

Programming Cautions. When handling bad data or nonexistent memory interrupts, the programmer should beware of the following.

NOTE

In general it is better not to use the interrupt for these conditions, as the page failure provides more information. Moreover, if the interrupt is used, the processor interrupts out of the page failure, which occurs first.

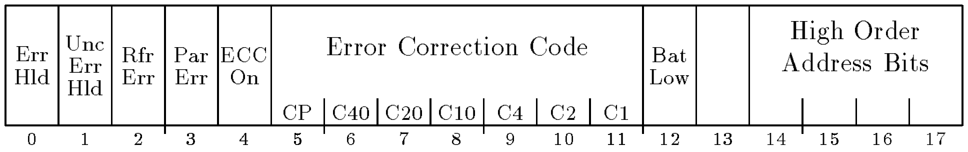
- Should an error flag be set while another interrupt request is being processed, the system would handle the lower priority interrupt before getting to the flag interrupt. This means PC may be pointing to a lower level interrupt routine rather than the program level at which the error occurred.
- Even without inadvertent interference from another level, the processor may perform another instruction between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may be pointing to the instruction following the one in which the error occurred.
- An error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second flag interrupt before the error–handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another — consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what has gone wrong.

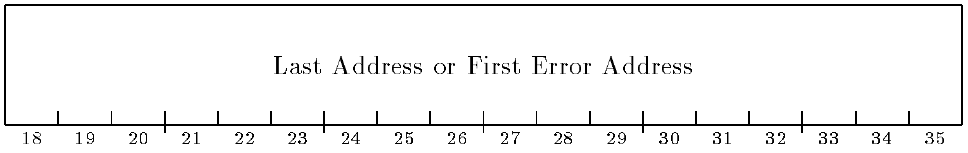
Memory Status

The memory controller reports information on error conditions by means of status that the program (or operator) can read or test using I/O instructions that address the controller (I/O address 0100000). Note that the errors reported may have nothing whatever to do with the program or processor: they may be the result of access by an adapter of the console. On every access the controller regularly loads the address and, if read, the error correction code into the status register. But if a read error (incorrect data read from the storage array) or refresh error occurs, the address and code are held — even through subsequent errors — until the processor or console writes a status word that clears the holding flag.

The remainder of this section identifies the information read as status and the functions that can be performed by writing status. For advice on how to use the information for diagnosing memory problems, the reader should turn to the maintenance documentation.

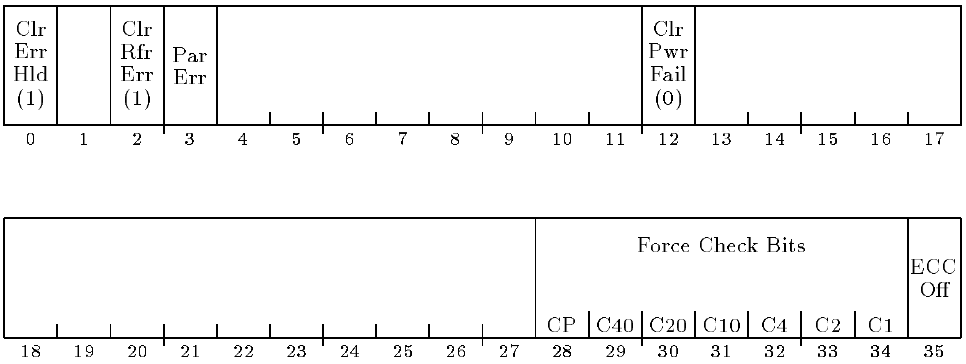
Read Status





0	The memory controller has detected a read error or a refresh error (bit 3) and has held the error correction code in bits 5–11 and the address supplied over the bus in bits 14–35.
1	The code and address are being held for a read error in which the data read was uncorrectable.
2	A refresh cycle was still not finished 10.3μs after the refresh logic requested it. The most likely cause is that the memory cycle logic was waiting for write data that failed to arrive. Setting this flag both clears and shuts down the cycle logic, so refreshing can continue but the memory is unavailable to the rest of the system until a write status clears the flag.
3	A parity error has been detected in information (command/address, data, status) received by the memory controller over the bus. This error indication is sent to the console, which may respond by turning off the processor clock.
4	The error correcting circuits are active.
5–11	This is the error correction code for the last read data access, unless bit 0 is 1, in which case it is the code for the cycle on which a read error occurred or for the last read access before a refresh error.
12	Battery backup power (if present) is low, and will not be able to sustain memory refresh in the event of an AC power failure.
14–35	This is the address supplied in the last bus transaction with memory, unless bit 0 is 1, in which case it is the address used in the data access that caused the error hold (a read address on a read error, a write address on a refresh error).

Write Status



0	A 1 in this bit clears Error Hold, which in turn clears Uncorrectable Error Hold and drops the hold on the error code and address
2	A 1 clears Refresh Error.
3	A 0 clears Parity Error, but a 1 sets it allowing checkout of the associated logic.
12	A 0 clears Power Failed.
28–34	A nonzero code forces the indication of errors where none exist, allowing checkout of the error detection and correction circuits.
35	A 1 disables the error correcting circuits. A 0 restores them to their normal, active state.

4.3 KI10 and KA10 System Operations

The information presented in this chapter is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own I/O, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

Programming for the system as a whole is programming in executive mode. In the KI10 executive mode is divided into kernel and supervisor modes. Only the kernel program is without instruction restrictions, and only it can access physical core unpaged. The supervisor program labors under the same instruction restrictions as the user and has no way of bypassing them, although it can read but not alter concealed pages (the kernel program can supply data tables to the supervisor program, and the latter cannot affect them). In the KA10 the executive program has no restrictions, and it manages protection and relocation hardware that is applicable only to the user.

The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working areas, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of the user. The KI10 kernel program accomplishes these objectives by handling all in–out for the system setting up page maps, trap locations, interrupt locations and the like for both itself and the users, keeping job accounts, and so forth. The KA10 executive program also handles in–out, job accounts and interrupts, but it manages the user working space by setting up protection and relocation registers, and it takes care of arithmetic and stack overflow via the interrupt.

Except for handling in–out, the activities of an operating system are the topics covered in this chapter. The first section, on the console, is applicable to both processors. The basic system information is covered in three sections separately for each: sections §4.3.2–§4.3.4 for the KI10, sections §4.3.5–§4.3.7 for the KA10. The last section discusses the DK10 real time clock, which is used in both. Of course the system programmer must also be quite familiar with all of the material presented in Chapters 1 and 2. In particular he must fully understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the `JRST` instruction, MUUOs, and I/O instructions (§2.9.4, §2.16, §2.18).

In several of the `CONI` bit assignment drawings in this section, bits that can cause interrupts are indicated by asterisks.

4.3.1 Console

Most console operations are entirely manual, and these are described in Appendix G.2. However the program can communicate with the console in a limited way, and the programmer must be familiar with the format and execution of the readin function.

Readin Mode

This mode of processor operation provides a means of placing information in memory without relying on a program already in memory or loading one word at a time manually. Its principal use is to read in a short loader program which is then used for loading other information. A loader program should ordinarily be used rather than readin mode, as a loader can check the validity of the information read.

Pressing the readin key on the console activates readin mode by starting the processor in a special hardware sequence that simulates a DATAI followed by a series of BLKI instructions, all of which address the device whose code is selected by the readin device switches at the left just above the console operator panel. Various devices can be used, and for each there are special rules that must be followed. But the readin mode characteristics of any particular device are treated in the discussion of the device (paper tape, DECtape, and standard magnetic tape). Here we are concerned only with the general characteristics.

The information read is a block of data (such as a loader program) preceded by a pointer for the BLKI instructions. The left half of the pointer contains the negative of the number of words in the block, the right half contains an address one less than that of the location that is to receive the first word.

To read in, the operator must set up the device he is using, set its code into the readin device switches, and press the readin key. This key function first duplicates the action of the console reset key, which clears both the processor and the in–out equipment; in particular it places the processor in executive mode, and in the KI10 selects kernel mode with executive paging disabled, so all access will be to the first 256K of physical memory unpaged. Following this the processor places the device in operation, brings the first word (the pointer) into location 0, and then reads the data block, placing the words in the locations specified by the pointer. Data can be placed anywhere in the first 256K of memory (including fast memory) except in location 0. The operation affects none of memory except location 0 and the block area.

Upon completing the block, the processor leaves readin mode and begins normal operation. This is done in the KI10 by jumping to the location containing the last word in the block, in the KA10 by executing the last word as an instruction. In the KA10 the processor stops after executing the first instruction if the single instruction switch is on.

Console–Program Communication

Neither the processor nor the priority interrupt system require all four types of I/O instructions, so the program can make use of their device codes for communicating with the console. Both processors have two instructions that transfer data between the console and program. But in the KI10, the program can actually operate some of the switches on the console. For this purpose it uses a data–out instruction with the device code for the paper tape reader (an input–only device). The KI10 program can also inspect the states of a number of operating and sense switches, but the bits for these are included in the left half words of the standard input conditions for the interrupt and processor (§4.3.2, §4.3.3).

DATAI APR, Data In, Console

70004	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the console data switches into location E.

Notes. MACRO also recognizes the mnemonic RSW (Read Switches) as equivalent to DATAI APR,.

DATAO PI, Data Out, Console

70054	*I*	*X*	*Y*
0 12	13	14 17	18 35

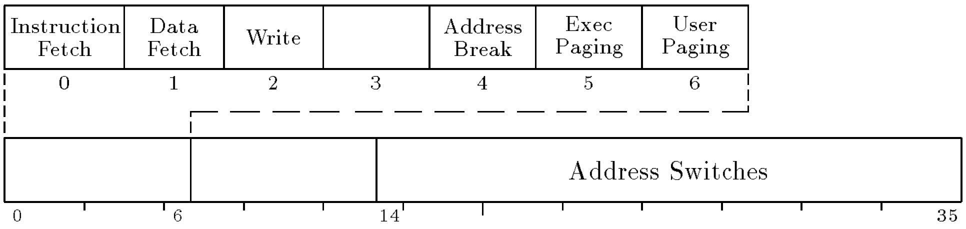
Unless the console MI program disable switch is on, display the contents of location E in the console memory indicators and turn on the triangular light beside the words PROGRAM DATA just above the indicators (turn off the light beside MEMORY DATA).

Once the indicators have been loaded by the program, no address condition selected from the console (Appendix G.2) can load them until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key.

DATAO PTR, Operating Data Out, Console

71054	*I*	*X*	*Y*
0 12	13	14 17	18 35

Unless the MI program disable switch is on, set up the console address and address–condition switches according to the contents of location E as shown (a 1 in a bit turns on the switch, a 0 turns it off).



For complete information on the use of these switches, see Appendix G.2.

Notes. On the KI10 console, all switches are pushbutton flip–flop combinations; the instruction of course controls the flip–flops, not the buttons.

4.3.2 KI10 Priority Interrupt

Most in–out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often result in loss of information and certainly results in operating the device below its maximum speed. The priority interrupt is designed with these considerations in mind, i.e., the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The hardware also allows conditions internal to the processor to signal the program by requesting an interrupt.

Interrupt requests are handled through seven levels arranged in a priority chain, with assignment of devices to levels entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35). Levels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be connected to a single level, and some can be connected to two levels (e.g., a device may signal that data is ready on one level, and that an error has occurred on another).

When a device requires service it sends an interrupt request signal over the in–out bus to its assigned level in the processor. The processor accepts the request depending upon certain conditions, such as that the level must be active (on). The request signal remains on the bus until turned off by an appropriate response from the processor: either given by the program (CONO, DATAO or DATAI, depending on the device), or generated automatically by the hardware. Thus if a request is not recognized or accepted when made, it will be when conditions are satisfied. A single level will shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request. The program can usually trigger a request from a device but delay its acceptance by turning on the level later.

The request signal is generally derived from a flag that is set by various conditions in the device. Often associated with these flags are enabling flags, where the setting of some device condition flag can request an interrupt on the assigned level only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device by a CONO. For example, a device may have half a dozen flags to indicate various internal conditions that may require service by an interrupt; by setting up the associated enabling flags, the program can determine which conditions shall actually request interrupts in any given circumstances.

Having accepted a request, the processor will do nothing further with it unless the priority interrupt system is on. But even with the system off, the processor will continue to accept requests on other levels; and when the system is finally turned on, it will respond as though all requests had just been accepted, handling the highest priority one first.

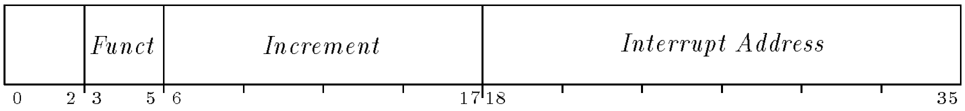
Starting an Interrupt

A request made to an active level is accepted immediately unless some level is already waiting for an interrupt to start or an interrupt is starting for some level. Once a request is accepted with the system on, the level must wait for the interrupt to start. The processor however will delay any action on the request if it is already holding an interrupt for the same level or for a level with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). When a waiting level has priority higher than the current program, the processor sends an interrupt–granted signal for the waiting level that has highest priority. This action makes use of the I/O bus. Should the bus be busy, the grant is sent as soon as the bus becomes available, taking precedence over any I/O instruction that may also be waiting (note that in this situation the program actually stops). The grant signal goes out on the bus and is transmitted serially from one device to the next. Upon receiving the grant, a device that is not requesting an interrupt on the specified level sends the signal on to the next device. A device that is requesting an interrupt on the specified level terminates the signal path and sends an interrupt function word back to the processor. Note that there are therefore two orders of priority associated with an interrupt: first the level, and then for all devices requesting interrupts simultaneously on the same level, proximity to the processor on the bus. For priority purposes, all devices on the left bus are closer than those on the right bus.

Upon receipt of the function word, the processor stops the current program at the first allowable point to start an interrupt for the waiting level for which the grant was made. Allowable stopping points are at the completion of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), between transfers in a BLT, between steps in the calculation of the first part of the quotient in double floating division, and while an I/O instruction is waiting for the bus. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

The action taken by the processor in starting an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the executive process table are associated with each level: locations $40+2N$ and $41+2N$, where N is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 57. The processor starts a "standard" interrupt for level N by executing the instruction in the first interrupt location for the level, i.e., location $40 + 2N$. The fixed locations however need not be used. The interrupt function word sent by the device may specify a standard interrupt using the fixed locations, or an equivalent interrupt using a pair of locations specified by the function word, or some other interrupt function entirely. The format of the function word and the operations the processor performs in response to the function selected by bits 3–5 of the word are as follows.

KI10 Interrupt Function Word



Bits 3–5	*Interrupt Function*
0	Processor waiting. If no response, perform a standard interrupt (see function 1). A device designed originally for use with the KA10 will work when connected to the KI10 bus, where it always requests a standard interrupt by providing no response to the grant. Note that for simultaneous requests on a given level, all KI10 devices that return a function word have priority over all KA10 devices and over any KI10 devices that do not return a function word. The last group includes the reader, punch and console terminal, which are contained in the processor, as well as the processor itself acting as a device (see processor conditions, §4.3.3).
1	Standard interrupt — execute the instruction in location $40 + 2N$ of the executive process table.
2	Dispatch — execute the instruction in the location specified by bits 18–35.
3	Increment — add the contents of bits 617 to the contents of the location specified by bits 18–35. The increment is a fixed point number in twos complement notation, bit 6 being the sign, and bit 17 corresponding to bit 35 of the memory word.
4	DATAO — do a DATAO for this device using the contents of bit 18–35 as the effective address.
5	DATAI — do a DATAI for this device using the contents of bit 18–35 as the effective address.
6	Reserved (produces a standard interrupt).
7	Reserved (produces a standard interrupt).

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode. No interrupt operation can set Overflow or either of the trap flags; hence an overflow trap can never occur as a direct result of an interrupt. A page failure that occurs in an interrupt operation is never trapped; instead it sets the In–Out Page Failure flag, which requests an interrupt on the level assigned to the processor (§4.3.3). These considerations of course do not apply to a service routine called by an interrupt instruction.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being performed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in the first or second interrupt location for a level, in direct response by the hardware (rather than by the program) to a request on that level. These locations may be the fixed ones for a standard interrupt or those given by the function word for a dispatch interrupt. §2.18 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. The interrupt instructions executed in a standard or dispatch interrupt fall into three categories.

- AOSx, SKIPx, SOSx, CONSx, BLKx. If the skip condition specified by the instruction is satisfied, the processor dismisses the interrupt and returns immediately to the interrupted program (i.e., it returns control to the unchanged PC). If the skip condition is not satisfied, the processor executes the instruction contained in the second interrupt location.

Satisfaction of the condition does not change PC, as this would skip the next instruction in the interrupted program. In effect the instruction skips back to the interrupted program by skipping the second interrupt location.

Note that the interpretation of a BLKI or BLKO as a skip instruction is consistent with the description given in §2.18 the condition being that the count is not zero.

CAUTION

> In the second interrupt location, a skip instruction whose condition is not satisfied hangs up the processor, which will keep repeating the instruction until the condition is satisfied.

- JSR, JSP, PUSHJ, MUUO. The processor holds an interrupt on the level, takes the next instruction from the location specified by the jump (as indicated by the newly changed PC), and enters either kernel mode or the mode specified by the new PC word of the MUUO. Hence the instruction is usually a jump to a service routine handled by the Monitor.
- *All Other Instructions.* In general the processor simply executes the instruction, dismisses the interrupt, and then returns to the interrupted program. If the instruction is a jump (other than those mentioned above), the processor jumps to the newly specified location; but it dismisses the interrupt and returns to the mode it was already in when the interrupt occurred. Hence it effectively returns to the interrupted program but in a different place, and the original contents of PC are lost.

Since the interrupt operations are performed in kernel mode regardless of the actual mode of the processor, an XCT is performed as a PXCT (§4.3.4). The ultimate effect of the XCT depends of course on the instruction executed — and its effect is as described here for the various categories.

CAUTION

> Neither an LUUO, a BLT, a DMOVEM, nor a DMOVNM will function in a reasonable manner as an interrupt instruction. Therefore do not use them.

Interrupt Programming

The program can control the priority interrupt system by means of condition I/O instructions. The device code is 004, mnemonic PI.

CONO PI, Conditions Out, Priority Interrupt

70060	I	X	Y
0 12	13	14 17	18 35

Perform the functions specified by the effective conditions E as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

Clr Pwr Fail Flag	Clr Par Err Flag	Parity Err Interrupt		Drop Prgm Req On Lvls	Clear PI Sys–tem	Selected Levels			PI System		Select Levels for Bits 22,24,25,26						
		Dis–able	En–able			Make Prgm Req On	Turn On	Turn Off	Turn Off	Turn On	1	2	3	4	5	6	7
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Bits 18–21 are actually for processor conditions (§4.3.3).

20 Prevent the setting of the Parity Error flag from requesting an interrupt on the level assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the level assigned to the processor.

22 On levels selected by is in bits 29–35, turn off any interrupt requests made previously by the program (via bit 24).

23 Deactivate the priority interrupt system, turn off all levels, eliminate all interrupt requests that have already been accepted but are still waiting, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by is in bits 29–35, and force the processor to accept them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a CONO that selects the same level and has a 1 in bit 22.

Remember that the processor allows the program to continue while it grants an interrupt. Thus when this bit forces acceptance of a request, another program instruction or two may be performed before the interrupt, even on the highest priority level. Moreover if the request is allowed to remain, additional instructions may be performed between successive interrupts. For other than the highest priority level, the greater the number of higher priority levels active, the greater the amount of program time available both initially and between successive interrupts. If the program forces an interrupt on the lowest priority level when all are active, there can be as much as 40μs of program time between the CONO PI, and its interrupt.

25 Turn on the levels selected by is in bits 29–35 so interrupt requests can be accepted on them.

26 Turn off the levels selected by is in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.

28 Activate the priority interrupt system so the processor can accept requests and can start, hold and dismiss interrupts.

CONI PI, Conditions In, Priority Interrupt

70064	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the priority interrupt (and nine console operating switches) into location *E* as shown.

Inst Fetch	Data Fetch	Write	Addr Stop	Addr Brk	Exec Pag	User Pag	Par Stop	NXM Stop			Program Requests on Levels						
											1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

			Interrupt in Progress on Levels							PI Sys–tem On	Levels On (Active)						
			1	2	3	4	5	6	7		1	2	3	4	5	6	7
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Levels that are active are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 0–8 reflect the settings of various console operating switches; for information on these switches refer to Appendix G.2.

Dismissing an Interrupt. Unless the interrupt operation dismisses the interrupt automatically, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that level or any level of lower priority (requests, however, can be accepted on lower priority levels.).

A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be on when the JEN is given). This instruction restores the level on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority levels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, PUSHJ, or MUUO. In the unlikely event that flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

> An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its level and all levels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Timing. The time a device must wait for an interrupt to start depends on the number of levels in use, and how long the service routines are for devices on higher priority levels. If only one device is using interrupts, it need never wait longer than $10\mu s$.

Special Considerations. On a return to an interrupted program, the processor always starts the interrupted instruction over from the beginning. This causes special problems in a BLT and in byte manipulation.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two–part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a PC word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an ILDB or IDPB would skip a byte. And if the routine restores the flag, the interrupted ILDB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In–Out flag is set, a user program generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- No requests can be accepted, not even on higher priority levels, while an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- Most in–out devices are designed to drop an interrupt request when the program responds, usually with a DATAI or DATAO. If an interrupt is handled neither by a BLKI or BLKO interrupt instruction nor by a service routine, the programmer must make sure the device is configured to drop the request on receipt of whatever response the program does give.
- The interrupt instruction that calls the routine must save PC if there is to be a return to the interrupted program. Generally a JSR is used as it saves both PC and the flags, and it uses no accumulator
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. For example, computations that can be performed outside the routine should not be included within it.

- If the routine uses a UUO it must first save the contents of the pair of locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type. For an MUUO, the routine must save locations 424 and 425 of the user process table. For an LUUO the routine must save location 40 in the executive process table and the location used by the UUO handler instruction to store the PC word.
- The routine must dismiss the interrupt (with a JEN) when returning to the interrupted program. The flags and UUO locations should be restored.

4.3.3 KI10 Processor Conditions

Page failures and overflow are handled by trapping, but there are a number of internal conditions that can signal the program by requesting an interrupt on a level assigned to the processor. The program can actually assign two levels — one for error conditions and one specifically for the clock. Control over the Power Failure and Parity Error flags is exercised by a CONO that addresses the priority interrupt system (§4.3.2). Control over other conditions and inspection of all are handled by condition I/O instructions that address the processor: the CONI also reads some console switches and maintenance functions. The processor also has a data–out instruction through which the program can perform margin checking of the system in both speed and voltage.

The error conditions are generally regarded as important enough to be assigned to the highest priority level. However for conditions that may be associated with user instructions (a parity error or unanswered memory reference), the common practice is for the error interrupt to switch over to the lowest priority level by means of a program–set request. Then the time to handle the situation, which may well be considerable, cannot interfere with high priority events.

One of the features controlled by the CONO for the processor is the automatic restart after power failure. This restart applies only when the levels on the power mains go below specification while the processor is running, and the power switch is on when power is restored — the machine never begins operation by itself when the operator turns the power switch on or off. Inadequate power, over temperature, etc. are indicated by the Power Failure flag. In order for the processor to restart itself, the program must respond in a particular way to the setting of Power Failure. If the program fails to respond properly, there is no restart.

The processor device code is 000, mnemonic APR.

CONO APR, Conditions Out, Arithmetic Processor

70020	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt levels specified by bits 30–35 of the effective conditions E and perform the functions specified by bits 18–29 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

Reset Timer	Clear In–Out Devices	Dis–able	En–able	Dis–able	En–able	Dis–able	En–able	Clear Clock		Clear In–Out Page Fail	Clear NXM	Priority Interrupt Assignment for Error			Priority Interrupt Assignment for Clock		
		Timer		Auto–Restart		Clock Interrupt											
18	19	20	21	22	23	24	25	26	27	28	29	30		32	33		35

A 1 in bit 19 produces the I/O reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system nor the processor conditions).

CONI APR, Conditions In, Arithmetic Processor

70024		*I*	*X*		*Y*	
0	12	13	14	17	18	35

Read the status of the processor (as well as various console switches and maintenance functions) into location *E* as shown (asterisks indicate bits that can cause interrupts).

	Mem Over–lap Dis–able	FM Man–ual	MI Prog Dis–able	Con–sole Data Lock	Con–sole Lock	50 Hz	Mar–gin En–able	Maint Mode	Power Alarm	Vol–tage Mon–itor Low		Sense Switches					
												1	2	3	4	5	5
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

	*			*				*		*	*						
Time Out	Par Error	Par Error Int En–abled	Timer En–abled	Power Fail	Auto–Re–start Dis–abled		Clock Int En–abled	Clock		In–Out Page Fail	NXM	Priority Interrupt Assignment for Error			Priority Interrupt Assignment for Clock		
18	19	20	21	22	23	24	25	26	27	28	29	30		32	33		35

Interrupts are requested on the error level (assigned by bits 30–32 of the CONO) by the setting of Power Failure, In–Out Page Failure, Nonexistent Memory, and if enabled, Parity Error. The setting of Clock Flag, if enabled, requests an interrupt on the clock level (assigned by bits 33–35 of the CONO).

Bits 12–17 reflect the states of the console sense switches, which are specifically for operator communication with the program. Bits 1–5 reflect the settings of various console operating switches; for information on these switches refer to Appendix G.2. Bits 7–10 are maintenance functions[71] for which the reader should refer to Chapter 10 of the *KI10 Maintenance Manual*.

6 The system is operating on 50 Hz line power. This is important to the program, not only because some I/O devices run slower on 50 Hz, but because the program must compensate for the time difference when using the line frequency clock (bit 26).

---

[71] The processor does not actually have a maintenance mode — the bit is simply the OR function of a number of console switches, any one of which being on implies that the processor is being operated for maintenance purposes.

18 Bit 21 is 1 and the program has not reset the timer (CONO APR, bit 18) during the last 1.2 seconds (the period of the timer may vary from 1.2 to 1.5 seconds). The setting of this flag clears the processor and the peripheral equipment, and restarts the processor in kernel mode at location 70.[72]

19 A word with even parity has been read from core memory. If bit 20 is 1, the setting of Parity Error requests an interrupt on the error level (see cautions below).

22 AC power has failed. The program should save PC, the flags, mode information and fast memory in core, and halt the processor. Note that PC may point to an interrupt service routine rather than the main program.

The setting of this flag requests an interrupt on the error level. After 4 ms the processor is cleared. But at that time, if the power switch is on and the program has cleared Power Failure (CONO PI,400000) and enabled the auto restart (CONO APR,010000), then when adequate power levels are restored, the processor will resume normal operation by executing the instruction in location 70 in kernel mode. The restart instruction should set up PC, which would otherwise be clear.

26 This flag is set at the ac power line frequency and can thus be used for low resolution timing (the clock has high long term accuracy). If bit 25 is 1, the setting of the Clock flag requests an interrupt on the clock level.

28 A page failure has occurred in an interrupt instruction. The setting of this flag requests an interrupt on the error level. An interrupt page failure caused by the console address break switch also sets this flag instead of producing an address failure (§4.3.4).

Note: A page failure in an interrupt instruction is regarded as a fatal error, and it causes an interrupt instead of a page failure trap. The kernel program is expected to set up the interrupt instructions so that a failure simply cannot occur.

29 The processor attempted to access a memory that did not respond within 100μs. The setting of this flag requests an interrupt on the error level (see cautions below).

Note: PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

Programming Cautions. When handling parity error or nonexistent memory interrupts, the programmer should beware of the following. Should an error flag be set during an interrupt grant, the processor would handle a lower priority interrupt before getting to the processor interrupt. This means PC may be pointing to a lower level interrupt service routine rather than the program level at which the error occurred. (Remember that during the grant procedure, the interrupt system is otherwise static and the program continues. Moreover the processor is effectively at the far end of the bus.)

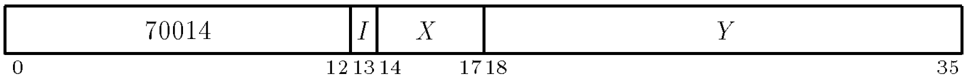
- Even without inadvertent interference from another level, it is quite likely the processor will perform one or perhaps two more instructions between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may well be pointing to the first or second instruction following the one in which the error occurred.

---

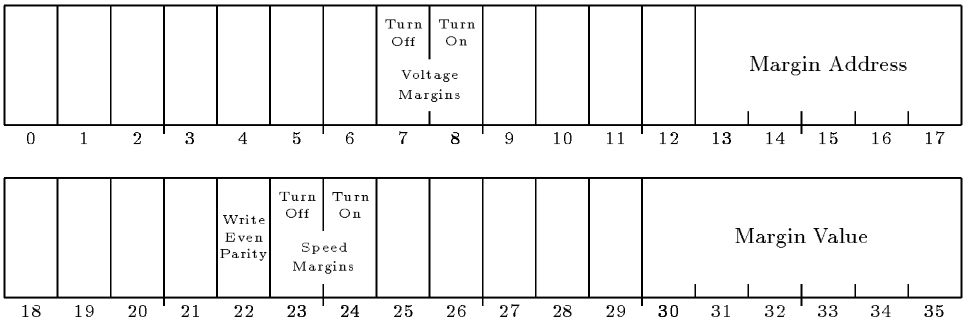
[72] The timer provides a restart similar to that following power failure. Running the machine under margins may result in significant logical errors. If the timer is enabled, failure of the program to reset it about every second allows it to time out. The restart instruction should set up PC, which would otherwise be clear.

- A processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second processor interrupt before the error–handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another — consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what went wrong.
- The error may have originated from a console key function, and thus be hidden from any investigation by the program.

DATAO APR, Maintenance Data Out, Arithmetic Processor



Supply diagnostic information and perform diagnostic functions according to the contents of location E as shown.



The margin value specified by bits 30–35 of the output word is translated to a voltage in the range 0–10 volts by a D–A converter, whose output is available at pin 2S02V2. Running margins requires a slowdown capacitor in the converter. But turning off the margin enable switch cuts out the capacitor, making the converter output suitable for external use, such as for operating audio equipment to play Bach or rock or Bacharach.

Notes. This instruction is primarily for maintenance, for which further information is given in Chapter 10 of the *KI10 Maintenance Manual*.

4.3.4 KI10 Program and Memory Management

General information about the machine modes and paging procedures is given in Chapter 1, in particular in §1.4. Here we are concerned principally with the special instructions the Monitor uses to operate the system, the special effects that ordinary instructions have in executive mode, and certain hardware procedures, in particular paging and page failures, that are necessary for an understanding of executive programming.

Paging

All of memory both virtual and physical is divided into pages of 512 words[73] each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18–bit addresses, where the left nine bits specify the page number and the right nine the location within the page. Physical memory can contain 8192 pages and requires 22–bit addresses, where the left thirteen bits specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18–bit addresses into 22–bit addresses. In this mapping the right nine bits of the virtual address are not altered; in other words a given location in a virtual page is the same location in the corresponding physical page. The transformation maps a virtual page into a physical page by substituting a 13–bit physical page number for the 9–bit virtual page number. The mapping procedure is carried out automatically by the hardware, but the page map that supplies the necessary substitutions is set up by the kernel mode program. Each word in the map provides information for mapping two consecutive pages with the substitution for the even numbered page in the left half, the odd numbered page in the right half.

The pager contains two 13–bit registers that the Monitor loads to specify the physical page numbers of the user and executive process tables. To retrieve a map word from a process table, the hardware uses the appropriate base page number as the left thirteen bits of the physical address and some function of the virtual page number as the right nine bits. For example the entire user space of 512 virtual pages at two mappings per word requires a page map of just half a page, and this is the first half page in the user process table. Thus locations 0–377 in the table hold the mappings for pages 0 and 1 to 776 and 777. To find the desired substitution from the 9–bit virtual page number, the hardware uses the left eight bits to address the location and the right bit to select the half word (0 for left, 1 for right). If the Monitor specifies a program as being a small user, that program is limited to two 16K blocks with addresses 0–37777 and 400000–37777. This is pages 0–37 and 400–37, and the mappings are in locations 0–17 and 200–217 in the page map.

The executive virtual address space is also 256K but the first 112K are not paged — in other words any address under 340000 given in kernel mode addresses one of the first 112K locations in physical memory directly. The other 144K is paged for supervisor or kernel mode anywhere into physical memory. For this there are two maps. The map for the second half of the virtual address space uses the same locations in the executive process table as are used in the user process table for the user map (locations 200–377 for pages 400–777). The map for the remaining 16K in the first half of the executive virtual address space is in the user process table, the mappings for pages 340–377 being in locations 400–17. Thus the Monitor can assign a different set of thirty–two physical pages (the per–process area) for its own use relative to each user. Then when switching from one user to another, the Monitor need change only the user process table. This single substitution can make whatever change is necessary in the executive address space for a particular user.

Figure 4.11 and Figure 4.12 show the organization of the virtual address spaces, the process tables and the mappings for both user and executive. The first illustration gives the correspondence between the various parts of each address space and the corresponding parts of the page map for it. The second illustration lists the detailed configuration of the process tables. Any table locations not used by the hardware can be used by the Monitor for software functions. Note that the numbers in the half locations in the page map are the virtual pages for which the half words give the physical substitutions. Hence location 217 in the user page map contains the physical page numbers for

---

[73] Actually page 0 has only 496 locations using a addresses 20–777, as addresses 0–17 reference fast memory, which is unrestricted and available to all programs. (In general a user cannot reference the first sixteen core locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to core and are not made by an instruction executed by a PXCT (see below).

virtual pages 436 and 437.

Although the virtual space is always 256K by virtue of the addressing capability of the instruction format, the Monitor usually limits the actual space for a given program by defining only certain pages as accessible.[74] The Monitor also specifies whether each page is public or not and writable or not. Each word in the page map has this format to supply the necessary information for two virtual pages.

Data for Even Virtual Page						Data for Odd Virtual Page					
A	*P*	*W*	*S*	*X*	*Physical Page Address Bits 14–26*	*A*	*P*	*W*	*S*	*X*	*Physical Page Address Bits 14–26*
0	1	2	3	4	5–17	18	19	20	21	22	23–35

Bits 5–17 and 23–35 contain the physical page numbers for the even and odd numbered virtual pages corresponding to the map location that holds the word. The properties represented by 1s in the remaining bits are as follows.

Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Public
W	Writable (not write–protected)
S	Software (not interpreted by the hardware)
X	Reserved for future use by DEC (do not use)

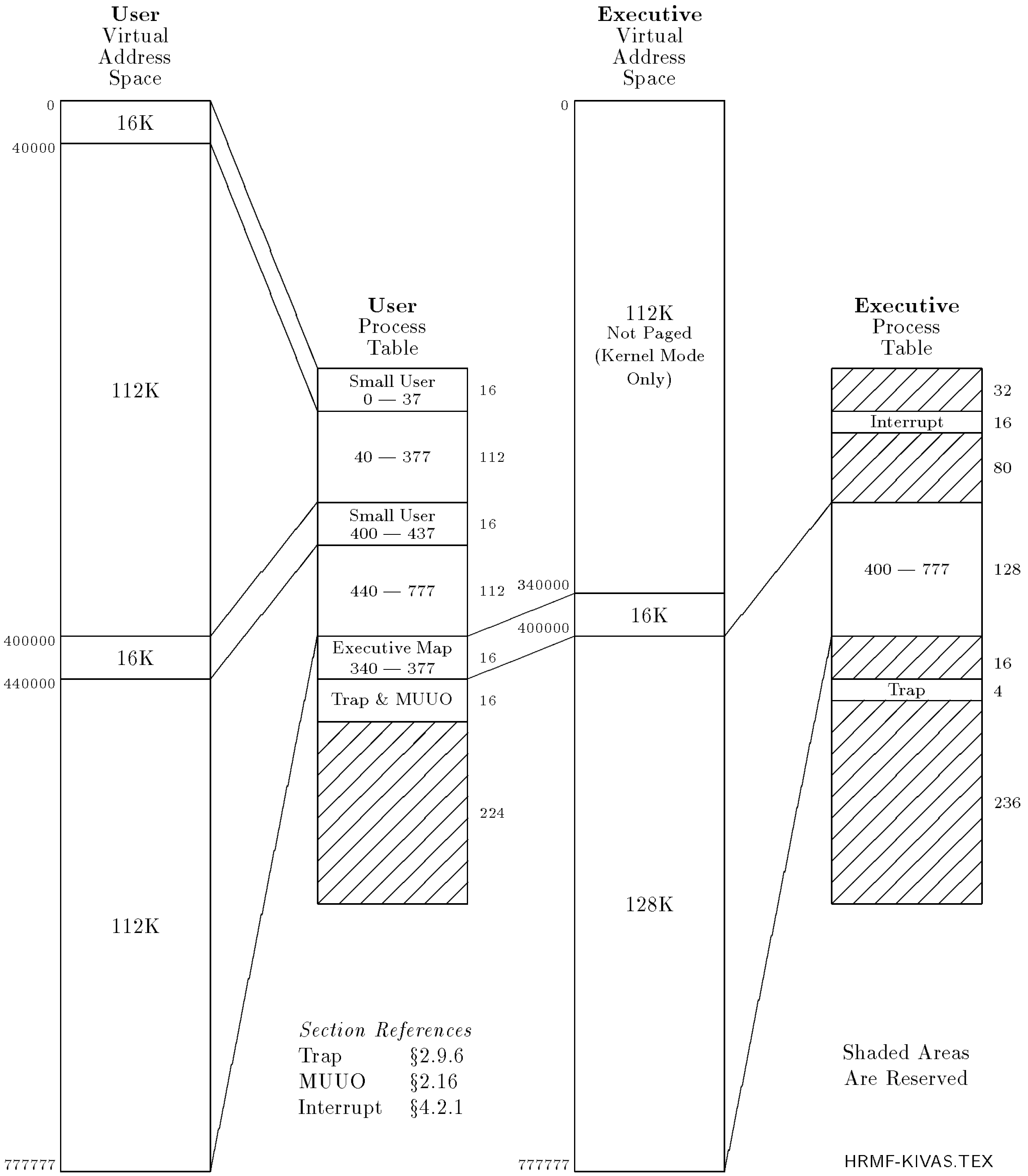
Associative Memory. If the complete mapping procedure described above were actually carried out in every instance, the processor would require two memory references for every reference by the program. To avoid this the pager contains a 32–word associative memory, in which it keeps the more recently used mappings for both the executive and the current user Each word is divided into two parts with one part containing a virtual page number specified by the program and the other containing the corresponding physical page number as determined from the page map. Hence the associative memory is a page table made up of a list of virtual pages and a list of physical pages, each with thirty–two corresponding locations. In the virtual list, each entry contains a 9–bit virtual page number, a single bit that indicates whether the specified page is in the user or executive address space, and a bit that indicates whether the entry is valid or not (it is not suitable to clear a location as 0 is a perfectly valid page number). Each corresponding entry in the physical list contains a 13–bit physical page number and the P, W and S bits from the map half word for that page. The A bit is not needed in the table as the mapping is not entered into the table at all if the page is not accessible. The program can inspect the contents of the page table by using the MAP instruction and I/O instructions that address the paging hardware (see below).

At each reference the hardware compares the page number supplied by the program with those in the virtual part of the page table. If there is a match for the appropriate address space, the corresponding entry in the physical list is used as the left thirteen bits in the physical address (provided of course

---

[74] There is no requirement that the accessible space be continuous — it can be scattered pages. The convention however is for the accessible space to be in two continuous virtual areas, low and high, beginning respectively at locations 0 and 400000. The low part is generally unique to a given user and can be used in any way he wishes. The (perhaps null) high part is a reentrant area, which is shared by several users and is therefore write–protected. The small user configuration is consistent with this arrangement.

Figure 4.11: Virtual Address Space and Page Map Layout (KI10)



Section References
Trap §2.9.6
MUUO §2.16
Interrupt §4.2.1

Shaded Areas
Are Reserved

HRMF-KIVAS.TEX

Figure 4.12: Process Table Configuration (KI10)

User Process Table

0	User Page 0	User Page 1
17	User Page 36	User Page 37
20	User Page 40	User Page 41
	Available to Software if Small User	
177	User Page 376	User Page 377
200	User Page 400	User Page 401
217	User Page 436	User Page 437
220	User Page 440	User Page 441
	Available to Software if Small User	
377	User Page 776	User Page 777
400	Executive Page 340	Executive Page 341
417	Executive Page 376	Executive Page 377
420	User Page Failure Trap Instruction	
421	User Arithmetic Overflow Trap Instruction	
422	User Pushdown Overflow Trap Instruction	
423	User Trap 3 Trap Instruction	
424	MUUO Stored Here	
425	MUUO Old PC Word	
426	Executive Page Failure Word	
427	User Page Failure Word	
430	Kernel No Trap MUUO New PC Word	
431	Kernel Trap MUUO New PC Word	
432	Supervisor No Trap MUUO New PC Word	
433	Supervisor Trap MUUO New PC Word	
434	Concealed No Trap MUUO New PC Word	
435	Concealed Trap MUUO New PC Word	
436	Public No Trap MUUO New PC Word	
437	Public Trap MUUO New PC Word	
440	Reserved	
777		

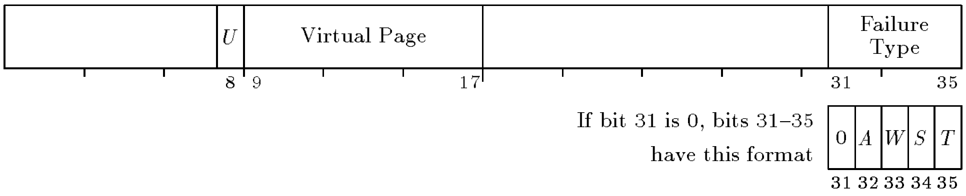
Executive Process Table

0	Available to Software	
37		
40	Executive LUUO Stored Here	
41	LUUO Handler Instruction	
42	Standard Priority Interrupt Instructions	
57		
60	Reserved	
177		
200	Executive Page 400	Executive Page 401
377	Executive Page 776	Executive Page 777
400	Reserved	
417		
420	Executive Page Failure Trap Instruction	
421	Executive Arithmetic Overflow Trap Instruction	
422	Executive Pushdown Overflow Trap Instruction	
423	Executive Trap 3 Trap Instruction	
424	Reserved	
777		

that the reference is allowable according to the P and W bits). If there is no match, the hardware makes a memory reference (referred to as a "page refill cycle") to get the necessary information from the page map and enters it into the page table at the location specified by a reload counter. This counter is incremented whenever it is used to reload the table, and also whenever the location to which it points is used for a mapping. Hence the counter tends to stay away from locations containing the page numbers most frequently referenced.

Page Failure

A page failure that occurs during an interrupt instruction terminates the instruction and sets the In–Out Page Failure flag, requesting an interrupt on the error level assigned to the processor. In all other circumstances, if the paging hardware cannot make the desired memory reference, it terminates the instruction immediately without disturbing memory, the accumulators or PC, places a page fail word in the user process table, and causes a page failure trap. If the attempted reference is in user virtual address space, the page fail word is placed in location 427 of the user process table, and the processor executes the trap instruction in location 420 of the same table.[75] If the attempted reference is in executive virtual address space, the page fail word is placed in location 426 of the user process table, and the processor executes the trap instruction in location 420 of the executive process table. The trap instruction is executed in the same address space in which the failure occurred. The page fail word supplies this information.



Whether the violation occurred in user or executive virtual address space is indicated by a 1 or a 0 in bit 8. If bit 31 is 1, the number in bits 31–35 (≥ 20) indicates the type of "hard" failure as follows.

23 Address failure —- this is a simulated page failure caused by the satisfaction of an address condition selected from the console. It indicates that while the console address break switch was on and the Address Failure Inhibit flag was clear (bit 8 of the PC word), the processor initiated a page check for access to the memory location that was specified by the paging and address switches and for which a comparison was enabled (whether or not a comparison can be made is a function of the setting of the paging switches (Appendix G.2) and the state of the User Address Compare Enable flag (see below)), and the intended memory reference was for the purpose selected by the address condition switches as follows:

The instruction fetch switch was on and the requested access was for retrieval of an

---

[75] When a page failure trap instruction is performed, PC points to the instruction that failed (or to an XCT that executed it), unless the failure occurred in an overflow trap instruction in which case PC points to the instruction that overflowed. After taking care of the failure, the processor can always return to the interrupted instruction. Either the instruction did not change anything, or the failure was in the second part of a two–part instruction, where First Part Done being set prevents the processor from repeating any unwanted operations in the first part.

ordinary instruction, including an instruction executed by an XCT or an LUUO (address 41).

The data fetch switch was on and the requested access was for retrieval of an address word in an effective address calculation or read–only retrieval of an operand (other than in an XCT). This switch can also cause a failure inadvertently[76] on the retrieval of a trap instruction or a PC word in an MUUO.

The write switch was on and the requested access was for writing,[77] either write–only or read–modify–write, including writing by an LUUO (address 40). This switch also causes a failure on the first write in an MUUO if the address switches contain the effective address of the MUUO (even though that address is not used for the access), and can cause a failure inadvertently[76] on the second write.

The Address Failure Inhibit flag, which can be set only by a JRSTF or MUUO, prevents an address failure during the next instruction — the completion of the next instruction automatically clears it. If an interrupt or trap intervenes, the flag has no effect and it is saved and cleared if the PC word is saved. If it is not saved, it affects the instruction following the interrupt or trap. Otherwise it affects the instruction following a return in which it is restored with the PC word. Using this flag, the Monitor can return to a user instruction that caused an address failure and "get by it."

22 Page refill failure — this is a hardware malfunction. The paging hardware did not find the virtual page listed in the page table, so it loaded paging information from the page map into the table but still could not find it.

20 Small user violation — a small user has attempted to reference a location outside of the limited small user address space.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page or transfer control into a concealed page at an invalid entry point (one not containing a JRST 1,).

If the violation is not one of these, then bits 31–35 have the format shown above where A, W, and S are simply the corresponding bits taken from the map half word for the page, and T indicates the type of reference in which the failure occurred — 0 for a read reference, 1 for a write or read–modify–write reference. The type of reference implies nothing about the cause of failure — it indicates only the reason the failed reference was being made.

The page fail trap instruction is set by the Monitor to transfer control to kernel mode. After rectifying the situation, the Monitor returns to the interrupted instruction, which starts over again from the beginning.[78] Even a two–part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores the First Part Done flag.

---

[76] Virtual addresses are supplied to the paging hardware via the address bus. An inadvertent failure occurs when the bus is not used for an access but it accidentally contains the number set into the address switches. The data fetch switch also catches the attempt to retrieve a dispatch interrupt instruction or inadvertently a standard interrupt instruction, but the page failure sets the In-out Page Failure flag instead of resulting in a trap for an address failure.

[77] The write switch causes a failure on an instruction fetch if a read-modify-write precedes it immediately (e.g. if there is no intervening interrupt, the program is not being single stepped, etc).The write switch causes a failure on an instruction fetch if a read-modify-write precedes it immediately (e.g. if there is no intervening interrupt, the program is not being single stepped, etc).

[78] In a soft page failure, the mapping entry for the page is removed from the page table on the assumption that the Monitor will change it. When the instruction is restarted, the hardware must go to the page map to get a new entry for the page table.

Note that a failure does not necessarily imply that anything is "wrong." The virtual address space of even a small user is 32K words, which may well be more than is needed in a given run. Hence the Monitor may have only ten or twenty pages of the user program in core at any given time, and these would be the virtual pages indicated as accessible. When the user attempts to gain access to a page that is not there (a virtual page indicated in the page map as inaccessible), the Monitor would respond to the page failure by bringing in the needed page from the drum or disk, either adding to the user space or swapping out a page the user no longer needs.

The same situation exists for writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered. Then in response to a write failure, the Monitor makes the page writable and indicates to itself (perhaps by means of the software bit in the page map) that that page has in fact been altered. When the user is done, the Monitor need write only the altered pages back onto the drum.

Monitor Programming

The kernel mode program is responsible for the overall control of the system. It is the only program that has access to any of physical core unpaged and that has no instruction restrictions. The kernel program handles all in–out for the system and must set up the page maps, trap locations, interrupt locations and the like. The supervisor program labors under the same instruction restrictions as the user but has no way of bypassing them — they always apply. Supervisor mode is limited to the 144K paged part of the executive address space, although within that space it can read but not alter concealed pages. The supervisor can give a JRSTF that clears Public provided it is also setting User; in other words the supervisor can return control to a concealed program but cannot enter kernel mode by manipulating the flags. The PC words supplied by MUUOs can manipulate the flags in any way, switching arbitrarily from one mode to another, but these are in the process table and assumed to be under control solely of kernel mode.

For accumulator, index register and fast memory references, the Monitor automatically uses fast memory block 0. For each user, the kernel mode program must assign a block. The usual procedure is to assign blocks 2 and 3 to individual user programs on a semipermanent basis for special applications and to assign block 1 to all other users. In this way the Monitor need not store blocks 2 and 3 when the special users are not running, and it need not store block 1 when it takes over control from an ordinary user temporarily. If the Monitor shared block 0 with any users, it would have to store the user accumulators even when taking control only temporarily. When switching from one user to another, the Monitor usually stores the first user's accumulators in his shadow area — this is locations 0–17 in user virtual page 0, an area not generally accessible to the user at all — and loads the new user S accumulators from his shadow area, where they were stored after the last time the new user ran.

Even while User is set, the interrupt instructions are not part of the user program and are thus subject only to executive restrictions. (The page failure and overflow trap instructions are executed in the user address space if caused by the user.) As interrupt instructions, JSR, JSP, and PUSHJ automatically take the processor out of user mode to jump to an executive service routine. An MUUO can also be used.

The pager has one non–I/O instruction and two I/O instructions primarily for diagnostic purposes. Otherwise control over the system is exercised by data I/O instructions. The device code for the pager is 010, mnemonic PAG.

DATAO PAG, Data Out, Paging

70114	*I*	*X*	*Y*
0 12	13	14 17	18 35

Invalidate all data in the associative memory, and set up the paging hardware according to the contents of location E as shown. Invalidating all data in the associative memory means setting the Word Empty bit in each location to indicate that the rest of the word is meaningless and should not be used.

Load Left	User Fast Memory Block	Small User	User Address Compare Enable	User Base Address
0	1 2	3	4	5 6 7 8 9 10 11 12 13 14 15 16 17

Load Right		Page Enable	Executive Base Address
18	19 20 21	22	23 24 25 26 27 28 29 30 31 32 33 34 35

Bits 0 and 18 are change bits. If bit 0 is 0, ignore the rest of the left half word. But if bit 0 is 1, load bits 5–17 into the user base register to select the user process table, select the fast memory block specified by bits 1 and 2 for the user, limit the address space to that of a small user if bit 3 is 1, and enable address comparison if bit 4 is 1. The Address Compare Enable bit functions in conjunction with the console paging switches, as explained in Appendix G.2

Similarly if bit 18 is 0, ignore the rest of the right half word. Otherwise load bits 23–35 into the executive base register to select the executive process table, and enable executive paging if bit 22 is 1. For normal operation of the system, bit 22 must be 1. A 0 in this bit disables overflow traps, and disables executive paging so there is no supervisor mode and no executive virtual addressing — in other words an executive program automatically runs in kernel mode with all access in the first 256K of physical memory unpaged.[79]

NOTE

> Neither turning on power nor pressing the reset switch invalidates the data in the associative memory. Therefore, after power has been off, the starting kernel program must do a DATAO PAG, to clear the associative memory of random data before entering executive or user paged address space.

---

[79] An executive mode program that does not set bit 22 and avoids other special KI10 features will run on a KA10 as well. This is useful for hardware diagnostics and bootstrap loaders (see readin mode,§4.3.1).

DATAI PAG, Data In, Paging

70104	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the paging hardware into location E. The information read is the same as that supplied by a DATAO (bits 0 and 18 are 0).

CONO PAG, Conditions Out, Paging

70120	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the executive stack pointer from bits 18–22 and the page table reload counter from bits 31–35 of the effective conditions E as shown.

Executive AC Stack Pointer		Page Table Reload Counter
18 19 20 21 22	23 24 25 26 27 28 29 30	31 32 33 34 35

The executive stack pointer specifies a block of sixteen locations in the user process table by supplying the left five bits for a 9–bit address that references a location in the table; this function is used only for accessing stacked fast memory blocks in an instruction executed by a PXCT (see below). Loading the reload counter causes it to point to the specified location in the page table.

CONI PAG, Conditions In, Paging

70124	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the processor serial number, the page table reload counter, and the contents of the location in the virtual page table specified by the counter into location E as shown.

Processor Serial Number	
0–9	10–17

Complement of Virtual Page Number	Executive Address Space		Word Empty	Page Table Reload Counter
18–26	27	28–29	30	31–35

Note that bits 18–26 contain the complement of the virtual page number in the selected location. A 1 in bit 27 indicates the page is in the executive address space; a 1 in bit 30 means the information in bits 18–27 is invalid. It is possible for the reload counter to change between the CONI and the CONO, so the CONI might read a different location than was selected by the CONO.

MAP Map an Address

257	*A*	*I*	*X*	*Y*
0–8	9–12	13	14–17	18–35

Map the virtual effective address E and place the resulting map data in AC right in the same format as it is in the page map, i.e., bits P, W and S in bits 19–21 and the physical page number in bits 23–35. Clear AC left.

Page Failure	*P*	*W*	*S*	No Match	Physical Page Address Bits 14–26
18	19	20	21	22	23–35

This instruction cannot produce a page failure, but if a page failure would have resulted had an ordinary instruction in the same mode attempted to write in location E, place a 1 in AC bit 18. If no match can be made by the paging hardware, place a 1 in bit 22. This results in four possible situations as a function of the states of bits 18 and 22.

Bit 18	*Bit 22*	*Meaning*
0	0	AC right contains valid map data.
0	1	There is no page failure but also no match, so the instruction must have made an unmapped reference — perhaps to fast memory or to the unpaged area in kernel mode.
1	0	There is a page failure but the map data is correct as a match exists.
1	1	There is a page failure, and since there is no match, the failure must have resulted from the instruction referencing an inaccessible page or from some prior failure (such as a page refill malfunction). Hence AC right contains invalid information.

The last three instructions above can be used to inspect the contents of the associative memory. The CONO selects a location, the CONI reads the contents of the virtual–page part of that location, and a MAP that addresses the specified virtual page reads the contents of the physical–page part of that location.

Previous–Context Execute

Ordinarily an instruction in a user program is performed entirely in user address space, and an instruction in the executive program is performed entirely in executive address space. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous–context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero *A* bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous context. At any point in time, the previous–context is essentially the circumstances in which the previous process was running. A PXCT can be given only in executive mode, but the previous–context may be the user, as following a call to the Monitor by the user. The previous context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. But note that it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory operand references are made. To work as a PXCT, an XCT must be given in executive mode, and bits 11 and 12 in its *A* field (9–12) must not both be 0 (in user mode *A* is ignored). But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the instruction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction all effective address calculation and accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a fast memory location.) If the instruction makes no memory operand references, as in a jump, shift or immediate mode instruction, its execution differs in no way from the normal case. The only difference is in *memory operand references.*

The previous–context is specified by two flags. Just as the current mode is indicated by the User and

Public flags, the mode in which the calling program was running is indicated by Previous Context User and Previous Context Public.[80] At a call these flags are set up by an MUUO PC word. Note that the restrictions on references made in the previous–context are those of the previous–context — not those in which the PXCT is given. Suppose the executive executes an instruction that references the concealed user area. Such a reference would fail if Previous Context Public were set; in other words the concealed area can be accessed by a PXCT only when such access is requested by the concealed program.

Which references in the executed instruction are made in the previous context is determined by is in bit 11 and 12 of the PXCT instruction word as follows: a 1 in bit 12 selects read and read–modify–write memory operand references; a 1 in bit 11 selects memory operand write references; and is in both bits selects all memory operand references. The meaning of previous–context address space is obvious for core memory references, namely user or executive virtual address space. But this is not so for fast memory. When Previous Context User is set, the user space for fast memory references depends on which fast memory block is currently selected for the user. if block 0 is selected, fast memory operand references of the types specified are made to the user shadow area. If some other block is selected, the specified fast memory references are made to the selected block.

If Previous Context User is clear, fast memory references of the types specified are made to the user process table, in particular to that set of sixteen locations specified by the executive stack pointer. The pointer is given by a CONO PAG,.

Previous–Context Fast Memory References

Previous	*Fast Memory Block Selected*	
Context User	*Zero*	*Nonzero*
1	User shadow area	Selected user block
0	AC stack	AC stack

Individual Instruction Effects. The effects of execution by a PXCT on different types of instructions are as follows.

- Instructions without memory operand references are not affected. This includes shifts, jumps, immediate mode instructions, CONSO, CONO, and even an XCT. In fact not only is a PXCT not affected when executed by a PXCT, but the first destroys any effect the second would otherwise have on a third instruction (in other words, a pair of PXCTs is equivalent to a pair of ordinary XCTs).
- Instructions that refer to one memory location for reading only or reading and writing are controlled by the read bit (MOVE, MOVES, ADDM, AOS). The read bit controls writing when the write is done to the same location as the read, whether the memory references are done as a single cycle including both read and write or as separate read and write cycles.
- Instructions that refer to one memory location for writing only are controlled by the write bit (MOVEM, MAP, HRLZM).
- Instructions that refer to two different memory locations are controlled by the read bit in the read part of the instruction and by the write bit in the write part (BLT, PUSH)

---

[80] Previous Context User and Previous Context Public are in the same flag bits that are used for User In–Out and Overflow in user mode. The former has no meaning in executive mode, and the latter is not really necessary as the executive program is not ordinarily interested in performing extensive mathematical procedures.

- BLKI and BLKO are controlled by the write bit and the read bit respectively. The pointer reference is done in the same address space as the data transfer.
- In byte instructions all pointer calculations are done in executive address space. The read and write bits affect only the second part, i.e., the load or deposit.

Philosophy. The purpose of the PXCT is to facilitate the handling of user requirements by the Monitor, but the selection made by Previous Context User of the references affected by the read and write bits is to allow the Monitor to make recursive calls to itself, i.e., to perform MUUOs in the process of carrying out an MUUO given by the user. Specifically the state of Previous Context User differentiates between the Monitor response directly to the user MUUO and its response to its own MUUOs.

The new PC word of an MUUO from the user would set Previous Context User so that core memory references can be made across the user–executive boundary, and fast memory references can be made to the user AC block. The point in choosing between the shadow area and the selected block if not block 0 is to reference the information that was held in the user AC block before the Monitor took over. If the user shared block 0 with other users and the Monitor, the Monitor will have saved his ACs in the shadow area of his address space. The other AC blocks are not disturbed when the Monitor takes over temporarily, so the Monitor need not save them and they will still hold the user information.

If in the course of carrying out a user MUUO, the Monitor should itself give an MUUO, the new PC word would clear Previous Context User. Thus at this level all core memory references are in the executive address space and fast memory references are to an AC block in the user process table as specified by the executive stack pointer. MUUO calls by the Monitor to itself can be nested to a number of levels, but in all cases Previous Context User is left clear. The particular AC block used at any level is specified by the stack pointer, which makes a different set of sixteen words available at each level using the same adddresses. Hence the AC stack in the user process table is effectively a pushdown stack kept by the stack pointer; at each level the program must change the pointer to specify the appropriate block. Each user process table would contain the blocks needed for carrying out MUUOs for that user.

Example. Suppose that the Monitor has been called by an MUUO from the user (hence Previous Context User is set) and wishes to save the user's ACs in the shadow area. Assume that every user runs with AC block 1, 2 or 3, and that the Monitor always sets up executive virtual page 342 to point to the same physical page as user page 0. Using accumulator T in block 0, the Monitor saves the user ACs by giving these two instructions,

```
MOVEI T,342000          ;Initalize pointer: from 0 to 342000
XCT   1,[BLT T,342017]
```

and restores them with these two.

```
MOVSI T,342000          ;From 342000 to 0
XCT   2,[BLT T,17]
```

4.3.5 KA10 Priority Interrupt

Most in–out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often result in loss of information and certainly results in operating the device below its maximum speed. The priority interrupt is designed with these considerations in mind, i.e., the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The hardware also allows conditions internal to the processor to signal the program by requesting an interrupt.

Interrupt requests are handled through seven levels arranged in a priority chain, with assignment of devices to levels entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35). Levels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be connected to a single level, and some can be connected to two levels (e.g., a device may signal that data is ready on one level and use another level to signal that an error has occurred).

When a device requires service it sends an interrupt request signal over the in–out bus to its assigned level in the processor. The processor accepts the request depending upon certain conditions, such as that the level must be active (on). The request signal remains on the bus until turned off by the program (CONO, DATAO, or DATAI, depending on the device). Thus if a request is not accepted when made, it will be accepted when the conditions are satisfied. A single level will shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request. The program can usually trigger a request from a device but delay its acceptance by turning on the level later

The request signal is generally derived from a flag that is set by various conditions in the device. Often associated with these flags are enabling flags, where the setting of some device condition flag can request an interrupt on the assigned level only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device by a CONO. For example, a device may have half a dozen flags to indicate various internal conditions that may require service by an interrupt; by setting up the associated enabling flags, the program can determine which conditions shall actually request interrupts in any given circumstances.

Having accepted a request, the processor will do nothing further with it unless the priority interrupt system is on. But even with the system off, the processor will continue to accept requests on other levels; and when the system is finally turned on, it will respond as though all requests had just been accepted, handling the highest priority one first.

Starting an Interrupt. A request made to an active level is accepted at the next memory access unless the processor is starting an interrupt for any level or holding an interrupt for the same level. Once a request is accepted with the system on, the level must wait for the interrupt to start. The processor however cannot start an interrupt if it is already holding an interrupt for a level with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). When there is a higher priority level waiting, the processor stops the current rogram at the first allowable point to start an interrupt for the waiting level that has highest priority. Allowable stopping points are following the retrieval of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), and between transfers in a BLT. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

Two memory locations are associated with each level: unrelocated locations $40 + 2N$ and $41 + 2N$, where N is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 57. The processor starts an interrupt for level N by executing the instruction in location $40 + 2N$. Interrupt locations for a second processor on the same memory are $140 + 2N$ and $141 + 2N$. Even though the processor may be in user mode when an interrupt occurs, interrupt instructions are performed in executive mode.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being performed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in location $40 + 2N$ or $41 + 2N$, in direct response by the hardware (rather than by the program) to a request on level N. §2.18 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. There are two categories of interrupt instructions.

- *Non–I/O Instructions.* After executing a non-I/O interrupt instruction, the processor holds an interrupt on the level and returns control to PC. Hence the instruction is usually a jump to a service routine. If the processor is in user mode and the interrupt instruction is a JSR, JSP, PUSHJ, JSA, or JRST, the processor leaves user mode (the Monitor thus handles all interrupt routines).

  If the interrupt instruction is not a jump, the processor continues the interrupted program while holding an interrupt — in other words it now treats the interrupted program as an interrupt routine. For example, the instruction might just move a word to a particular location. Such procedures are usually reserved for maintenance routines or very sophisticated programs.

- *Block or Data I/O Instructions.* One or the other of two actions can result from executing one of these as an interrupt instruction.

  > If the instruction in $40 + 2N$ is a BLKI or BLKO and the block is not finished (i.e., the count does not cause the left half of the pointer to reach zero), the processor dismisses the interrupt and returns to the interrupted program. The same action results if the instruction is a DATAI or DATAO.
  >
  > If the instruction in $40 + 2N$ is a BLKI or BLKO and the count does reach zero, the processor executes the instruction in location $41 + 2N$. This cannot be an I/O instruction and the actions that result from its execution as an interrupt instruction are those given above for non–I/O instructions.

CAUTION

> The execution, as an interrupt instruction, of a CONO, CONI, CONSO, or CONSZ in location $40 + 2N$ or any I/O instruction in location $41 + 2N$ hangs up the processor

Interrupt Programming. The program can control the interrupt system by means of condition I/O instructions. The device code is 004, mnemonic PI.

CONO PI, Conditions Out, Priority Interrupt

70060	I	X	Y
0 … 12	13	14 … 17	18 … 35

Perform the functions specified by the effective conditions E as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

Clr Pwr Fail Flag	Clr Par Err Flag	Parity Err Interrupt			Clear PI System	Selected Levels			PI System		Select Levels for Bits 24,25,26						
		Dis–able	En–able			Make Prgm Req On	Turn On	Turn Off	Turn Off	Turn On	1	2	3	4	5	6	7
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Bits 18–21 are actually for processor conditions (§4.3.6)

20 Prevent the setting of the Parity Error flag from requesting an interrupt on the level assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the level assigned to the processor.

23 Deactivate the priority interrupt system, turn off all levels, eliminate all interrupt requests that have already been accepted but are still waiting, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by is in bits 29–35, and force the processor to accept them even on levels that are off. There is at most one interrupt on a given level, and a request is lost if it is made by this means to a level on which an interrupt is already being held.

25 Turn on the levels selected by is in bits 29–35 so interrupt requests can be accepted on them.

26 Turn off the levels selected by is in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.

28 Activate the priority interrupt system so the processor can accept requests and can start, hold and dismiss interrupts.

CONI PI, Conditions In, Priority Interrupt

70064	I	X	Y
0 … 12	13	14 … 17	18 … 35

Read the status of the priority interrupt (and several bits of processor conditions) into location E as shown.

Power Fail	Parity Error	Parity Error Int En-abled	Interrupt in Progress on Levels							PI Sys-tem On	Levels On (Active)						
			1	2	3	4	5	6	7		1	2	3	4	5	6	7
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held. A 1 in bit 28 means the interrupt system is on.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18–20 actually read processor status conditions (§4.3.6) as follows.

18 AC power has failed. The program should save PC, the flags and fast memory in core, and halt the processor. Note that PC may point to an interrupt service routine rather than the main program.

The setting of this flag requests an interrupt on the level assigned to the processor. If the flag remains set for 5 ms, the processor is cleared.

19 A word with even parity has been read from core memory. If bit 20 is set, the setting of the Parity Error flag requests an interrupt on the level assigned to the processor, at which time PC points to the instruction being performed or to the one following it.

Dismissing an Interrupt. Automatic dismissal of an interrupt occurs only in a DATAI or DATAO, or in a BLKI or BLKO with an incomplete block. Following any non–I/O interrupt instruction, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that level or any level of lower priority (requests, however, can be accepted on lower priority levels).

A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be on when the JEN is given). This instruction restores the level on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority levels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, or PUSHJ. In the unlikely event that flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

> An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its level and all levels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Timing. The time a device must wait for an interrupt to start depends on the number of levels in use, and how long the service routines are for devices on higher priority levels. If only one device is using interrupts, it need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is made. The maximum time can be considered to be about 15 μs for FDVL, but a ridiculously long shift could take over 35 μs.

Special Considerations and Programming Suggestions. If the interrupt routine uses a UUO it must first save the contents of the pair of locations that will be changed by it in case the interrupted program was in the process of handling a UUO. Hence the routine must save unrelocated location 40 and the location used by the UUO handler instruction to store the PC word. In all other respects, the special considerations and programming suggestions given at the end of the section on the KI10 interrupt hold for the KA10 (§4.3.2).

4.3.6 KA10 Processor Conditions

There are a number of internal conditions that can signal the program by requesting an interrupt on a level assigned to the processor. Most of these conditions are generally regarded as important enough to be assigned to the highest priority level. Except in the case of a power failure however, the common practice is for the processor interrupt to switch over to the lowest priority level by means of a program–set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

Flags for power failure and parity error are handled by the condition I/O instructions that address the priority interrupt system (§4.3.5). The remaining flags are handled by condition instructions that address the processor. Its device code is 000, mnemonic APR.

CONO APR, Conditions Out, Arithmetic Processor

70020	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt level specified by bits 33–35 of the effective conditions E and perform the functions specified by bits 18–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

Clear Pushdown Overflow	Clear All In–Out Devices		Clear Address Break Flag	Clear Memory Protect Flag	Clear NXM Flag	Disable	Enable	Clear Clock Flag	Disable	Enable	Clear Floating Overflow	Disable	Enable	Clear Overflow	Priority Interrupt Assignment		
						Clock Interrupt			Floating Overflow Interrupt			Overflow Interrupt					
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Enabling a particular flag to interrupt means that henceforth the setting of the flag will request an interrupt on the level assigned (by bits 33–35) to the processor. Disabling prevents the flag from triggering a request.

A 1 in bit 19 produces the I/O reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system, nor the processor flags cleared by this instruction or CONO PI,).

CONI APR, Conditions In, Arithmetic Processor

70024	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the status of the processor into the right half of location E as shown (all interrupt requests are made on the level assigned to the processor).

	*		*	*	*			*			*			*			
	Push–down Over–flow	User In–Out	Ad–dress Break	Mem–ory Pro–tect–ion	NXM		Clock Inter–rupt En–abled	Clock		Float–ing Over–flow Int Enb	Float–ing Over–flow	Trap Off–set	Over–flow Inter–rupt En–abled	Over–flow	Priority Interrupt Assignment		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Bits that can cause interrupts on the level assigned to the processor are those indicated by asterisks, and also Power Failure and Parity Error, bits 18 and 19 read by a CONI PI,.

With the possible exception of an illegal memory reference on an instruction fetch, if the highest priority active level is assigned to the processor, then the occurrence of any processor interrupt condition is guaranteed to produce a processor interrupt with no lower priority interrupt intervening between it and the program level at which the processor condition occurred. The actual relationship between PC and the instruction associated with a given condition is as stated in its description.

19 Pushdown Overflow — in a PUSH or PUSHJ the count in AC left reached zero; or in a POP or POPJ the count reached −1. The setting of this flag requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred. The location of the offending instruction is implied by PC for PUSH or POP, is indicated by the last item in the stack for PUSHJ, but is indeterminate for POPJ.

20 User In–Out — even if the processor is in user mode, there are no instruction restrictions (but memory restrictions still apply) (§4.3.7).

21 Address Break — while the console address break switch was on, the processor requested access to the memory location specified by the address switches and the memory reference was for the purpose selected by the address condition switches as follows:

> The instruction switch was on and access was for retrieval of an instruction (including an instruction executed by an XCT or contained in an interrupt location or a trap for an unimplemented operation) or an address word in an effective address calculation.
>
> The data fetch switch was on and access was for retrieval of an operand (other than in an XCT).
>
> The write switch was on and access was for writing a word in memory, other than in a read–modify–write.

The setting of this flag requests an interrupt, at which time PC points to the instruction that was being executed or to the one following it. However PC bears no relation to the break if the access was requested for a console key function.

22 Memory Protection — a user program attempted to access a memory location outside of its area or to write in a write–protected part of its area, and the user instruction was terminated at that time. The setting of this flag requests an interrupt, at which time PC points either to the instruction that caused the violation or to the one following it, unless the illegal reference was for fetching an instruction. In this exceptional case it is possible for a lower level interrupt to occur between the violation and its interrupt, even with the processor assigned to the highest priority active level.

This flag can also be set by an instruction executed from the console while the USER MODE light is on, in which case PC bears no relation to the violation.

23 Nonexistent Memory — the processor attempted to access a memory that did not respond within 100 μs. The setting flag requests an interrupt, at which time PC points either to the instruction containing the unanswered reference or to the one following it. However PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

26 Clock — this flag is set at the AC power line frequency and can thus be used for low resolution timing (the clock has high long term accuracy). If bit 25 is set, the setting of the Clock flag requests an interrupt.

29 Floating Overflow — this is one of the flags saved in a PC word, and the conditions that set it are given in §2.9.3. If bit 28 is set, the setting of Floating Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.

30 Trap Offset — the processor is using locations 140–161 for unimplemented operation traps and interrupt locations.

32 Overflow — this is one of the flags saved in a PC word, and the conditions that set it are given in §2.9.3. If bit 31 is set, the setting of Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.

CAUTION

For an address break, a memory protection violation, a parity error, or a nonexistent memory, a processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the processor will fetch the next user instruction before it accepts the program–set interrupt request. This makes it very likely that the same error will recur, producing a loop between the processor interrupt and the interrupted program.

4.3.7 KA10 Program and Memory Management

Every user is assigned a core area and the rest of core is protected from him — he cannot gain access to the protected area for either storage or retrieval of information. The assigned area is divided into two parts. The low part is unique to a given user and can be used for any purpose. The high part may be for a single user, or it may be shared by several users. The Monitor can write–protect the high part so that the user cannot alter its contents, i.e., he cannot write anything in it. The Monitor would do this when the high part is to be a pure procedure to be used reentrantly by several users. One high pure segment may be used with any number of low impure segments. The user can request that the Monitor write–protect the high part of a single program, e.g., in order to debug a reentrant program. All users write programs beginning at address 0 for the low part, and beginning usually at 400000 for the high part. The programmed addresses are retained in the object program but are relocated by the hardware to the physical area assigned to the user as each access is made while the program is running.

The size and position of the user area are defined by specifying protection and relocation addresses for the low and high blocks, as shown in Figure 4.13. The protection address determines the maximum address the user can give; any address larger than the maximum is illegal. The relocation address is the address, as seen by the Monitor and the hardware, of the first location in the block. The Monitor defines these addresses by loading four 8–bit registers, each of which corresponds to the left eight bits (15–25) of an address whose right ten bits are all 0.

To determine whether an address is legal its left eight bits are compared with the appropriate protection register, so the maximum user address consists of the register contents in its left eight bits, 1777 in its right ten bits (i.e., it is equal to the protection address plus 1777). Since the set of all addresses begins at zero, a block is always an integral multiple of 1024_{10} (2000_8) locations. Relocation is accomplished simply by adding the contents of the appropriate relocation register to the user address, so the first address in a block is a multiple of 2000. The relative user and relocated address configurations are therefore as illustrated here, where P_l, R_l, P_h, and R_h are respectively the protection and relocation addresses for the low and high parts as derived from the 8–bit registers loaded by the Monitor. If the low part is larger than 128K locations, i.e., more than half the maximum memory capacity ($P_l \geq 400000$), the high part starts at the first location after the low part (at location $P_l + 2000$). The high part is limited to 128K. If the Monitor defines two parts but does not write–protect the high part, the user has a two–part nonreentrant program.

If the user attempts to access a location outside of his assigned area, or if the high part is write–protected and he attempts to alter its contents, the current instruction terminates immediately, the Memory Protection flag is set (status bit 22 read by CONI APR,), and an interrupt is requested on the level assigned to the processor (§4.3.6).

Addressing Summary. Let A_u be the address supplied by the user, and let A_p be the physical core address generated from it by the relocation hardware.

If $A_u \leq 17$, then $A_p = A_u$ (fast memory, no relocation).

If $20 \leq A_u \leq P_l + 1777$, then $A_p = (A_u + R_l) \bmod 2^{18}$.

If the greater of $\{ {400000 \atop P_l + 2000} \} \leq A_u \leq P_h + 1777$, then $A_p = (A_u + R_h) \bmod 2^{18}$.

Any other value of A_u is illegal. These are $A_u > P_l + 1777$ if either $A_u < 400000$ or $A_u > P_h + 1777$.

Note: If a relocated address is in the range 0–17, the reference is to core rather than fast memory. (E.g., $R_h = 0$, $P_h = 400000$, and $P_l < 400000$ then relocated references to addresses in the range 400000–400017 actually reference core addresses 0–17.)

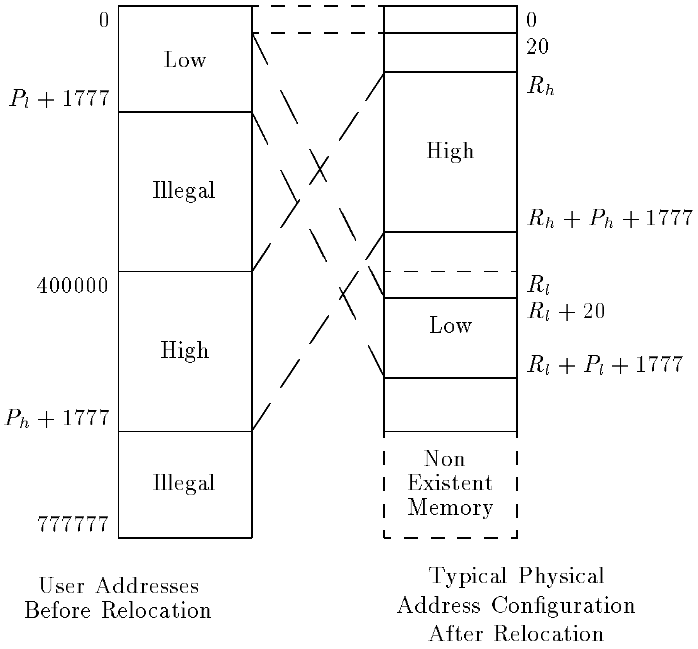
Monitor Programming

The Monitor must assign the core area for each user program, set up trap and interrupt locations,

Figure 4.13: Relocation of User Addresses in the KA10

Note that the relocated low part is actually in two sections with the larger beginning at $R_l + 20$. This is because addresses 0–17 are not relocated, all users having access to the accumulators. The Monitor uses the first sixteen locations in the low user block to store the user's accumulators when his program is not running.

Some systems have only the low pair of protection and relocation registers. In this case the user program is always nonreentrant and the assigned area comprises only the low part.



specify whether the user can give I/O instructions, transfer control to the user program, and respond appropriately when an interrupt occurs or an instruction is executed in unrelocated 41 or 61. Core assignment is made by this instruction.

DATAO APR, Data Out, Arithmetic Processor

70014	I	X	Y
0 12	13	14 17	18 35

Load the protection and relocation registers from the contents of location E as shown, where P_l, P_h, R_l and R_h are the protection and relocation addresses defined above. If write–protect bit P (bit 17) is 1, do not allow the user to write in the high part of his area.

$P_{l_{18-25}}$		$P_{h_{18-25}}$	P	$R_{l_{18-25}}$		$R_{h_{18-25}}$	
0 7	8	9 16	17	18 25	26	27 34	35

Notes. For a two part nonreentrant program, set $P = 0$. For a one–part nonreentrant program, make $P_h \leq P_l$. If the hardware has only one set of protection and relocation registers, the user area is defined by P_l and R_l, the rest of the word is ignored.

Giving a JRSTF with a 1 in bit 6 of the PC word allows the user to handle his own input–output. The Monitor can also transfer control to the user with this instruction by programming a 1 in bit 5 of the PC word, or it may jump to the user program with a JRST 1, which automatically sets User. The set state of this flag implements the user restrictions.

While User is set, certain instructions are not part of the user program and are therefore completely unrestricted, namely those executed in the interrupt locations (which are not relocated) and in unrelocated trap locations 41 and 61. Illegal instructions and UUO codes 000 and 040–077 are trapped in unrelocated 40; codes 100–127 are trapped in unrelocated 60. (The trap locations are 140–141 and 160–161 in a second KA10 processor.) BLKI and BLKO can be used in the even interrupt locations, and if there is no overflow, the processor returns to the interrupted user program. JSR should ordinarily be used in the remaining even interrupt locations, in odd interrupt locations following block I/O instructions, and in 41 and 61. The JSR clears User and should jump to the Monitor. JSP, PUSHJ, JSA and JRST are acceptable in that they clear User, but the first two require an accumulator (all accumulators should be available to the user) and the latter two do not save the flags.

After taking appropriate action, the Monitor can return to the user program with a JRSTF or JEN that restores the flags including User and User In–Out.

4.3.8 Real Time Clock DK10

This processor option can be used to signal the end of a specified real time interval or to measure the real time taken by an event. With appropriate software the DK10 can easily be used to keep the time of day. The basic element in the clock[81] is an 18–bit binary counter that is incremented repeatedly by a clock source; a 100 kHz ± .01% crystal–controlled source is available internally, or a source of any frequency up to 400 kHz can be provided externally. Operation is synchronized so that the program can read the counter at any time without missing a count. Associated with the counter is an 18–bit interval register, which can be loaded by the program. Each time the count reaches the number held in the register, the clock requests an interrupt while the counter clears and begins a new count. With the internal clock source, whose period is 10 μs, the total count is about 2.6 seconds.

The program turns the clock on and off by enabling and disabling the counter. The clock has two modes of operation: with the User Time flag clear, the counter operates continuously; with User Time set, the counter stops while the processor is handling interrupts. Hence in the latter mode the clock discounts interrupt time and can be used to time user programs. In a system that contains two clocks, one can be used by the Monitor to time user programs while the other is used to keep the time of day.

Instructions. The clock device code is 070, mnemonic CLK. A second clock would have device code 074.

---

[81] The clock referred to throughout this section is the DK10 real time clock and should not be confused with the line frequency clock whose flag is one of the processor conditions (§4.3.3 or §4.3.6).

CONO CLK, Conditions Out, Clock

70720	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 23–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

					Set Count Over–flow	Set Count Done	Count	Clear Clock	Clear User Time	Set User Time	Turn Clock Off	Turn Clock On	Clear Count Over–flow	Clear Count Done	Priority Interrupt Assignment		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

A 1 in bit 26 clears the clock counter and the Count Done, Count Overflow and User Time flags, turns off the clock, and drops the PI assignment (assigns zero). The effect of giving conflicting conditions is indeterminate.

A 1 in bit 25 increments the counter provided the clock is off (this is for maintenance only).

CONI CLK, Conditions In, Clock

70724	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the contents of the interval register into the left half of location *E* and read the status of the clock into bits 26–35 as shown (asterisks indicate bits that can cause interrupts).

								Ex–ternal Source		User Time		Clock On	* Count Over–flow	* Count Done	Priority Interrupt Assignment		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Interrupts are requested on the assigned level by the setting of Count Overflow and Count Done.

26 The counter is connected to an external source (0 indicates the internal source is connected).

28 The counter cannot be incremented while an interrupt is being held or a request has been accepted and the level is waiting for an interrupt to start. Note that to time a user properly, the Monitor must also compensate for any noninterrupt time taken from the user.

DATAO CLK, Data Out, Clock

70714	*I*	*X*	*Y*
0 12	13	14 17	18 35

Load the contents of the right half of location E into the interval register.

Notes. The comparison of the counter against the interval register that follows every count is inhibited while this instruction is loading the register.

DATAI CLK, Data In, Clock

70704	*I*	*X*	*Y*
0 12	13	14 17	18 35

Read the current contents of the clock counter into the right half of location E.

Notes. The counter is always stable while being read, and any count held back is picked up immediately afterward.

Initially the program should give a CONO CLK,1000 to clear the clock, and then give a DATAO to select the interval and a CONO to turn on the clock, select the mode, and assign the interrupt level. Following turn on the first count may occur at any time up to the full period of the source. When the count reaches the specified interval, Count Done sets, requesting an interrupt on the assigned level. At the same time, the counter clears and a new count begins with the next pulse. The program should respond with a CONO to clear Count Done. Remember that although a CONO need not affect the mode or the clock state, every CONO must renew the PI assignment.

The interval can be changed at any time simply by giving a DATAO However, if the program does not clear the counter at the same time, then it should make sure that the count has not yet reached the value of the new interval. If the count is already beyond that point, the counter will continue until it overflows. When the counter overflows, either because the count started too high, the program specified the maximum count (2^{18} is selected by loading zero), or there is a malfunction of some sort, Count Overflow sets, requesting an interrupt, and a new count begins.

To use the clock to time some operation, turn it on with the counter at zero. For a counter reading of C, the elapsed time is

$$T(C + nI)$$

where T is the period of the source, n is the number of clock interrupts since the clock was started, and I is the interval selected by the program. To cause the clock to request an interrupt after $T \times n$ μs, where $n \leq 2^{18}$ and T is the period of the source in microseconds, load the interval register with n expressed in binary. There is an average indeterminacy of half a count every time the counter starts and stops. Therefore, when the clock is keeping user time, there is an average indeterminacy of one count for every *group* of overlapping interrupts any requests (not for every interrupt, as the counter is inhibited while there is any request or interrupt being held).

For keeping the time of day, the program can use a memory location to maintain a count of the clock interrupts. The location should be cleared at midnight — note that an error of .01% amounts to 8.64 seconds in 24 hours — and the time can be determined by combining its contents with the current contents of the clock counter. If the location itself is to be used as a low resolution clock kept in hours, minutes and seconds, it is better to use a more convenient interval than the full count. Using the internal source, an interval of 2.5 seconds, which is octal 750220, is the most straightforward interval with the fewest interrupts. To interrupt every second the interval would be 303240.

Appendix A

Instructions and Mnemonics

A.1 Formats

The diagrams below show the formats of the various types of instructions, pointers, arithmetic operands, and other special words employed by the user in the KL10 processor. All of these apply to the XKL-1 processor, with the sole exception of the In–Out instruction format. Most of these formats apply to the KS10 processor, except extended addressing, In–Out instructions, and giant–range floating–point operands.

A.1.1 Instruction Words

Basic Instructions

Instruction Code	*A, F*	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

In–Out Instructions

7	Device Code	Funct	*I*	*X*	*Y*
0 2	3 9	10 12	13	14 17	18 35

Instructions Executed Under EXTEND

Instruction Code	0	*I*	*X*	*Y*
0 8	9 12	13	14 17	18 35

A.1.2 Address and Program Control Words

Local Indirect Word

1	0	Reserved	I	X	Y
0	1	2 … 12	13	14 … 17	18 … 35

Global Indirect Word

0	I	X	Y
0	1	2 … 5	6 … 35

Local Index Register

In Non–zero Section must be ≤ 0 or Bits 6–17 = 0	Local Index
0 … 17	18 … 35

Global Index Register

<table>
<tr><td rowspan="2">0</td><td rowspan="2">Available to Software</td><td colspan="2">Global Index</td></tr>
<tr><td>Non–Zero Section Number</td><td></td></tr>
<tr><td>0</td><td>1 … 5</td><td>6 … 17</td><td>… 35</td></tr>
</table>

An index register used in a global indirect word always uses bits 6–35.

Saved Flags

Overflow Previous Context Public	Carry 0	Carry 1	Floating Overflow	First Part Done	User	User In–Out Previous Context User	Public	Address Failure Inhibit	Trap 2	Trap 1	Floating Under–flow	No Divide
0	1	2	3	4	5	6	7	8	9	10	11	12

PC Word

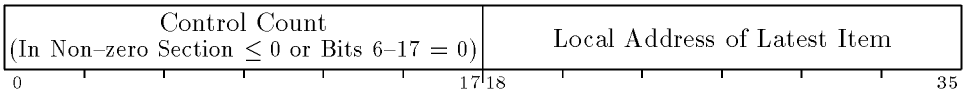
Flags	00	In–Section PC
0 … 12	13 … 17	18 … 35

Flag–PC Double Word

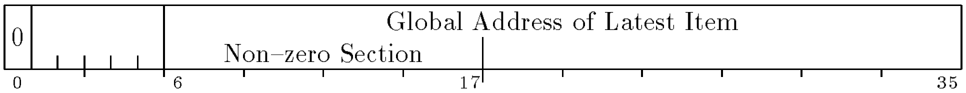
<table>
<tr><td colspan="2">Flags</td><td>0</td><td>Processor–Dependent Information</td></tr>
<tr><td>00</td><td colspan="3">PC</td></tr>
<tr><td>0 … 5</td><td>6 … 12</td><td>13 … 17</td><td>18 … 35</td></tr>
</table>

A.1.3 Stack, Byte Pointers

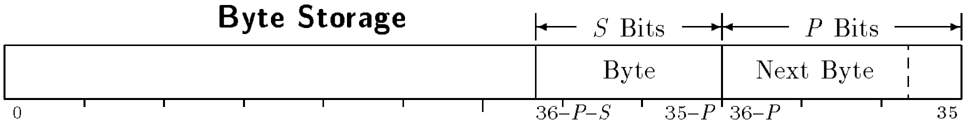
Local Stack Pointer



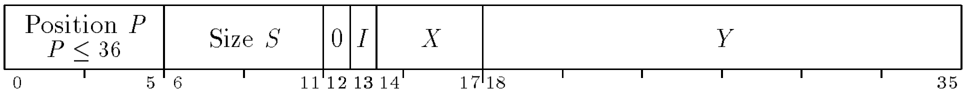
Global Stack Pointer



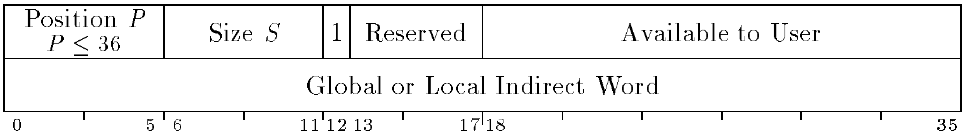
Byte Storage



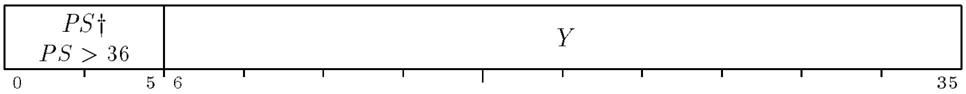
Local Byte Pointer



Two–Word Global or Local Byte Pointer



One–Word Global Byte Pointer



† To decode PS into P and S, use the following table, in which the values of P and S are given in decimal and PS is given in octal.

PS	*P*	*S*	*PS*	*P*	*S*	*PS*	*P*	*S*
45	36	6	56	20	8	67	36	9
46	30	6	57	12	8	70	27	9
47	24	6	60	4	8	71	18	9
50	18	6	61	36	7	72	9	9
51	12	6	62	29	7	73	0	9
52	6	6	63	22	7	74	36	18
53	0	6	64	15	7	75	18	18
54	36	8	65	8	7	76	0	18
55	28	8	66	1	7	77	Illegal	

A.1.4 Arithmetic Operands

Fixed Point Operands (Single Precision)

Sign 0+ 1−	Binary Number (Twos Complement)
0	1 35

Fixed Point Operands (Double Precision)

Sign 0+ 1−	High Order Part of Binary Number (Twos Complement)
Sign Copy	Low Order Part of Binary Number (Twos Complement)
0	1 35

Single Precision Floating Point Operands

Sign 0+ 1−	Excess 128 Exponent (Ones Complement)	Fraction (Twos Complement)
0	1 8	9 35

Double Precision Floating Point Operands

Sign 0+ 1−	Excess 128 Exponent (Ones Complement)	High Order Fraction (Twos Complement)
0	Low Order Extension of Fraction (Twos Complement)	
0	1 8	9 35

Giant–Range Floating Point Operands

Sign 0+ 1−	Excess 1024 Exponent (Ones Complement)	High Order Fraction (Twos Complement)
0	Low Order Extension of Fraction (Twos Complement)	
0	1 11	12 35

A.2 Instruction Mnemonics – Numeric Listing

PDP–10 Instruction Set

	0	1	2	3	4	5	6	7
000	MUUO	LUUO CMPSL†	LUUO CMPSE†	LUUO CMPSLE†	LUUO EDIT†	LUUO CMPSGE†	LUUO CMPSN†	LUUO CMPSG†
010	LUUO CVTDBO†	LUUO CVTDBT†	LUUO CVTBDO†	LUUO CVTBDT†	LUUO MOVSO†	LUUO MOVST†	LUUO MOVSLJ†	LUUO MOVSRJ†
020	LUUO XBLT†	LUUO GSNGL†	LUUO GDBLE†	LUUO GDFIX†	LUUO GFIX†	LUUO GDFIXR†	LUUO GFIXR†	LUUO DGFLTR†
030	LUUO GFLTR†	LUUO GFSC†	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO
040	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
050	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
060	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
070	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
100	MUUO	MUUO	GFAD	GFSB	JSYS	ADJSP	GFMP	GFDV
110	DFAD	DFSB	DFMP	DFDV	DADD	DSUB	DMUL	DDIV
120	DMOVE	DMOVN	FIX	EXTEND	DMOVEM	DMOVNM	FIXR	FLTR
130	UFA§	DFN§	FSC	IBP	ILDB	LDB	IDPB	DPB
140	FAD	FADL§	FADM	FADB	FADR	FADRI	FADRM	FADRB
150	FSB	FSBL§	FSBM	FSBB	FSBR	FSBRI	FSBRM	FSBRB
160	FMP	FMPL§	FMPM	FMPB	FMPR	FMPRI	FMPRM	FMPRB
170	FDV	FDVL§	FDVM	FDVB	FDVR	FDVRI	FDVRM	FDVRB
200	MOVE	MOVEI	MOVEM	MOVES	MOVS	MOVSI	MOVSM	MOVSS
210	MOVN	MOVNI	MOVNM	MOVNS	MOVM	MOVMI	MOVMM	MOVMS
220	IMUL	IMULI	IMULM	IMULB	MUL	MULI	MULM	MULB
230	IDIV	IDIVI	IDIVM	IDIVB	DIV	DIVI	DIVM	DIVB
240	ASH	ROT	LSH	JFFO	ASHC	ROTC	LSHC	MUUO
250	EXCH	BLT	AOBJP	AOBJN	JRST	JFCL	XCT	MAP
260	PUSHJ	PUSH	POP	POPJ	JSR	JSP	JSA§	JRA§
270	ADD	ADDI	ADDM	ADDB	SUB	SUBI	SUBM	SUBB

† This instruction is available only under `EXTEND`. § This instruction is obsolete.

PDP–10 Instruction Set

	0	1	2	3	4	5	6	7
300	CAI	CAIL	CAIE	CAILE	CAIA	CAIGE	CAIN	CAIG
310	CAM	CAML	CAME	CAMLE	CAMA	CAMGE	CAMN	CAMG
320	JUMP	JUMPL	JUMPE	JUMPLE	JUMPA	JUMPGE	JUMPN	JUMPG
330	SKIP	SKIPL	SKIPE	SKIPLE	SKIPA	SKIPGE	SKIPN	SKIPG
340	AOJ	AOJL	AOJE	AOJLE	AOJA	AOJGE	AOJN	AOJG
350	AOS	AOSL	AOSE	AOSLE	AOSA	AOSGE	AOSN	AOSG
360	SOJ	SOJL	SOJE	SOJLE	SOJA	SOJGE	SOJN	SOJG
370	SOS	SOSL	SOSE	SOSLE	SOSA	SOSGE	SOSN	SOSG
400	SETZ	SETZI	SETZM	SETZB	AND	ANDI	ANDM	ANDB
410	ANDCA	ANDCAI	ANDCAM	ANDCAB	SETM	SETMI XMOVEI[‡]	SETMM	SETMB
420	ANDCM	ANDCMI	ANDCMM	ANDCMB	SETA	SETAI	SETAM	SETAB
430	XOR	XORI	XORM	XORB	IOR	IORI	IORM	IORB
440	ANDCB	ANDCBI	ANDCBM	ANDCBB	EQV	EQVI	EQVM	EQVB
450	SETCA	SETCAI	SETCAM	SETCAB	ORCA	ORCAI	ORCAM	ORCAB
460	SETCM	SETCMI	SETCMM	SETCMB	ORCM	ORCMI	ORCMM	ORCMB
470	ORCB	ORCBI	ORCBM	ORCBB	SETO	SETOI	SETOM	SETOB
500	HLL	HLLI XHLLI[‡]	HLLM	HLLS	HRL	HRLI	HRLM	HRLS
510	HLLZ	HLLZI	HLLZM	HLLZS	HRLZ	HRLZI	HRLZM	HRLZS
520	HLLO	HLLOI	HLLOM	HLLOS	HRLO	HRLOI	HRLOM	HRLOS
530	HLLE	HLLEI	HLLEM	HLLES	HRLE	HRLEI	HRLEM	HRLES
540	HRR	HRRI	HRRM	HRRS	HLR	HLRI	HLRM	HLRS
550	HRRZ	HRRZI	HRRZM	HRRZS	HLRZ	HLRZI	HLRZM	HLRZS
560	HRRO	HRROI	HRROM	HRROS	HLRO	HLROI	HLROM	HLROS
570	HRRE	HRREI	HRREM	HRRES	HLRE	HLREI	HLREM	HLRES

[‡] This operation is available only in non-zero sections.

PDP–10 Instruction Set

	0	1	2	3	4	5	6	7
600	TRN	TLN	TRNE	TLNE	TRNA	TLNA	TRNN	TLNN
610	TDN	TSN	TDNE	TSNE	TDNA	TSNA	TDNN	TSNN
620	TRZ	TLZ	TRZE	TLZE	TRZA	TLZA	TRZN	TLZN
630	TDZ	TSZ	TDZE	TSZE	TDZA	TSZA	TDZN	TSZN
640	TRC	TLC	TRCE	TLCE	TRCA	TLCA	TRCN	TLCN
650	TDC	TSC	TDCE	TSCE	TDCA	TSCA	TDCN	TSCN
660	TRO	TLO	TROE	TLOE	TROA	TLOA	TRON	TLON
670	TDO	TSO	TDOE	TSOE	TDOA	TSOA	TDON	TSON
700	APR0	APR1	APR2	APR3	PMOVE	PMOVEM	NMOVE	NMOVEM
710	LDLPN	RDCFG	MUUO	MUUO	AMOVE	AMOVEM	UMOVE	UMOVEM
720	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
730	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
740	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
750	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
760	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
770	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO

A.2.1 APR0, APR1, APR2, and APR3 Instructions (XKL-1 only)

AC	APR0	APR1	APR2	APR3
00	APRID		RDSPB	
01	RDADB	RDUBR	RDCSB	RDCTY
02	SYSID	CLRPT	RDPUR	
03	WRADB	WRUBR	RDCSTM	WRCTY
04	WRAPR	WREBR	RDITM	WRCTYS
05	RDAPR	RDEBR	RDTIME	RDCTYS
06	SZAPR	WRCTX	DRDPTB	SZCTYS
07	SNAPR	RDCTX	WRTIME	SNCTYS
10	WCTRLF	DRDCSH	WRSPB	
11	RCTRLF	SWPIA	WRCSB	
12	SIMIRD	SWPVA	WRPUR	
13	WRKPA	SWPUA	WRCSTM	
14	WRPI	DWRCSH	WRITM	
15	RDPI	SWPIO		
16	SZPI	SWPVO	DWRPTB	
17	SNPI	SWPUO		

A.3 Instruction Mnemonics – Alphabetic Listing

⋆ Accumulator field must be non–zero.
‡ Operation is not available in section zero.
† Operation exists only under `EXTEND`.
§ Operation is obsolete.

ADD	270000,,0
ADDB	273000,,0
ADDI	271000,,0
ADDM	272000,,0
ADJBP ⋆	133000,,0
ADJSP	105000,,0
AMOVE	714000,,0
AMOVEM	715000,,0
AND	404000,,0
ANDB	407000,,0
ANDCA	410000,,0
ANDCAB	413000,,0
ANDCAI	411000,,0
ANDCAM	412000,,0
ANDCB	440000,,0
ANDCBB	443000,,0
ANDCBI	441000,,0
ANDCBM	442000,,0
ANDCM	420000,,0
ANDCMB	423000,,0
ANDCMI	421000,,0
ANDCMM	422000,,0
ANDI	405000,,0
ANDM	406000,,0
AOBJN	253000,,0
AOBJP	252000,,0
AOJ	340000,,0
AOJA	344000,,0
AOJE	342000,,0
AOJG	347000,,0
AOJGE	345000,,0
AOJL	341000,,0
AOJLE	343000,,0
AOJN	346000,,0
AOS	350000,,0
AOSA	354000,,0
AOSE	352000,,0
AOSG	357000,,0
AOSGE	355000,,0
AOSL	351000,,0
AOSLE	353000,,0
AOSN	356000,,0
APR0	700000,,0
APR1	701000,,0
APR2	702000,,0
APR3	703000,,0
APRID	700000,,0
ASH	240000,,0
ASHC	244000,,0
BLKI §	7xx000,,0
BLKO §	7xx100,,0
BLT	251000,,0
CAI	300000,,0
CAIA	304000,,0
CAIE	302000,,0
CAIG	307000,,0
CAIGE	305000,,0
CAIL	301000,,0
CAILE	303000,,0
CAIN	306000,,0
CAM	310000,,0
CAMA	314000,,0
CAME	312000,,0
CAMG	317000,,0
CAMGE	315000,,0
CAML	311000,,0
CAMLE	313000,,0
CAMN	316000,,0
CLRPT	701100,,0
CMPSE †	002000,,0
CMPSG †	007000,,0
CMPSGE †	005000,,0
CMPSL †	001000,,0
CMPSLE †	003000,,0
CMPSN †	006000,,0
CONI §	7xx240,,0
CONO §	7xx200,,0
CONSO §	7xx340,,0
CONSZ §	7xx300,,0
CVTBDO †	012000,,0
CVTBDT †	013000,,0
CVTDBO †	010000,,0
CVTDBT †	011000,,0
DADD	114000,,0
DATAI §	7xx040,,0
DATAO §	7xx140,,0
DDIV	117000,,0
DFAD	110000,,0
DFDV	113000,,0
DFMP	112000,,0
DFN §	131000,,0
DFSB	111000,,0
DGFLTR †	027000,,0
DIV	234000,,0
DIVB	237000,,0
DIVI	235000,,0
DIVM	236000,,0
DMOVE	120000,,0
DMOVEM	124000,,0
DMOVN	121000,,0
DMOVNM	125000,,0
DMUL	116000,,0
DPB	137000,,0
DRDCSH	701400,,0
DRDPTB	702300,,0
DSUB	115000,,0
DWRCSH	701600,,0
DWRPTB	702700,,0
EDIT †	004000,,0
EQV	444000,,0
EQVB	447000,,0
EQVI	445000,,0
EQVM	446000,,0
EXCH	250000,,0
EXTEND	123000,,0
FAD	140000,,0
FADB	143000,,0
FADL §	141000,,0
FADM	142000,,0
FADR	144000,,0
FADRB	147000,,0
FADRI	145000,,0
FADRM	146000,,0
FDV	170000,,0
FDVB	173000,,0
FDVL §	171000,,0
FDVM	172000,,0
FDVR	174000,,0
FDVRB	177000,,0
FDVRI	175000,,0
FDVRM	176000,,0
FIX	122000,,0

Instruction Mnemonics — Alphabetic Listing (Continued)

⋆ Accumulator field must be non–zero. † Operation exists only under EXTEND.
‡ Operation is not available in section zero. § Operation is obsolete.

FIXR	126000,,0
FLTR	127000,,0
FMP	160000,,0
FMPB	163000,,0
FMPL §	161000,,0
FMPM	162000,,0
FMPR	164000,,0
FMPRB	167000,,0
FMPRI	165000,,0
FMPRM	166000,,0
FSB	150000,,0
FSBB	153000,,0
FSBL §	151000,,0
FSBM	152000,,0
FSBR	154000,,0
FSBRB	157000,,0
FSBRI	155000,,0
FSBRM	156000,,0
FSC	132000,,0
GDBLE †	022000,,0
GDFIX †	023000,,0
GDFIXR †	025000,,0
GFAD	102000,,0
GFDV	107000,,0
GFIX †	024000,,0
GFIXR †	026000,,0
GFLTR †	030000,,0
GFMP	106000,,0
GFSB	103000,,0
GFSC †	021000,,0
GSNGL †	021000,,0
HALT	254200,,0
HALTRM	254540,,0
HLL	500000,,0
HLLE	530000,,0
HLLEI	531000,,0
HLLEM	532000,,0
HLLES	533000,,0
HLLI	501000,,0
HLLM	502000,,0
HLLO	520000,,0
HLLOI	521000,,0
HLLOM	522000,,0
HLLOS	523000,,0
HLLS	503000,,0
HLLZ	510000,,0
HLLZI	511000,,0
HLLZM	512000,,0
HLLZS	513000,,0
HLR	544000,,0
HLRE	574000,,0
HLREI	575000,,0
HLREM	576000,,0
HLRES	577000,,0
HLRI	545000,,0
HLRM	546000,,0
HLRO	564000,,0
HLROI	565000,,0
HLROM	566000,,0
HLROS	567000,,0
HLRS	547000,,0
HLRZ	554000,,0
HLRZI	555000,,0
HLRZM	556000,,0
HLRZS	557000,,0
HRL	504000,,0
HRLE	534000,,0
HRLEI	535000,,0
HRLEM	536000,,0
HRLES	537000,,0
HRLI	505000,,0
HRLM	506000,,0
HRLO	524000,,0
HRLOI	525000,,0
HRLOM	526000,,0
HRLOS	527000,,0
HRLS	507000,,0
HRLZ	514000,,0
HRLZI	515000,,0
HRLZM	516000,,0
HRLZS	517000,,0
HRR	540000,,0
HRRE	570000,,0
HRREI	571000,,0
HRREM	572000,,0
HRRES	573000,,0
HRRI	541000,,0
HRRM	542000,,0
HRRO	560000,,0
HRROI	561000,,0
HRROM	562000,,0
HRROS	563000,,0
HRRS	543000,,0
HRRZ	550000,,0
HRRZI	551000,,0
HRRZM	552000,,0
HRRZS	553000,,0
IBP	133000,,0
IDIV	230000,,0
IDIVB	233000,,0
IDIVI	231000,,0
IDIVM	232000,,0
IDPB	136000,,0
ILDB	134000,,0
IMUL	220000,,0
IMULB	223000,,0
IMULI	221000,,0
IMULM	222000,,0
IOR	434000,,0
IORB	437000,,0
IORI	435000,,0
IORM	436000,,0
JCRY	255300,,0
JCRY0	255200,,0
JCRY1	255100,,0
JEN	254500,,0
JFCL	255000,,0
JFFO	243000,,0
JFOV	255040,,0
JOV	255400,,0
JRA §	267000,,0
JRST	254000,,0
JRSTF	254100,,0
JSA §	266000,,0
JSP	265000,,0
JSR	264000,,0
JSYS	104000,,0
JUMP	320000,,0
JUMPA	324000,,0
JUMPE	322000,,0
JUMPG	327000,,0
JUMPGE	325000,,0
JUMPL	321000,,0
JUMPLE	323000,,0
JUMPN	326000,,0

Instruction Mnemonics — Alphabetic Listing (Continued)

⋆ Accumulator field must be non–zero. † Operation exists only under EXTEND.
‡ Operation is not available in section zero. § Operation is obsolete.

LDB	135000,,0
LDLPN	710000,,0
LSH	242000,,0
LSHC	246000,,0
MAP	257000,,0
MOVE	200000,,0
MOVEI	201000,,0
MOVEM	202000,,0
MOVES	203000,,0
MOVM	214000,,0
MOVMI	215000,,0
MOVMM	216000,,0
MOVMS	217000,,0
MOVN	210000,,0
MOVNI	211000,,0
MOVNM	212000,,0
MOVNS	213000,,0
MOVS	204000,,0
MOVSI	205000,,0
MOVSLJ †	016000,,0
MOVSM	206000,,0
MOVSO †	014000,,0
MOVSRJ †	017000,,0
MOVSS	207000,,0
MOVST †	015000,,0
MUL	224000,,0
MULB	227000,,0
MULI	225000,,0
MULM	226000,,0
NMOVE	706000,,0
NMOVEM	707000,,0
ORCA	454000,,0
ORCAB	457000,,0
ORCAI	455000,,0
ORCAM	456000,,0
ORCB	470000,,0
ORCBB	473000,,0
ORCBI	471000,,0
ORCBM	472000,,0
ORCM	464000,,0
ORCMB	467000,,0
ORCMI	465000,,0
ORCMM	466000,,0
PMOVE	704000,,0
PMOVEM	705000,,0
POP	262000,,0
POPJ	263000,,0
PORTAL	254040,,0
PUSH	261000,,0
PUSHJ	260000,,0
PXCT ⋆	256000,,0
RCTRLF	700440,,0
RDADB	700040,,0
RDAPR	700240,,0
RDCFG	711000,,0
RDCSB	702040,,0
RDCSTM	702140,,0
RDCTX	701340,,0
RDCTY	703040,,0
RDCTYS	703240,,0
RDEBR	701240,,0
RDITM	702200,,0
RDPI	700640,,0
RDPUR	702100,,0
RDSPB	702000,,0
RDTIME	702240,,0
RDUBR	701040,,0
ROT	241000,,0
ROTC	245000,,0
SETA	424000,,0
SETAB	427000,,0
SETAI	425000,,0
SETAM	426000,,0
SETCA	450000,,0
SETCAB	453000,,0
SETCAI	451000,,0
SETCAM	452000,,0
SETCM	460000,,0
SETCMB	463000,,0
SETCMI	461000,,0
SETCMM	462000,,0
SETM	414000,,0
SETMB	417000,,0
SETMI	415000,,0
SETMM	416000,,0
SETO	474000,,0
SETOB	477000,,0
SETOI	475000,,0
SETOM	476000,,0
SETZ	400000,,0
SETZB	403000,,0
SETZI	401000,,0
SETZM	402000,,0
SFM	254600,,0
SIMIRD	700500,,0
SKIP	330000,,0
SKIPA	334000,,0
SKIPE	332000,,0
SKIPG	337000,,0
SKIPGE	335000,,0
SKIPL	331000,,0
SKIPLE	333000,,0
SKIPN	336000,,0
SNAPR	700340,,0
SNCTYS	703340,,0
SNPI	700740,,0
SOJ	360000,,0
SOJA	364000,,0
SOJE	362000,,0
SOJG	367000,,0
SOJGE	365000,,0
SOJL	361000,,0
SOJLE	363000,,0
SOJN	366000,,0
SOS	370000,,0
SOSA	374000,,0
SOSE	372000,,0
SOSG	377000,,0
SOSGE	375000,,0
SOSL	371000,,0
SOSLE	373000,,0
SOSN	376000,,0
SUB	274000,,0
SUBB	277000,,0
SUBI	275000,,0
SUBM	276000,,0
SWPIA	701440,,0
SWPIO	701640,,0
SWPUA	701540,,0
SWPUO	701740,,0
SWPVA	701500,,0
SWPVO	701700,,0
SYSID	700100,,0
SZAPR	700300,,0
SZCTYS	703300,,0

Instruction Mnemonics — Alphabetic Listing (Continued)

⋆ Accumulator field must be non–zero. † Operation exists only under EXTEND.
‡ Operation is not available in section zero. § Operation is obsolete.

SZPI	700700,,0
TDC	650000,,0
TDCA	654000,,0
TDCE	652000,,0
TDCN	656000,,0
TDN	610000,,0
TDNA	614000,,0
TDNE	612000,,0
TDNN	616000,,0
TDO	670000,,0
TDOA	674000,,0
TDOE	672000,,0
TDON	676000,,0
TDZ	630000,,0
TDZA	634000,,0
TDZE	632000,,0
TDZN	636000,,0
TLC	641000,,0
TLCA	645000,,0
TLCE	643000,,0
TLCN	647000,,0
TLN	601000,,0
TLNA	605000,,0
TLNE	603000,,0
TLNN	607000,,0
TLO	661000,,0
TLOA	665000,,0
TLOE	663000,,0
TLON	667000,,0
TLZ	621000,,0
TLZA	625000,,0
TLZE	623000,,0
TLZN	627000,,0
TRC	640000,,0
TRCA	644000,,0
TRCE	642000,,0
TRCN	646000,,0
TRN	600000,,0
TRNA	604000,,0
TRNE	602000,,0
TRNN	606000,,0
TRO	660000,,0
TROA	664000,,0
TROE	662000,,0
TRON	666000,,0
TRZ	620000,,0
TRZA	624000,,0
TRZE	622000,,0
TRZN	626000,,0
TSC	651000,,0
TSCA	655000,,0
TSCE	653000,,0
TSCN	657000,,0
TSN	611000,,0
TSNA	615000,,0
TSNE	613000,,0
TSNN	617000,,0
TSO	671000,,0
TSOA	675000,,0
TSOE	673000,,0
TSON	677000,,0
TSZ	631000,,0
TSZA	635000,,0
TSZE	633000,,0
TSZN	637000,,0
UFA §	130000,,0
UMOVE	716000,,0
UMOVEM	717000,,0
WCTRLF	700400,,0
WRADB	700140,,0
WRAPR	700200,,0
WRCSB	702440,,0
WRCSTM	702540,,0
WRCTX	701300,,0
WRCTY	703140,,0
WRCTYS	703200,,0
WREBR	701200,,0
WRITM	702600,,0
WRKPA	700540,,0
WRPI	700600,,0
WRPUR	702500,,0
WRSPB	702400,,0
WRTIME	702340,,0
WRUBR	701140,,0
XBLT †	020000,,0
XCT	256000,,0
XHLLI ‡	501000,,0
XJEN	254300,,0
XJRST	254640,,0
XJRSTF	254240,,0
XJRSTP	254440,,0
XMOVEI ‡	415000,,0
XOR	430000,,0
XORB	433000,,0
XORI	431000,,0
XORM	432000,,0
XPCW	254340,,0

A.4 Algebraic Representation

[To be supplied]

A.5 Powers of Two

2^N	N	2^{-N}
1	0	1.
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

Appendix B

Character Codes

The following chart presents the 7–bit ASCII code now used in these systems. Various versions of this code dating back to 1965 were used at various points in the evolution of the DECsystem–10, and the confusion that has caused continues to this day.

The ASCII Character Set

	0	1	2	3	4	5	6	7
000	NUL							BEL
010	BS	HT	LF	VT	FF	CR		
020								
030				ESC				
040	SP	!	"	#	$	%	&	’
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	‘	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	DEL

Special Characters	
NUL	Null Character
BEL	Bell
BS	Backspace
HT	Horizontal Tab
LF	Line Feed
VT	Vertical Tab
FF	Form Feed
CR	Carriage Return
ESC	Escape
SP	Space
DEL	Delete

Appendix C

Processor Compatibility

This appendix details some differences between the KL10 processor and the XKL-1 processor. These differences can be discerned in user mode. The executive mode differences are too numerous to include here. The chapter devoted to XKL-1 processor System Operations should be compared to the section on KL10 System Operations.

- In the BLT instruction, the KL10 computes the ending value for the BLT AC and stores it in AC before moving any data. The XKL-1 processor stores the BLT AC at the end of the instruction (unless AC and E are equal).

  ```
  MOVSI 0,400          ;put 400,,0 in 0
  BLT 0,1              ;copy contents of 400 to 0 and contents of 401 to 1
  ```

  In the KL10, the result is indeterminate: usually, a copy of 400 will be found in AC 0, however, an interrupt may intervene, which would result in AC 0 containing 402000002 after the instruction. In the XKL-1 processor, AC 0 will always contain 402000002 after the instruction.

  As mentioned in the description of BLT, this construct is defined as being indeterminate. It should be avoided.

  This difference between the KL10 and the XKL-1 processor is shown most distinctly in the following:

  ```
  SETZ 0,
  BLT 0,0
  ```

  In the KL10, the result in 0 is 1,,1; in the XKL-1 processor it is zero. The KL10 computes the ending value of the BLT AC and stores it in 0 before copying 0 to 0. The XKL-1 processor does not store the ending value, because it recognizes that the last address stored in is the BLT AC.

- The GFSC instruction may signal extreme underflow as overflow and extreme overflow as underflow in the KL10. In the XKL-1 processor, extreme underflow is reported as underflow, and extreme overflow is reported as overflow.
- The GSNGL instruction in the KL10 stores a result in AC when the G–format operand has an exponent in the range 1570_8 to 1577_8. The XKL-1 processor does not affect AC in this case.
- A LDB using a one–word global byte pointer that specifies the non–existent byte to the left of the real bytes in a word produces different results on the KL10 and the XKL-1 processor. In the KL10, the effect is to copy the byte pointer to AC. In the XKL-1 processor, the result is zero in AC.

  ```
  LDB AC,[610000,,0]
  ```

  In the KL10, AC will contain a copy of AC 0 (the address specified in bits 6–35 of the byte pointer). In the XKL-1 processor, AC will contain 0.
- In the following sequence, in which AC is not 0,

  ```
  MOVSI AC,400000        ;the negative number of largest magitude
  ADJBP AC,[430100,,0]   ;POINT 1,0,0
  ```

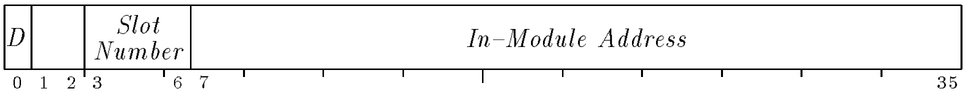
  the KL10 copies the byte pointer to AC. The XKL-1 processor performs the indicated adjustment and stores the result in AC.
- In the KL10, DFMP does rounding in a manner other than that described in this manual. After normalization, if the 1/2 LSB is one, one is added to the LSB. For results that are negative, this varies from the described behavior in the case where 1/2 LSB is one and there are no other bits that are 1 to the right of the 1/2 LSB.

Appendix D

Internal Device Bit Assignments

D.1 XKL-1 processor Internal Device Bit Assignments

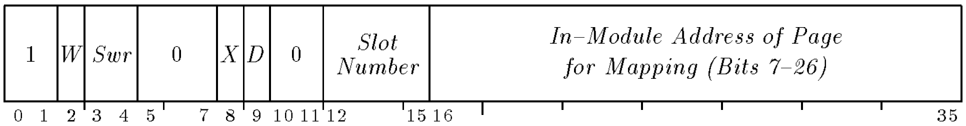
Bus Address Word:



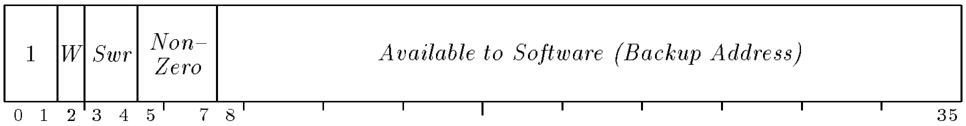
Page map pointer word formats (supersection pointers and section pointers are similar):

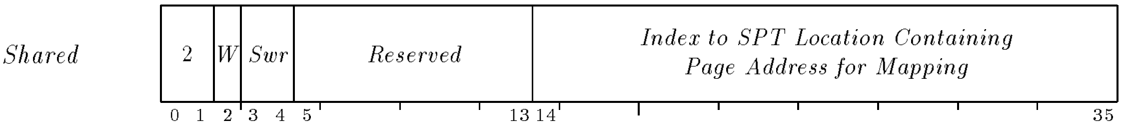
No Access

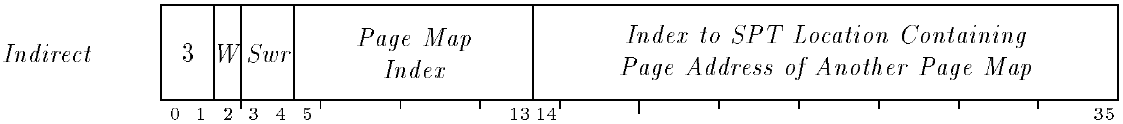
Immediate In–Memory



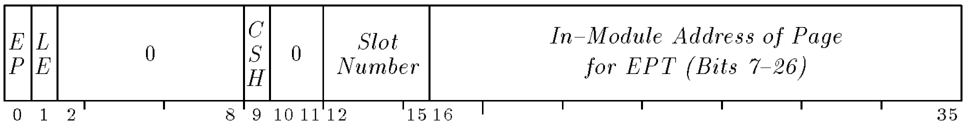
Immediate Not In–Memory



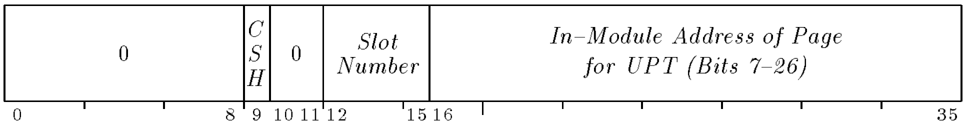




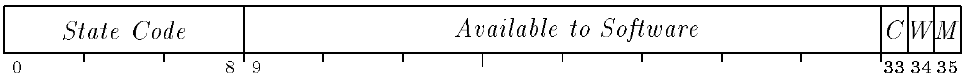
Data for WREBR:



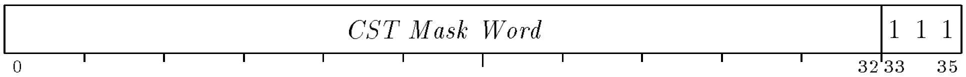
Data for WRUBR:



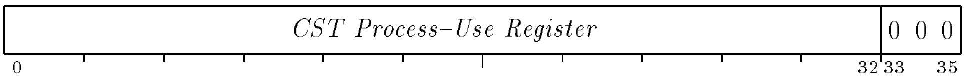
CST entry format. State codes in the range 0–7 cause “age” traps.



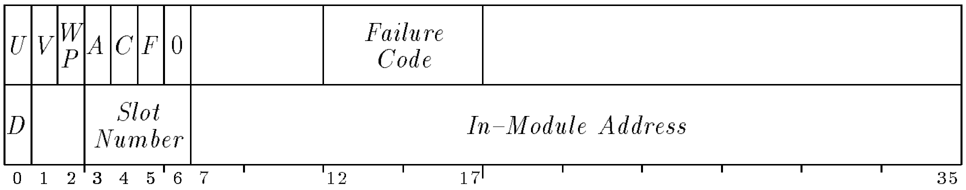
Data for WRCSTM



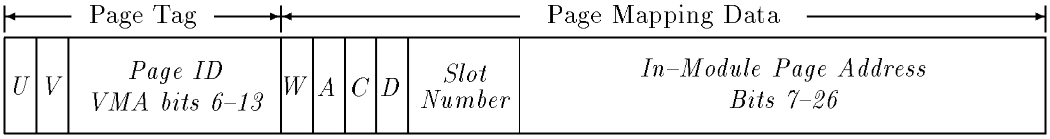
Data for WRPUR



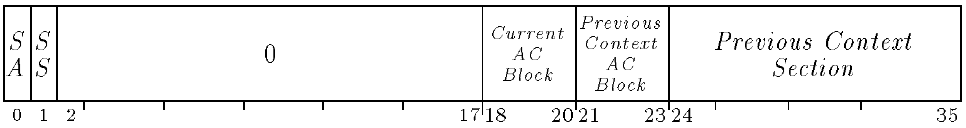
Page–Failure or MAP double word:



Pager Translation Buffer (PTB) tag and data format:

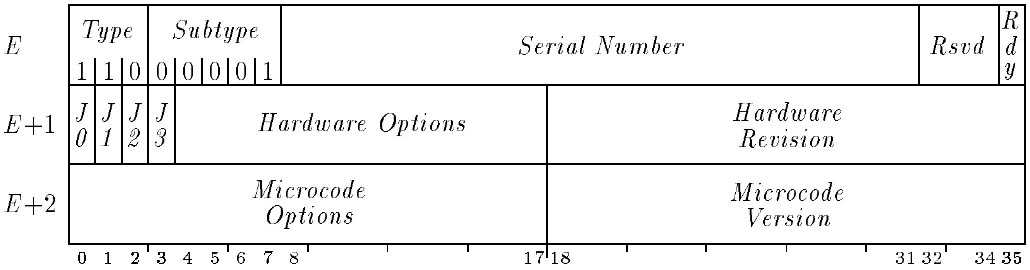


Data for WRCTX

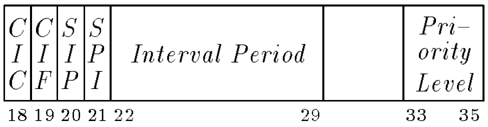


Data for APRID

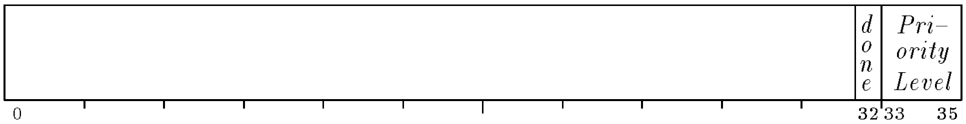
Processor Identification Tripleword



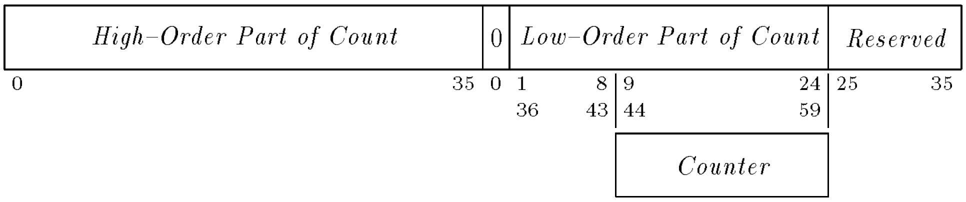
Data for WRITM



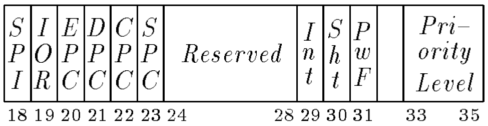
Data for RDITM



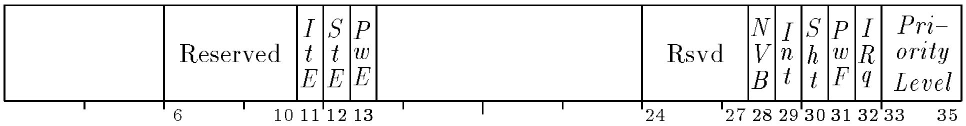
Data for RDTIME



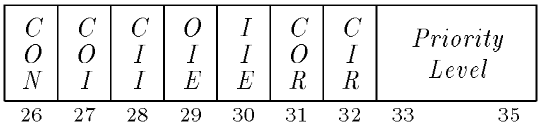
Data for WRAPR



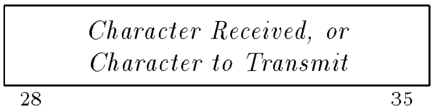
Data for RDAPR



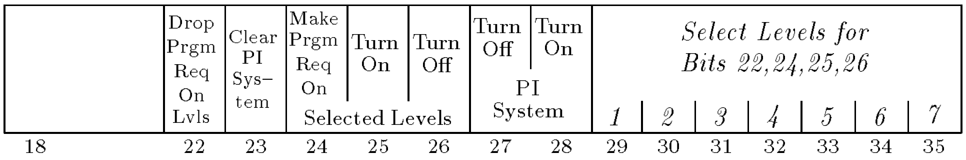
Data for RDCTYS, WRCTYS, SZCTYS, and SNCTYS



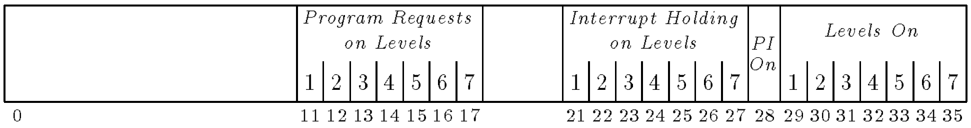
Data for RDCTY or WRCTY



Data for WRPI

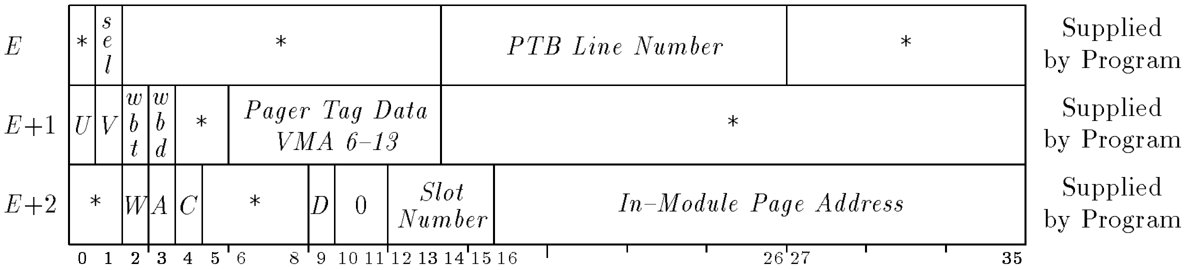


Data for RDPI



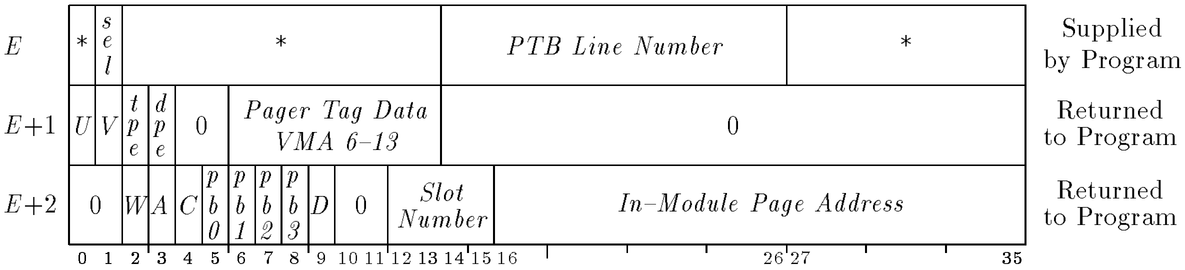
Data for DWRPTB

Page Translation Buffer Tag and Data Tripleword — DWRPTB



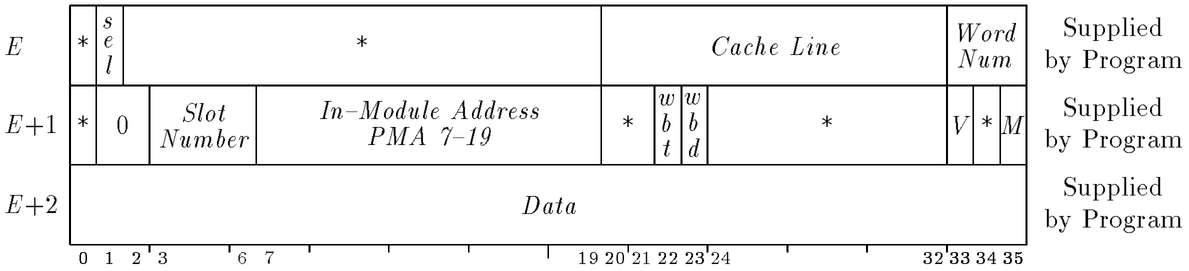
Data for DRDPTB

Page Translation Buffer Tag and Data Tripleword — DRDPTB



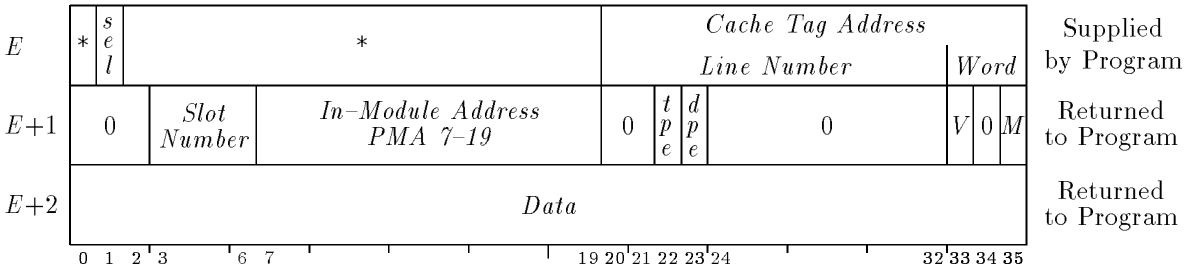
Data for DWRCSH

Cache Tag and Data Tripleword — DWRCSH

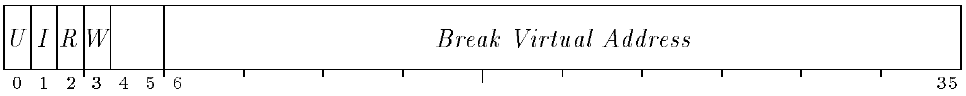


Data for DRDCSH

Cache Tag and Data Tripleword — DRDCSH

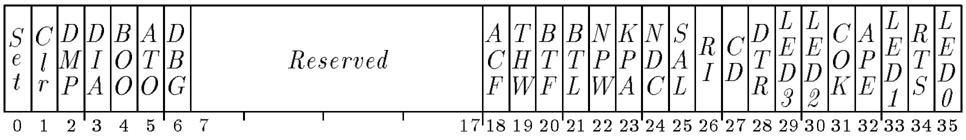


Data for WRADB



Data for WCTRLF

Control Flags for WCTRLF and RCTRLF



Data for SYSID

Data Format for SYSID

0–23	24–35
System Identification Number	*Reserved*
Backplane Serial Number	*Reserved*

Trap vector in UPT or EPT:

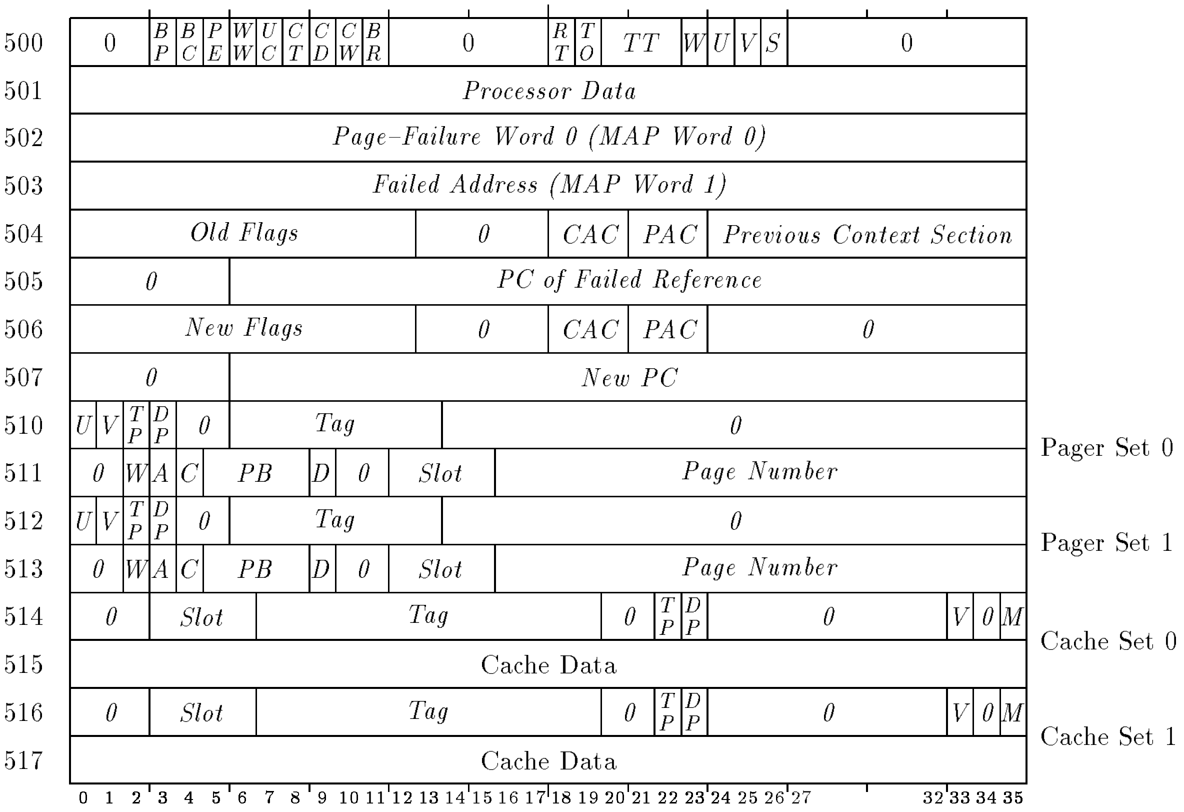
	0	1–5	6–12	13–17	18–20	21–23	24–35	
4x0	*T*	*Reserved*						`UP.BFL==:0`
4x1	*Reserved*							
4x2	*Reserved*							
4x3	*Reserved*							
4x4	*Old Flags*			0	*CAC*	*PAC*	*PCS*	`UP.OFL==:4`
4x5	0		*Old PC*					`UP.OPC==:5`
4x6	*New Flags*			0	*CAC*	*PAC*	0	`UP.NFL==:6`
4x7	0		*New PC*					`UP.NPC==:7`

Page–Failure data block in UPT (soft failures):

	0–5	6–12	13–17	18–20	21–23	24–35	
500	*Reserved*						`UP.PFB==:500`
501	*Reserved*						`UP.PFD==:501`
502	*Page–Failure Word 0 (MAP Word 0)*						`UP.PF0==:502`
503	*Failed Address (MAP Word 1)*						`UP.PF1==:503`
504	*Old Flags*		*0*	*CAC*	*PAC*	*Previous Context Section*	`UP.POF==:504`
505	*0*	*PC of Failed Reference*					`UP.POP==:505`
506	*New Flags*		*0*	*CAC*	*PAC*	*0*	`UP.PNF==:506`
507	*0*	*New PC*					`UP.PNP==:507`

Page–Failure Block at UPT 500

Page–Failure data block in EPT (hard failures):

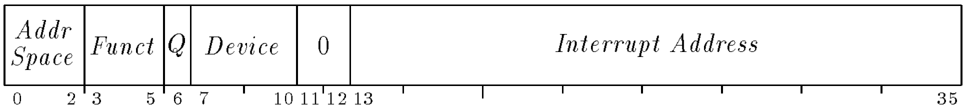


Page–Failure Block at EPT 500

D.2 KL10 Internal Device Bit Assignments

Interrupt function word:

KL10 Interrupt Function Word



Data for CONO PI, (PI=4)

Write Even Parity				Drop Prgm Req On Lvls	Clear PI Sys–tem	Make Prgm Req On	Turn On	Turn Off	Turn Off	Turn On	Select Levels for Bits 22,24,25,26						
Addr	Data	Dir				Selected Levels			PI System		1	2	3	4	5	6	7
18	19	20		22	23	24	25	26	27	28	29	30	31	32	33	34	35

Data for CONI PI,

	Program Requests on Levels							W E P A	W E P D	W E P R	Interrupt Holding on Levels							PI On	Levels On						
	1	2	3	4	5	6	7				1	2	3	4	5	6	7		1	2	3	4	5	6	7
0	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Format for WRFIL data, cache strategy table:

Refill Table Data		Refill Table Address	
18 20		27 33	35

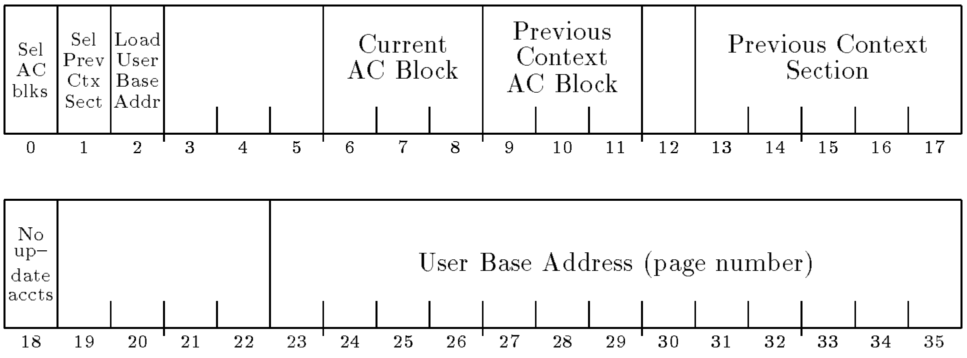
Format for APRID data:

Microcode Options				*Microcode Version*	*Hardware Options*						*Processor Serial Number*
T 2 0	*X A*	*X M*			*5 0 H*	*C c h*	*C h n*	*X A H*	*M O S*		
0	1	2	8	9 17	18	19	20	21	22	23	24 35

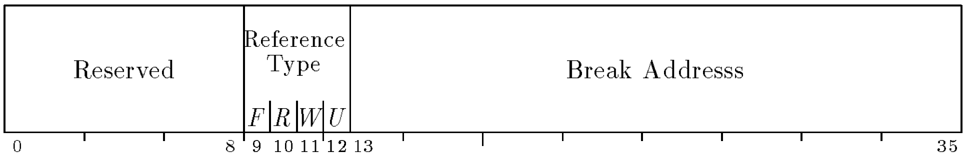
Format for CONO PAG, (PAG=10) data:

Cache Strategy			T–20 Pag	Enb Pag	Executive Base Address (page number)												
Look	Load																
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

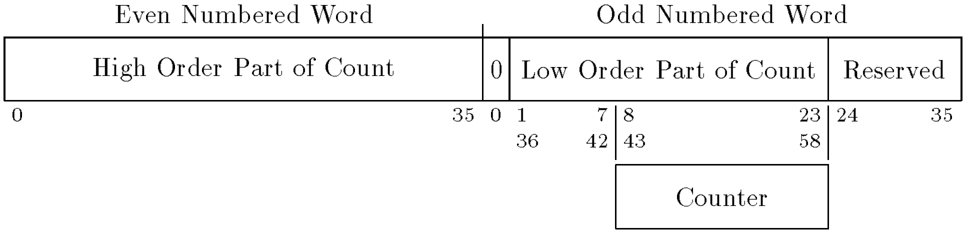
Format for DATAO PAG, data:



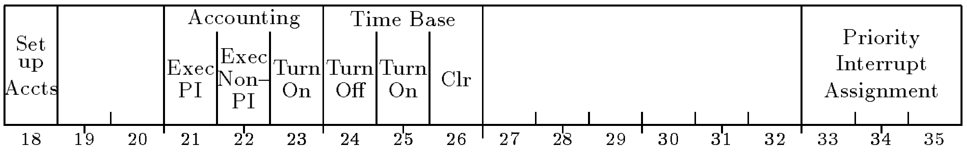
Format for DATAO APR, (APR=0) data, set address break:



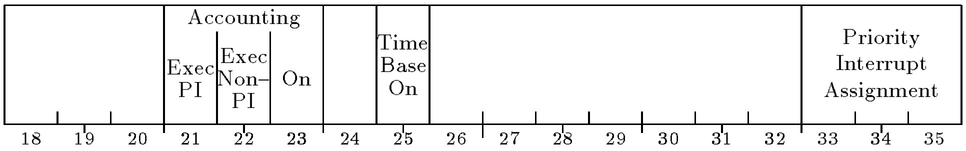
Double Word count format:



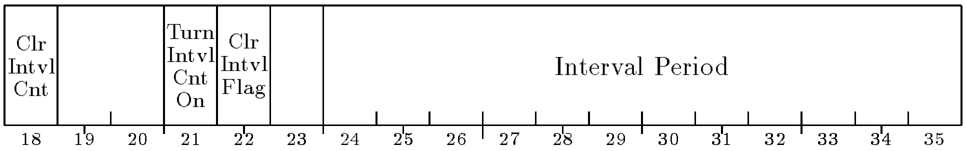
Format for CONO MTR, (MTR=24) data:



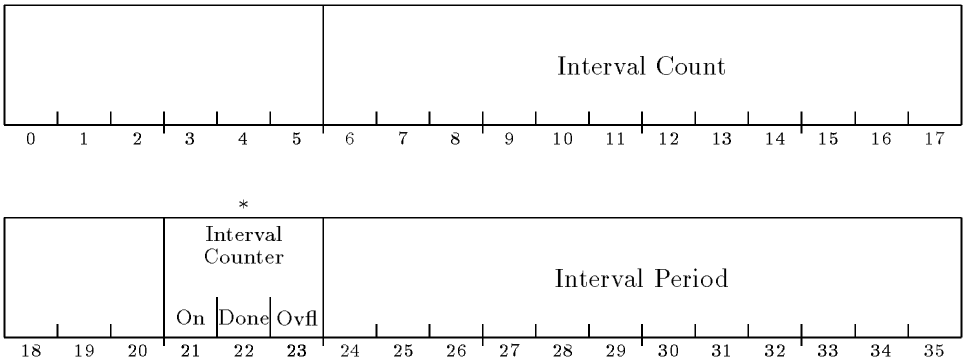
Format for CONI MTR, data:



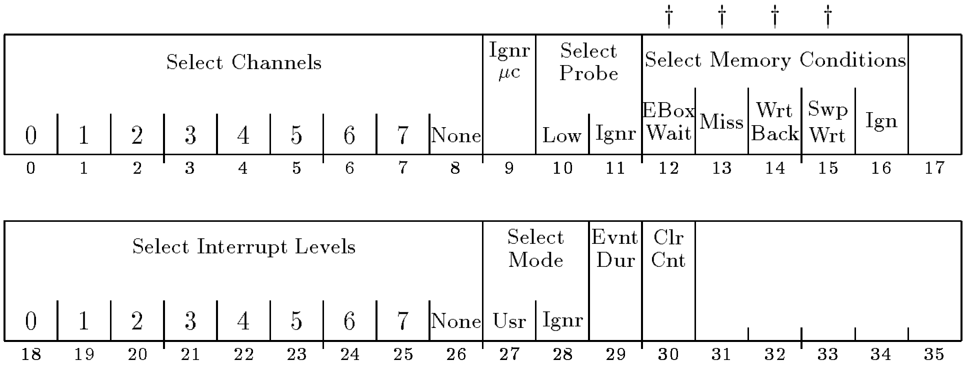
Format for CONO TIM, (TIM=20) data:



Format for CONI TIM, data:



Format for WRPAE data:



Format for CONO APR, data:

	Clr All I/O Dev	Enb	Dis	Clr	Set	Select Flags for Bits 20–23									Priority Interrupt Assignment		
		Selected Flags				SBus Err	No Mem	I/O Page Fail	MB Par	Cch Dir	Addr Par	Pwr Fail	Swp Done				
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Format for CONI APR, data:

						Flags Enabled to Interrupt											
						SBus Err	No Mem	I/O Page Fail	MB Par	Cch Dir	Addr Par	Pwr Fail	Swp Done				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

						*	*	*	*	*	*	*	*				
	Swp Busy					SBus Err	No Mem	I/O Page Fail	MB Par	Cch Dir	Addr Par	Pwr Fail	Swp Done	Int Req	Priority Interrupt Assignment		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Format for RDERA data:

Word Number		Reference Identification					Inde–termi–nate		0					High Order Address Bits			
		Swp	Chn	Data	Src	Wrt											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Physical Address of First Word of Transfer																	
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

D.2.1 TOPS–10 (KI or non–extended KL) Paging

EPT or UPT page map entry:

0	1	2	3	4	5–17	18	19	20	21	22	23–35
Data for Even Virtual Page						Data for Odd Virtual Page					
A	*P*	*W*	*S*	*C*	*Physical Page Address Bits 14–26*	*A*	*P*	*W*	*S*	*C*	*Physical Page Address Bits 14–26*

Page–failure word:

0	1	2	3	4	5	6	7	8	9–17	18–35
U	1 *Failure Type*							*V*		Virtual Address
	0	*A*	*W*	*S*	*T*	*P*	*C*			

Valid MAP data:

0	1	2	3	4	5	6	7	8	9–13	14–35
U	0	*A*	*W*	*S*	0	*P*	*C*	1	00	Physical Address

Data for invalid mapping:

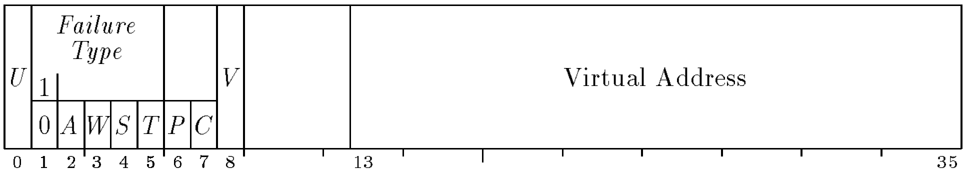
0	1–5	6	7	8	9–13	14–35
U	1 *Failure Type*	*P*	*C*	1	00	Physical Address

D.2.2 Extended KL Paging (TOPS–20 or TOPS–10 7.02 and later)

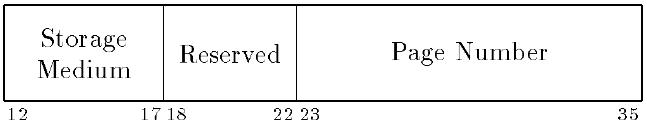
Page map entry:

A	*P*	*M*	*W*	*C*	Physical Page Address Bits 14–26

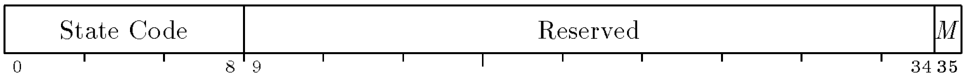
Page–failure word:



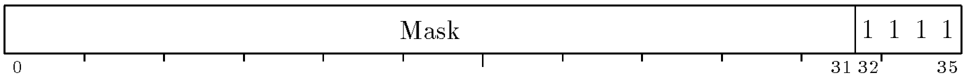
Page address:



CST entry:

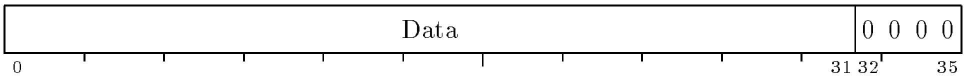


In AC 0, block 6:



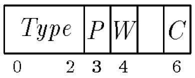
CST Mask Word

In AC 1, block 6:



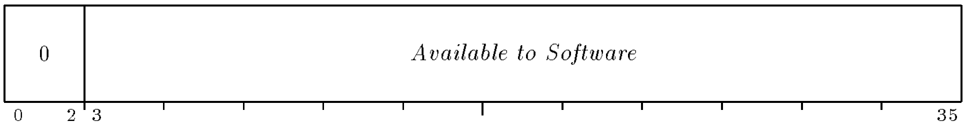
CST Data Word

Access pointer, leftmost 7 bits:

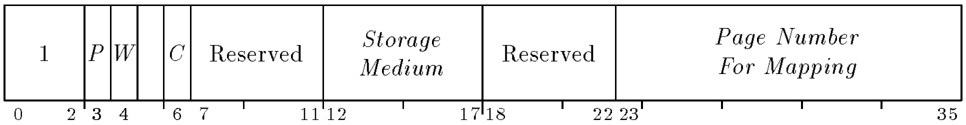


Page access pointers (section access pointers are similar):

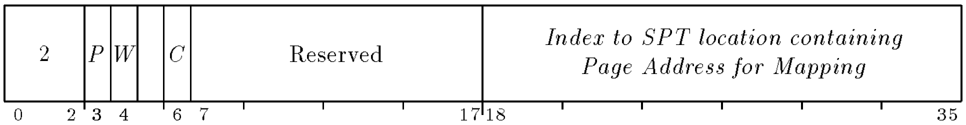
No Access



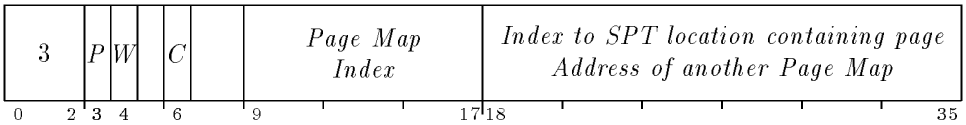
Immediate



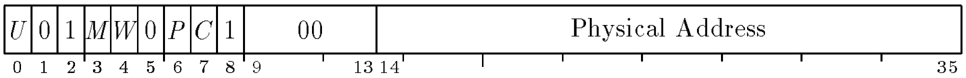
Shared



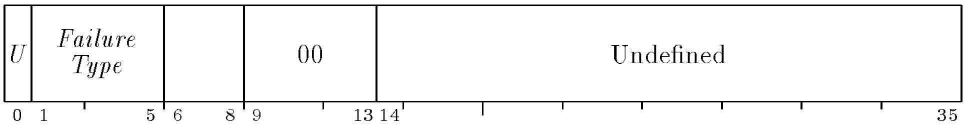
Indirect



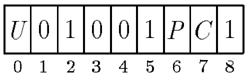
MAP instruction, true mapping:



MAP instruction, no valid mapping:



Page–failure trap, illegal write:



Page–failure trap, no valid mapping:

U	0	0	0	0	T	0	0	1
0	1	2	3	4	5	6	7	8

Appendix E

TDBOOT Command Summary

The contents of this appendix have been created mechanically from the online help texts contained within the TDBoot program. Hopefully, this approach encourages the accuracy and completeness of both this printed form and the online help texts. Because this is automatically generated, the typographical conventions used in this appendix vary slightly from those found elsewhere in this manual. This appendix reflects TDBoot Version 3.3(119)-1.

E.1 Macro–console commands

The commands that are processed by the macro–console are listed below. (Micro–console commands, all of which start with the period (".") character, are listed in the next section.)

The syntax "C[ONTINUE]" in a command or other keyword means that "C" is an explicit shorthand for "CONTINUE", even though it might otherwise be ambiguous.

`BOOT` Load an executable file into memory and start it. The optional argument can specify a path to a file on a TOPS–20 or TOPS–10 structure, a network node, a specific SCSI tape device nexus, an operating system tape specifier, or a sequential SCSI tape name.

`BOOT path`

The path argument can have one of the following formats:

`str:<dir.sub>name.typ.gen`	To specify a TOPS–20 disk file
`str:name.ext[p,pn,sub]`	To specify a TOPS–10 disk file
`node::'remote-file'`	To specify a network file (not implemented yet)
`# slot bus target [unit]`	To specify an explicit SCSI nexus
`MTAnnn:`	To specify a SCSI tape by its sequential nexus position
`channel, controller, unit`	Operating system tape specifier

If no argument is specified, the default boot string is used (see the "DEFINE BOOT" command). If no structure, node, "#", "MTAnnn:", or operating system tape specifier is specified, a TOPS–20 structure is located whose AUTOMATIC–STARTUP parameter has been set to this system's ID number. If none is found, the TOPS–10 structure with the name beginning with "DSK" and the lowest letter "x" (starting at "A") is used. If a TOPS–20 structure is used, the remaining file fields are defaulted from the following file specification:

```
<SYSTEM>MONITR.EXE.0
```

or if a TOPS–10 structure is used, the remaining file fields are defaulted from the following file specification:

```
SYSTEM.EXE[1,4]
```

For a network file, specify a node name terminated by two colons. Following the node name should be a remote file specification enclosed in either single or double quotes (either ' or "). The interpretation of the specification is left to the remote system.

*** Network files are not yet implemented.

For an explicit SCSI nexus, enter a pound sign ("#"), followed by the slot number of a mass–storage controller, a SCSI bus number, a SCSI device ID, and finally, an optional SCSI logical unit number. This method cannot be used to boot from a structure, since a structure could include more than one SCSI disk.

For a sequential SCSI tape, specify "MTAnnn:" where "nnn" is the sequential number of the desired tape. The tapes are numbered starting at zero as they are encountered while scanning the mass–storage controllers, SCSI buses, SCSI devices, and SCSI logical units, in that order. Thus MTA2: specifies the third SCSI tape encountered during the scan.

For an operating system tape specifier, specify three decimal numbers: channel, controller, and unit separated by commas. The decimal channel number is constructed by counting buses. Buses 0 thru 3 on the mass–storage controller in the lowest numbered slot number are channels 0 thru 3; buses 0 thru 3 on the mass–storage controller in the next lowest slot number are channels 4 thru 7; the next would be 8 thru 11, and so on. For SCSI devices with LUN zero, the controller number can be –1 and the unit number is the decimal target number. Otherwise, the controller number is the decimal target number, and the unit number is the LUN. For example, if there are two mass–storage controllers in the system, the tape with target number 8 and LUN 0 on bus 1 in slot 7 (the higher numbered mass–storage controller slot) would be accessed by:

5,–1,8

`BOOT switches`

Switches should be entered after all other arguments.

`/CACHE`

Enable caching of loaded pages. This switch implies /CST. The cachable bit is set in all CST entries. Do not use this option if the loaded program uses direct I/O between system memory and the XRH or XNI unless the program clears the appropriate cachable bits in the CST first. This switch must be specified with the first program loaded into memory.

`/CST`

When loading the program into memory, create a CST. This CST will be used when the program is run (by /START now, or START later). This switch must be specified with the first program loaded into memory.

`/CORE-DUMP`

Set the data mode to Core Dump format. In this mode, each 36–bit word corresponds to five 8–bit bytes. The first four 8–bit bytes contain bits 0–7, 8–15, 16–23, and 24–31, respectively. The last byte contains bits 32–35 right justified. This switch applies only to sequential–access devices (tapes).

`/DEBUG`

Set CF%DBG in control flags (WCTRLF/RCTRLF) when starting program. This can be used to tell a program (such as the monitor) that it should run in debug mode.

`/DDT`

Load TDBoot and its copy of DDT into memory and start DDT with symbols set up for the loaded program. This switch must be specified with the first program loaded into memory.

`/HIGH-DENSITY`

Set the data mode to High–Density format. In this mode, two 36–bit words correspond to nine 8–bit bytes. The first four 8–bit bytes contain bits 0–7, 8–15, 16–23, and 24–31 of the first word, respectively. The fifth byte contains bits 32–35 of the first word in the high–order 4 bits and bits 0–3 of the second word in the low–order 4 bits. The last four 8–bit bytes contain bits 4–11, 12–19, 20–27, and 28–35 of the second word, respectively. This switch applies only to sequential–access devices (tapes).

`/MERGE`

Merge the specified program with any programs already in memory.

`/NOLONG-TRANSFERS`

Do not use multi–page transfers. This switch applies only to direct–access devices (disks). By default, TDBoot will attempt to combine transfers involving sequetial pages on disk into large groups called “long transfers”. This switch forces each page to be processed as a separate transfer.

`/PROTECT`

Load this program into protected locations in high physical memory along with the paging data. This is useful for keeping a program out of the way of a program loaded into low memory.

`/REWIND`

Rewind the SCSI device before performing the requested operation.

`/START:address`

After loading the specified program, start it at the address supplied. If the address begins with a "+" is is interpreted as an entry–vector offset.

`/W`

"/W" is a synonym for "/REWIND", provided for backwards compatibility.

`CLEAR` Clear various system variables.

`CLEAR ADDRESS-BREAK`

Clear any address break set for the current program.

`CLEAR CACHE`

Clear CPU cache. Any modified CPU cache entries are written back to memory, and all cache entries are invalidated. If the /INVALIDATE switch is present, the CPU cache is merely invalidated without writing modified entries back to memory.

`CLEAR CONFIGURATION slot`

Clear the configuration for the specified slot. Until it is configured again the slot will be treated as if no device is present.

`CLEAR MEMORY`

Clear all or selected system memory modules so that they contain zeros and good parity. The STATIC and COMBINED memory tests are performed first to insure proper operation (see the TEST MEMORY command for more details on these tests).

`CLEAR MEMORY arguments`

The optional decimal slot number argument (0–15) indicates a specific memory to clear. No argument will clear all memories in the system. Switches may follow the slot number.

`CLEAR MEMORY switches`

Switches should be entered after all other arguments.

`/NOCACHING`

Disable use of the cache to accelerate the operation. This will force direct memory accesses even though they are slower. Use this if you think there are problems with the cache. Caching will automatically be disabled if the CACHE–TEST startup parameter is disabled, or if there were initialization errors. Since using the cache requires using the pager, caching is implicitly disabled if paging is disabled (PAGER–TEST is disabled, or /NOPAGING was specified).

`/NOPAGING`

Disable use of paging to accelerate the operation. This will force the use of PMOVEM even though it is slower. Use this if you think there are problems with the pager. Paging will automatically be disabled if the PAGER–TEST startup parameter is disabled, or if there were initialization errors. Since using the cache requires using the pager, caching is implicitly disabled if paging is disabled.

`CLEAR NVRAM`

Clear entire saved system configuration and reinitialize. Preserve the TDBoot parameters (those accessed via Enable and Disable).

`C[ONTINUE] [address]`

Resume running an interrupted program. The optional argument specifies the new PC at which to continue. With or without the argument, the flags remain unchanged.

`DAYTIME` Print the current date and time as stored in the hardware timebase. If the hardware timebase does not appear to be set properly, a message to that effect is printed instead of the date and time.

`DDT` Load TDBoot into memory, mapped in the same section it normally runs at, and enter DDT via an unsolicited breakpoint. You may type $P (escape, then "P") to proceed TDBoot running in system memory.

`DEFINE` Change certain static information which is saved in nonvolatile RAM. (The LIST command displays these values.)

`DEFINE AUTO-BOOT-DELAY`

Define the auto–boot delay for this processor. The argument is the decimal delay time in seconds (range 0–255). This is how long this processor waits prior to initiating an automatic boot after power–on. This value is saved in nonvolatile RAM.

`DEFINE AUXILIARY-PORT`

Define the state of the auxiliary terminal port. The state is saved in nonvolatile RAM and is used to set up the auxiliary port during initialization.

`DEFINE AUXILIARY-PORT OFF`

Turn off the auxiliary port.

`DEFINE AUXILIARY-PORT ON`

Turn on the auxiliary port.

`DEFINE BOOT-DEFAULTS arguments`

Define the default BOOT command arguments to be the remainder of the command line. See the BOOT command for the usage of this string. This value is saved in nonvolatile RAM.

`DEFINE CONFIGURATION {slot | *}`

Clear the saved system configuration for the specified slot, poll that slot, and save the new configuration. With "*" as the argument, clear, poll, and save the configuration for all system slots. Unlike other commands, the "*" is not the default and must be explicit. The new configuration information is saved in nonvolatile RAM.

`DEFINE DAYLIGHT-SAVINGS`

Define the default daylight savings handling. When the processor is intialized, this value is copied into dynamic storage (same as the "SET DAYLIGHT–SAVINGS" command). This value is saved in nonvolatile RAM.

`DEFINE DAYLIGHT-SAVINGS ALWAYS`

Always process date and time with daylight savings time in effect. This may be useful if the AUTOMATIC setting is not appropriate for your site.

`DEFINE DAYLIGHT-SAVINGS AUTOMATIC`

Automatically determine when to process date and time with daylight savings time in effect. The determination is based on the rules in effect in most of the USA at the time of this writing. If this does not seem to be appropriate, consider using the ALWAYS or NEVER settings.

`DEFINE DAYLIGHT-SAVINGS NEVER`

Never process date and time with daylight savings time in effect. This may be useful if the AUTOMATIC setting is not appropriate for your site.

`DEFINE DUMP-DEFAULTS arguments`

Define the default DUMP command arguments to be the remainder of the command line. See the DUMP command for the usage of this string. This value is saved in nonvolatile RAM.

`DEFINE IP-ADDRESS a.b.c.d`

Define the Internet Protocol Address for a network port. The first argument is the decimal slot number (1–15) of a network controller. The second argument is the port number (0–3) of a network connection. The optional third argument is an IP address in dot format (a.b.c.d), where each letter is replaced by a decimal number (0–255). If the third argument is no supplied, the IP address is cleared. These settings are not currently used, but are reserved for diagnostics.

`DEFINE SCSI-ID slot bus {id | OFFLINE}`

Define the SCSI target number for the mass–storage controller connected to a SCSI bus. The arguments specify the mass–storage controller, SCSI bus, and what ID to use (or to set the bus off line). These values are saved in nonvolatile RAM.

The "slot" is the decimal slot number (1–15) of a mass–storage controller. The "bus" is the SCSI bus number (0–3). The "id" is the decimal SCSI target number (0–15) which the mass–storage controller should use to identify itself on the specified bus. If you expect to communicate with any 8–bit SCSI devices, do not use SCSI target numbers above 7, as these devices which don't implement target numbers greater than 7. The OFFLINE keyword will set the specified bus off line. That is, it will not be used for any purpose until a valid SCSI target number has again been set up for it.

`DEFINE SLOT slot {OFF-LINE | ON-LINE}`

Define the specified slot as being either On–line or Off–line.

A slot that has been set off–line will not be scanned when looking for devices. The device in the slot (if any) will not be tested and it will not be used.

Setting a slot on–line allows it to be scanned. If a device is in the slot the device is testable and usable.

`DEFINE SYNC-DELAY seconds`

Define the syncrhonization delay for this processor. The argument is the decimal delay time in seconds (range 0–255). This value is used for multi–processor synchronization. This value is saved in nonvolatile RAM.

`DEFINE TIMEZONE hh:mm`

Define the local timezone as the specified number of decimal hours and minutes different from Greenwich. The value may be between –12:00 and 12:00 inclusive. Timezones west of Greenwich are negative, while timezones east of Greenwich are positive. The values –12:00 and 12:00 are the same time, but on opposite sides of the international date line, and thus differ by 1 day. For example, the United States eastern timezone would be –5:00 (standard time is 5 hours and 0 minutes earlier than Greenwich). When the processor is intialized, this value is copied into dynamic storage (same as the "SET TIMEZONE" command). Users should note that the sign of this value differs from the equivalent one used in TOPS–20 (in the 7–SETSPD program when read from the 7–CONFIG.CMD file). This value is saved in nonvolatile RAM.

`D[EPOSIT]`

Deposit into various system entities.

`D[EPOSIT] A-MEMORY address`

Store data in the microcode private memory (MemA). Supply an address (0–17777) followed by a 36–bit data word.

`D[EPOSIT] CACHE a1 a2 a3`

Store data in the CPU cache. See the description of the DWRCSH instruction for the format of the three arguments.

`D[EPOSIT] DEVICE-REGISTER slot address data`

Perform a device–control cycle (also known as a "write to I/O space") to write data into a device register. The first argument is the slot number of the desired device. The second argument is the in–module address of the register. The third argument is the data to be written.

`D[EPOSIT] FLAGS-AND-CONTEXT data`

Store new values for the processor flags, and current–context and previous–context context AC blocks, and previous–context section. The only argument contains the processor flags in bits 0–17, the current AC block number in bits 18–20, the previous–context AC block number in bits 21–23, and the previous–context section in bits 24–35. Care should be used when manipulating the processor flags and context, because this may adversely affect program operation.

`D[EPOSIT] MEMORY slot address data`

Using a physical memory address, write data into system memory. The first argument is the slot number of the desired device. The second argument is the in–module memory address. The third argument is the data to be written.

`D[EPOSIT] NVRAM address data`

Store data in the nonvolatile RAM. Supply an address (0–17777) followed by an 8–bit data byte.

`D[EPOSIT] PC address`

Set the current program counter. The argument is the address at which to continue program execution. Care should be used when manipulating the program counter, because this may adversely affect program operation.

`D[EPOSIT] REGISTER n data`

Write data into the current–context registers. The first argument is an octal register number (0–17). The second argument is the data to be written to the register. Following the second argument, you may type /BLOCK:n to specify the AC block number to examine. The default is the current AC block as specified in the flags and context word.

`/BLOCK:n`

Specify the AC block (0–7) to deposit in. The default is the current AC block as specified in the flags and context word.

`D[EPOSIT] SYSTEM-VIRTUAL address data`

Using a system virtual address, write data into system memory. The first argument is the system virtual address. The second argument is the data to be written. The /DETAIL switch prints the details of the virtual address resolution.

`/DETAIL`

Include a detailed printout of the virtual address resolution process while preforming the requested operation. This is useful for determining the source of a page fault or other error when referencing the specified virtual address.

`D[EPOSIT] TRANSLATION-BUFFER a1 a2 a3`

Store data in the CPU page–translation buffer (also known as the pager). See the description of the DWRPTB instruction for the format of the three arguments. Since the microcode clears the pager upon exit from the macro–console, this command is of limited utility. It can be used more effectively from a copy of TDBoot running in system memory.

`D[EPOSIT] USER-VIRTUAL address data`

Using a user virtual address, write data into system memory. The first argument is the user virtual address. The second argument is the data to be written. The /DETAIL switch prints the details of the virtual address resolution.

`/DETAIL`

Include a detailed printout of the virtual address resolution process while preforming the requested operation. This is useful for determining the source of a page fault or other error when referencing the specified virtual address.

`DIRECTORY [path]`

List all selected files in a TOPS–20 or TOPS–10 directory. The optional argument can specify a path to the directory, as well as which files to list. There are also switches available to select what information will be printed for each file. This command is valid only for disk devices. The path for a TOPS–20 disk has the following format:

```
str:<dir.sub>name.typ.gen
```

Any unspecified file fields are default from the following file specification:

```
<SYSTEM>MONITR.EXE.*
```

The path for a TOPS–10 structure has the following format:

```
str:name.ext[p,pn,sub]
```

Any unspecified file fields are default from the following file specification:

```
SYSTEM.EXE[1,4]
```

`/CREATION`

Print date and time of creation for each file.

`/LENGTH`

Print length for each file. For TOPS–20, this is the number of bytes and the byte size; for TOPS–10, this is the number of words.

`/L[ONG]`

This is a shorthand for /SIZE, /LENGTH, and /WRITE.

`/READ`

Print date and time of last read for each file.

`/SIZE`

Print size for each file. For TOPS–20, the size is in pages; for TOPS–10, the size is in blocks.

`/WRITE`

Print date and time of last write for each file.

`DISABLE parameter-list`

Disable various system startup parameters. Provide a list of parameter names separated by commas.

The parameter names and their meanings are listed with the ENABLE command.

DUMP Copy the entire physical system memory image (and some of the processor state) to the file specified by the optional path argument. The optional argument can specify a path to a file on a TOPS–20 or TOPS–10 structure, a network node, a specific SCSI tape device nexus, a sequential SCSI tape name, or an operating system tape specifier. The default "path" is specified by the "DEFINE DUMP" command. If the target device is a disk structure, the given file must exist and it must be be capacious enough to hold all of system memory.

DUMP path

The syntax of the path argument to the DUMP command is the same as for the BOOT command. The default DUMP string is specified via the DEFINE DUMP command.

If a structure is used, the remaining file fields are defaulted from the following file specification for TOPS–20:

```
<SYSTEM>DUMP.EXE.0
```

or the following file specification for TOPS–10:

```
CRASH.EXE[1,4]
```

DUMP switches

Switches should be entered after all other arguments.

/ALL-MEMORY

Dump or save all of physical memory, if possible.

/CORE-DUMP

Set the data mode to Core Dump format. In this mode, each 36–bit word corresponds to five 8–bit bytes. The first four 8–bit bytes contain bits 0–7, 8–15, 16–23, and 24–31, respectively. The last byte contains bits 32–35 right justified. This switch applies only to sequential–access devices (tapes).

/HIGH-DENSITY

Set the data mode to High–Density format. In this mode, two 36–bit words correspond to nine 8–bit bytes. The first four 8–bit bytes contain bits 0–7, 8–15, 16–23, and 24–31 of the first word, respectively. The fifth byte contains bits 32–35 of the first word in the high–order 4 bits and bits 0–3 of the second word in the low–order 4 bits. The last four 8–bit bytes contain bits 4–11, 12–19, 20–27, and 28–35 of the second word, respectively. This switch applies only to sequential–access devices (tapes).

/MAPPED-MEMORY

Dump or save only exec mapped memory.

/NOLONG-TRANSFERS

Do not use multi–page transfers. This switch applies only to direct–access devices (disks). By default, TDBoot will attempt to combine transfers involving sequetial pages on disk into large groups called "long transfers". This switch forces each page to be processed as a separate transfer.

`/REWIND`

Rewind the SCSI device before performing the requested operation.

`/W`

"/W" is a synonym for "/REWIND", provided for backwards compatibility.

`ENABLE` Enable various system startup parameters. Provide a list of parameter names separated by commas.

`ENABLE AUTO-BOOT`

The AUTO–BOOT startup parameter. If this paramenter is enabled at startup, after a short delay (see DEFINE AUTO–BOOT–DELAY), the system will attempt to boot automatically using the defaults set up via the DEFINE BOOT command.

`ENABLE BUS-POLL`

The BUS–POLL startup parameter. If this parameter is enabled at startup, the system will poll the bus to determine the hardware configuration.

`ENABLE CACHE-TEST`

The CACHE–TEST startup parameter. If this parameter is enabled at startup, the system will perform a diagnostic check of the cache. This parameter must be enabled to automatically utilize the cache during some memory tests (see TEST MEMORY).

`ENABLE CLEAR-MEMORY`

The CLEAR–MEMORY startup parameter. If this parameter is enabled at startup, the system will test memory, which leaves all of memory with good parity. If TEST–MEMORY is disabled and CLEAR–MEMORY is enabled, all memory is cleared at startup. This prevents any latent parity errors from affecting system behavior. Prior to clearing memory, the static memory tests are performed (see TEST MEMORY/STATIC). If the PAGER–TEST and CACHE–TEST parameters are enabled, then the combined tests are performed (see TEST MEMORY/COMBINED).

`ENABLE CONFIGURE-MEMORY`

The CONFIGURE–MEMORY startup parameter. If this paremter is enabled at startup, the system will configure all available memory. Prior to configuring memory, each memory may also be tested (see TEST–MEMORY) or cleared (see CLEAR–MEMORY).

`ENABLE MASS-STORAGE-CONFIGURATION`

The MASS–STORAGE–CONFIGURATION startup parameter. If this parameter is enabled at startup, the system will report the mass–storage controller device configuration and start the disks.

`ENABLE PAGER-TEST`

The PAGER–TEST startup parameter. If this parameter is enabled at startup, the system will perform a diagnostic check of the pager. This parameter must be enabled to automatically utilize the pager during some memory tests (see TEST MEMORY), and when clearing memory (see CLEAR MEMORY). Since the pager is needed to use the cache during the TEST MEMORY and CLEAR MEMORY commands, the use of the cache is implicitly disabled when this parameter is disabled.

`ENABLE TEST-MEMORY`

The TEST–MEMORY startup parameter. If this parameter is enabled at startup, the system will perform a diagnostic check of the system memory. Prior to other testing, the static memory tests are performed (see TEST MEMORY/STATIC). If the PAGER–TEST and CACHE–TEST parameters are enabled, then the combined tests are performed (see TEST MEMORY/COMBINED).

`ENABLE *`

This is a simple abbreviation used to ENABLE or DISABLE all startup parameters.

`E[XAMINE]`

Examine various system entities.

`E[XAMINE] A-MEMORY address`

Print the contents of the specified location in the microcode private memory (MemA). The argument is the octal address (0–17777).

`E[XAMINE] CACHE set-and-line-index`

Print the contents of the specified CPU cache line. See the description of the DRDCSH instruction for the format of the argument.

`E[XAMINE] DEVICE-REGISTER slot address`

Perform a device status request cycle (also known as a "read from I/O space") to read data from a device register. The next argument is the in–module address of the register.

`E[XAMINE] FLAGS-AND-CONTEXT`

Print current program flags and context information.

`E[XAMINE] INTERRUPT`

Print current interrupt system status.

`E[XAMINE] MASS-STORAGE slot`

Print the contents of a location in one of the memory spaces in the specified mass–storage controller. The first argument is the slot number of the desired device. The second argument is a register number (0–17), or a keyword describing the name of the space to be examined.

`E[XAMINE] MASS-STORAGE slot register`

Print the result of reading the specified mass–storage controller register. This argument is the octal number of the desired register. The optional next argument specifies an octal address to be supplied in the register address bits as appropriate for the register.

`E[XAMINE] MASS-STORAGE slot ALU`

Print the contents of a location in the ALU of a mass–storage controller. The next argument is the location to read in the mass–storage controller ALU.

`E[XAMINE] MASS-STORAGE slot COMMUNICATIONS-REGION`

Print the contents of the mass–storage controller communications region register. The optional argument specifies an offset (0–7) in the communications region whose contents is printed as well.

`E[XAMINE] MASS-STORAGE slot DRAM`

Print the contents of the specified DRAM address of the specified mass–storage controller. The next argument is the address of the entry to read in the mass–storage controller DRAM.

`E[XAMINE] MASS-STORAGE slot ERROR-ADDRESS`

Print the contents of the error–address register of the specified mass–storage controller.

`E[XAMINE] MASS-STORAGE slot HASH-TABLE`

Print the contents of the specified hash–table entry (two words) in the DRAM of the specified mass–storage controller. The next argument is the hash–table entry to read in the specified mass–storage controller DRAM. The actual words read from DRAM are the address times two and the address times two plus one.

`E[XAMINE] MASS-STORAGE slot MICROCODE-VERSION`

Print the contents of the microcode version register of the specified mass–storage controller.

`E[XAMINE] MASS-STORAGE slot SERIAL-NUMBER`

Print the contents of the serial number register of the specified mass–storage controller.

`E[XAMINE] MASS-STORAGE slot SRAM`

Print the contents of a location in the SRAM of the specified mass–storage controller. The next argument is the location to read in the SRAM.

`E[XAMINE] MASS-STORAGE slot STATUS`

Print the contents of the status register of the specified mass–storage controller.

`E[XAMINE] M[EMORY] slot address`

Print the contents of the specified physical memory location. The next argument is the in–module memory address.

`E[XAMINE] NETWORK slot`

Print the interpreted contents of a register in the specified Network Contoller. The first argument is the slot number of the desired network controller. The second argument is the name of the register to be examined.

`E[XAMINE] NETWORK slot BUS-STATUS`

Prints the contents of the bus status register of the specified network controller.

`E[XAMINE] NETWORK slot STATUS`

Print the contents of the status register of the specified network controller.

`E[XAMINE] NEXT [n]`

Print the logically next value of the preceding EXAMINE or DEPOSIT command, whichever was most recent. The optional argument is a decimal repeat count. This is the default if no arguments are specified after the EXAMINE command.

`E[XAMINE] NVRAM address`

Print the contents of the specified byte in the nonvolatile RAM. The argument is an octal address (0–17777).

`E[XAMINE] PC`

Print current program PC.

`E[XAMINE] PREVIOUS [n]`

Print the logically previous value from the most recent EXAMINE or DEPOSIT command. The optional argument is a decimal repeat count.

`E[XAMINE] REGISTER {n | *}`

Examine the specified register (octal argument) or all registers ("*" argument). Following the argument, you may type /BLOCK:n to specify the AC block number to examine. The default is the current AC block as specified in the flags and flags and context word.

`/BLOCK:n`

Specify the AC block (0–7) to examine. The default is the current AC block as specified in the flags and context word.

`E[XAMINE] SYSTEM-VIRTUAL address`

Print the contents of the specified location in system memory. The argument is a system virtual address. The /DETAIL switch prints the details of the virtual address resolution.

`/DETAIL`

Include a detailed printout of the virtual address resolution process while preforming the requested operation. This is useful for determining the source of a page fault or other error when referencing the specified virtual address.

`E[XAMINE] TRANSLATION-BUFFER set-and-line-index`

Print the contents of the specified CPU page–translation buffer entry. See the description of the DRDPTB instruction for the format of the argument. Since the microcode clears the pager upon exit from the macro–console, this command will find only valid data in the pager on the first macro–console command following the placing of that data in the pager. It can be used more effectively from a copy of TDBoot running in system memory.

`E[XAMINE] USER-VIRTUAL address`

Print the contents of the specified location in system memory. The argument is a user virtual address. The /DETAIL switch prints the details of the virtual address resolution.

`/DETAIL`

Include a detailed printout of the virtual address resolution process while preforming the requested operation. This is useful for determining the source of a page fault or other error when referencing the specified virtual address.

`E[XAMINE] *`

Examine the macro PC, flags, context, PI, and current register block.

`EXIT` Resume running an interrupted program. If debugging TDBoot, this command will HALT the program.

`FORMAT` Format a disk for the operating system.

This command performs the SCSI Mode Select and Format Unit operations needed to set the disk for 512–word (2304 byte, long) physical records or 128–word (576 byte, short) physical records. After "hard" formatting the disk, this command goes on to perform other bookkeeping operations ("soft formatting") that are needed to make the disk ready for use by the operating system.

Restrictions: this command takes a long time. Disk types known to the program can be formatted; unknown disks may be formatted if they comply with any version of "standard behavior".

`FORMAT arguments`

The arguments specify which disk to format as:

slot bus target–id [unit]

Where "slot" is the decimal slot number (1–15) of the mass–storage controller, "bus" is the SCSI bus number (0–3), "id" is the decimal SCSI target number (0–15) of the desired disk drive, and "unit" is the optional SCSI logical unit number (0–7) within the target.

`FORMAT switches`

Switches should be entered after all other arguments.

`/FORCE-SIMPLE`

Soft format using only a simple 1 or 2 entry zone table. Normally, TDBoot will determine the proper number and sizes or zones for the disk. If you are told that there are too many zones, you might consider performing the soft format again using this switch.

Note: The zone table generated using this switch will most likely not match the actual geometry of the disk. This may cause operating systems or other software to incorrectly optimize transfers to this device. However, the errors should be relatively small.

`/HARD`

Perform a hard format followed by a soft format.

`/LONG`

Format using only long (2304 byte/512 word) blocks.

`/SHORT`

Format using only short (576 byte/128 word) blocks.

`/SOFT`

Perform only a soft format. The unit must already have an acceptable hard format.

`/TYPE:keyword`

Select a specific format algorithm. The following algorithms are available:

`GENERIC`	Default format algorithm
`HP`	Algorithm applied to Hewlett–Packard drives
`SEAGATE`	Algorithm applied to Seagate drives

`GET [path]`

GET is an alternate name for the LOAD command.

`H[ALT]` Stop the currently running program. Certain commands are allowed only when the program is halted. The program may be resumed using the CONTINUE command.

`HELP` The HELP command with no argument can be used to get a list of commands and brief descriptions. To get a more detailed description of a particular command, give the command name as the argument to the HELP command (e.g., “HELP HALT”). This may be extended for commands with multiple keywords (e.g., “HELP EXAMINE PC”). Commands with switches will provide help information by typing the command name followed by “switches” and one of the switches (e.g., “HELP BOOT switches /START:”).

`INTERRUPT`

Set the interrupt request flag (AP%INT) in the processor flags accessible by RDAPR and continue the program. A program can use this to receive interrupt requests by enabling an interrupt when this bit is set or by sampling the bit periodically.

`LIST` Display various static system parameters stored in nonvolatile RAM. (These are the parameters that are changed by the DEFINE, ENABLE, and DISABLE commands.)

`LIST AUTO-BOOT-DELAY`

Print the auto–boot delay in seconds for this processor stored in nonvolatile RAM.

`LIST AUXILIARY-PORT`

Print the state of the auxiliary terminal port stored in nonvolatile RAM.

`LIST BOOT-DEFAULTS`

Print the default BOOT command argument stored in nonvolatile RAM.

`LIST CONFIGURATION`

Print the static bus configuration stored in nonvolatile RAM.

`LIST DAYLIGHT-SAVINGS`

Print the static DAYLIGHT–SAVINGS setting stored in nonvolatile RAM.

`LIST DUMP-DEFAULTS`

Print the default DUMP command argument stored in nonvolatile RAM.

`LIST IP-ADDRESSES`

Print the Internet Protocol Address for each network connection in the system stored in nonvolatile RAM.

`LIST PARAMETERS`

Print the saved parameter settings stored in nonvolatile RAM.

`LIST SCSI-IDS`

Print the static SCSI target ID number for each SCSI bus on each mass–storage controller in the system stored in nonvolatile RAM.

`LIST SYNC-DELAY`

Print the synchronization delay in seconds for this processor stored in nonvolatile RAM.

`LIST TIMEZONE`

Print the static local TIMEZONE value stored in nonvolatile RAM.

`LIST *`

Print the state of all information stored in nonvolatile RAM. This is the default if no arguments are specified after the LIST command.

`L[OAD]` Load an executable file into memory, but do not start it. The optional argument can specify a path to a file on a TOPS–20 or TOPS–10 structure, a network node, a specific SCSI tape device nexus, a sequential SCSI tape name, or an operating system tape specifier. GET is an alternate name for LOAD.

`L[OAD] path`

The path argument for LOAD has the same format as described for the path argument to the BOOT command.

`L[OAD] switches`

The switches for LOAD are the same as those described in the BOOT command. However, the /DEBUG and /START switches do not apply to the LOAD command.

`REINITIALIZE`

Clear the dynamic configuration data and rescan the system configuration (if enabled). This will also cause the cache and pager tests to be run, as well as testing or clearing memory, if enabled.

`RESET` Place the specified components and/or devices in their initial state. A slot number or a keyword may be entered.

`RESET slot`

The "slot" argument is a decimal slot number (0–15) indicating the single slot to reset. Slot 0 refers to this CPU.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`/SHUTDOWN`

Shut down the specified mass–storage controllers. The mass–storage controllers will attempt to complete all outstanding requests and flush their caches back to disk.

`RESET BUS`

Reset all bus devices except this CPU.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`/SHUTDOWN`

Shut down the specified mass–storage controllers. The mass–storage controllers will attempt to complete all outstanding requests and flush their caches back to disk.

`RESET CPU`

Reset only CPUs.

`RESET CPU arguments`

The optional decimal slot number argument (0–15) indicates a specific CPU to be reset (0 is a synonym for this CPU). A missing argument or "*" will reset all CPUs in the system.

`RESET CPU switches`

Switches should be entered after all other arguments.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`RESET MASS-STORAGE`

Reset only mass–storage controllers.

`RESET MASS-STORAGE arguments`

The following arguments may be used to reset a subset of the mass–storage devices:

slot bus target–id

The “slot” argument specifies a mass–storage controller, the “bus” argument specifies a SCSI bus, and the “id” argument specifies a SCSI target id number. A missing argument or “*” will select all slots, buses, or ids corresponding to the argument.

`RESET MASS-STORAGE switches`

Switches should be entered after all other arguments.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`/SHUTDOWN`

Shut down the specified mass–storage controllers. The mass–storage controllers will attempt to complete all outstanding requests and flush their caches back to disk.

`RESET MEMORY`

Reset only memories.

`RESET MEMORY arguments`

The optional decimal slot number argument (0–15) indicates a specific memory to be reset. A missing argument or “*” will reset all memories in the system.

`RESET MEMORY switches`

Switches should be entered after all other arguments.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`RESET NETWORK`

Reset only network controllers.

`RESET NETWORK arguments`

The following arguments may be used to reset a subset of the network ports:

slot port

The "slot" argument specifies a network controller, and the "port" argument specifies a network port. A missing argument or "*" will select all slots or ports corresponding to the argument.

`RESET NETWORK switches`

Switches should be entered after all other arguments.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`RESET *`

Reset all system devices, including this CPU. This is the default if no arguments are specified after the RESET command.

`/HARD`

Reset the specified device(s) in a hard manner. A hard reset may not wait for data transfers to be completed, and thus could discard data destined for the disk or network.

`/SHUTDOWN`

Shut down the specified mass–storage controllers. The mass–storage controllers will attempt to complete all outstanding requests and flush their caches back to disk.

`REWIND scsi-device-specifier`

Rewind the specified SCSI device (usually a tape). The device is specified using one of the following formats:

`# slot bus target [unit]`	To specify an explicit SCSI nexus
`MTAnnn:`	To specify a SCSI tape by its sequential nexus position
`channel, controller, unit`	Operating system tape specifier

`R[UN] [path]`

RUN is an alternate name for the BOOT command.

`SAVE` Save the current program as an executable image to the file specified by the optional path argument. The optional argument can specify a path to a file on a TOPS–20 or TOPS–10 structure, a network node, a specific SCSI tape device nexus, a sequential SCSI tape name, or an operating system tape specifier. There is no default path. If the target device is a disk structure, the given file must exist and it must be be capacious enough to hold the program. *** Not implemented yet ***

`SAVE path`

The syntax of the path argument to the SAVE command is the same as for the BOOT command. There is no default path string for the SAVE command; a path must be supplied.

If a structure is used, the remaining file fields are defaulted (note: no default filename). For TOPS–20 and TOPS–10, the default file type (extension) is ".EXE". For TOPS–20, the default generation is 0.

`SAVE switches`

The switches for the SAVE command are the same as those described for the DUMP command.

`SCAN` Check various memory elements for data patterns and/or errors.

`SCAN CACHE`

Check the CPU cache for data patterns and/or errors.

`SCAN CACHE DATA value [mask]`

Check the CPU cache for data patterns. The first argument is the search data to be matched against any cache data entry. The optional second argument is the mask to be anded with the cache data entry before comparing it to the search data. If the second argument is not specified, it defaults to all ones.

`SCAN CACHE ERRORS`

Check the CPU cache memory for errors. Each cache line, set, and word is examined and any cache data or tag parity, or both cache sets match errors are printed. This is the default if no arguments are specified after the SCAN CACHE command.

`/REPAIR`

When an error is encountered during the scan, an attempt is made to repair it. In the case or a parity error, this means rewriting the tag or data with good parity. In the case of both sets matched, one of the sets will be cleared if it has not been modified.

`SCAN CACHE TAG value [mask]`

Check the CPU cache memory for tag patterns. The first argument is the search tag to be matched against any cache tag entry. The optional second argument is the mask to be anded with the cache tag entry before comparing it to the search tag. See the description of the DRDCSH instruction E+1 word for the format of the tag. If the second argument is not specified, it defaults to all ones.

`SCAN MEMORY`

Check system memory for data patterns and/or errors.

`SCAN MEMORY DATA value [mask]`

Check system memory for data patterns. The first argument is the search data to be matched against any pager data entry. The optional second argument is the mask to be anded with the pager data entry before comparing it to the search data. If the second argument is not specified, it defaults to all ones. Only configured memories are scanned.

`SCAN MEMORY ERRORS`

Check system memory for parity errors. All memory modules which are configured are read and any parity errors are printed. This is the default if no arguments are specified after the SCAN MEMORY command.

`/REPAIR`

When an error is encountered during the scan, an attempt is made to repair it. In the case or a parity error, this means rewriting the data with good parity.

`SCAN PAGER`

Check the CPU pager for data patterns and/or errors.

`SCAN PAGER DATA value [mask]`

Check the CPU pager for data patterns. The first argument is the search data to be matched against any pager data entry. The optional second argument is the mask to be anded with the pager data entry before comparing it to the search data. If the second argument is not specified, it defaults to all ones.

`SCAN PAGER ERRORS`

Check the CPU pager for errors. Each pager line and set is examined and any pager data or tag parity, or both pager sets match errors are printed. This is the default if not arguments are specified after the SCAN PAGER command.

`/REPAIR`

When an error is encountered during the scan, an attempt is made to repair it. In the case or a parity error, this means rewriting the tag or data with good parity. In the case of both sets matched, one of the sets will be cleared.

`SCAN PAGER TAG value [mask]`

Check the CPU pager for tag patterns. The first argument is the search tag to be matched against any pager tag entry. The optinal second argument is the mask to be anded with the pager tag entry before comparing it to the search tag. If the second argument is not specified, it defaults to all ones. See the description of the DRDPTB instruction E+1 word for the format of the tag.

`SET` Change certain dynamic information saved in MemA (in a manner less permanent than the DEFINE command). (See also the SHOW command.)

`SET AUXILIARY-PORT {ON | OFF}`

Set the state of the auxiliary terminal port. The new state is applied immediately to the auxiliary port. The keywords and their meanings are listed with the DEFINE AUXILIARY–PORT command.

`SET ADDRESS-BREAK`

Set an address break after loading a program into memory. If you set an address break and then load the first program, the implicit program reset that is done will clear the break.

`SET ADDRESS-BREAK arguments`

The single argument is a 30–bit virtual address on which to break.

`SET ADDRESS-BREAK switches`

Switches should be entered after all other arguments.

`/EXECUTE`

Select an address break when an instruction is fetched from the specified address.

`/READ`

Select an address break when data is read from the specified address.

`/USER`

Select an address break when the specified address is accessed in user mode.

`/WRITE`

Select an address break when data is written to the specified address.

`SET CONFIGURATION`

Explicitly configure various parts of the machine.

`SET CONFIGURATION slot`

The “slot” argument is a decimal slot number (0–15) indicating the single slot to configure. That slot will be polled and the results saved in the dynamic configuration database for that slot. Any discrepancies between the saved configuration and the current configuration for that slot will be printed.

`SET CONFIGURATION BUS`

Explicitly poll each slot in the machine and initialize the dynamic configuration. It will also print any discrepancies between the saved configuration and the current configuration for each slot.

`SET CONFIGURATION MASS-STORAGE`

Set up the dynamic configuration of all mass–storage controllers in the system. Start all disks and report device configuration.

`SET CONFIGURATION MEMORY`

Configure memory. Each memory is tested for proper operation and configured for use.

`/CLEAR`

Enable clearing memory after configuration and testing. The default setting for this switch is the same as the previous "CONFIGURE MEMORY" command. At powerup, it is initialized to the same setting as that of the CLEAR–MEMORY startup parameter.

`/FORCE`

Reconsider any memory units that have been put off line. Such a unit will have to pass some tests before it is accepted. This setting is remembered as part of the dynamic configuration and will be used until the processor is reset or TDBoot is REINITIALIZEd or the /NOFORCE switch is used.

`/FORWARD`

Configure the memory in the forward direction; that is, with the lowest physical slot numbers having the lowest addresses. This setting is remembered as part of the dynamic configuration and will be used until the processor is reset or TDBoot is REINITIALIZEd.

`/NOCACHING`

Disable use of the cache to accelerate the operation. This will force direct memory accesses even though they are slower. Use this if you think there are problems with the cache. Caching will automatically be disabled if the CACHE–TEST startup parameter is disabled, or if there were initialization errors. Since using the cache requires using the pager, caching is implicitly disabled if paging is disabled (PAGER–TEST is disabled, or /NOPAGING was specified).

`/NOCLEAR`

Disable clearing memory after configuration and testing. The default setting for this switch is the same as the previous "CONFIGURE MEMORY" command. At powerup, it is initialized to the same setting as that of the CLEAR–MEMORY startup parameter.

`/NOFORCE`

Do not reconsider any memory units that have been taken off line. This setting is remembered as part of the dynamic configuration and will be used until the processor is reset or TDBoot is REINITIALIZEd.

`/NOPAGING`

Disable use of paging to accelerate the operation. This will force the use of PMOVEM even though it is slower. Use this if you think there are problems with the pager. Paging will automatically be disabled if the PAGER–TEST startup parameter is disabled, or if there were initialization errors. Since using the cache requires using the pager, caching is implicitly disabled if paging is disabled.

`/NOTEST`

Disable memory testing after configuration. The default setting for this switch is the same as the previous "CONFIGURE MEMORY" command. At powerup, it is initialized to the same setting as that of the TEST–MEMORY startup parameter.

`/REVERSE`

Configure the memory in the highest physical slot number to have the lowest addresses. This setting is remembered as part of the dynamic configuration and will be used until the processor is reset or TDBoot is REINITIALIZEd.

`/TEST`

Enable memory testing after configuration. The default setting for this switch is the same as the previous "CONFIGURE MEMORY" command. At powerup, it is initialized to the same setting as that of the TEST–MEMORY startup parameter.

`SET CONFIGURATION NETWORK`

Set up the dynamic configuration of all network connections in the system. There is currently no dynamic conguration to setup for the network controllers. This command is reserved for future expansion.

`SET CONFIGURATION *`

Perform the default system configuration which includes the BUS, MEMORY, and MASS–STORAGE. This is the default if no arguments are specified after the SET CONFIGURATION command.

`SET DAYLIGHT-SAVINGS`

Set default daylight savings handling.

`SET DAYLIGHT-SAVINGS ALWAYS`

Always process date and time with daylight savings time in effect. This may be useful if the AUTOMATIC setting is not appropriate for your site.

`SET DAYLIGHT-SAVINGS AUTOMATIC`

Automatically determine when to process date and time with daylight savings time in effect. The determination is based on the rules in effect in most of the USA at the time of this writing. If this does not seem to be appropriate, consider using the ALWAYS or NEVER settings.

`SET DAYLIGHT-SAVINGS NEVER`

Never process date and time with daylight savings time in effect. This may be useful if the AUTOMATIC setting is not appropriate for your site.

`SET SCSI-ID slot bus {id | OFFLINE}`

Set the initiator target ID for a connected SCSI bus. The arguments specify the mass–storage controller, SCSI Bus, and what ID to use (or to set the bus off line).

This command changes only the dynamic configuration of the SCSI interface. The "SET CONFIGURATION BUS" and "SET CONFIGURATION *" commands will set the SCSI bus states from their static values saved via DEFINE SCSI–ID.The "slot" is the decimal slot number (1–15) of a mass–storage controller. The "bus" is the SCSI bus number (0–3). The "id" is the decimal SCSI target number (0–15) which the SCSI controller should use to identify itself on the specified bus.

If you expect to communicate with any 8–bit SCSI devices, do not use SCSI target numbers above 7, as these devices which don't implement target numbers greater than 7. The OFFLINE keyword will set the specified bus off line. That is it will not be used for any purpose until a valid SCSI target number has again been set up for it.

`SET TIMEZONE hh:mm`

Set the current local timezone as the specified number of decimal hours and minutes different from Greenwich. The value may be between –12:00 and 12:00 inclusive. Timezones west of Greenwich are negative, while timezones east of Greenwich are positive. The values –12:00 and 12:00 are the same time, but on opposite sides of the international date line, and thus differ by 1 day. For example, the United States eastern timezone would be –5:00 (standard time is 5 hours and 0 minutes earlier than Greenwich). Users should note that the sign of this value differs from the equivalent one used in TOPS–20 (in the 7–SETSPD program when read from the 7–CONFIG.CMD file).

`SHOW` Show various system settings.

`SHOW ADDRESS-BREAK`

Print the contents of the program address break register.

`SHOW AUXILIARY-PORT`

Print the current state of the auxiliary terminal port.

`SHOW CAPACITY`

This command prints the capacity of all or selected direct–access devices. This is similar to the "SHOW CONFIGURATION MASS–STORAGE ... /CAPACITY" command, except that it includes only direct–access devices.

`SHOW CAPACITY arguments`

The following arguments may be used to include a subset of the mass–storage devices:

slot bus target–id

The "slot" argument specifies a mass–storage controller, the "bus" argument specifies a SCSI bus, and the "id" argument specifies a SCSI target id number. All arguments are in decimal. A missing argument or "*" will select all slots, buses, or ids corresponding to the argument.

`SHOW CONFIGURATION`

Print the dynamic system configuration. A slot number or a keyword may be entered.

`SHOW CONFIGURATION slot`

The "slot" argument is a decimal slot number (0–15) indicating the single slot for which to show the configuration. Slot 0 refers to this CPU. The slot number may be followed by switches appropriate to the type of device in that slot.

`SHOW CONFIGURATION CPU`

Show only the configuration of CPUs.

`SHOW CONFIGURATION CPU arguments`

The optional decimal slot number argument (0–15) indicates a specific CPU to include (0 is a synonym for this CPU). A missing argument or "*" will include all CPUs in the system.

`SHOW CONFIGURATION CPU switches`

Switches should be entered after all other arguments.

`/LONG`

Print a detailed configuration.

`/SHORT`

Print a minimum configuration.

`SHOW CONFIGURATION MASS-STORAGE`

Show only the configuration of mass–storage controllers.

`SHOW CONFIGURATION MASS-STORAGE arguments`

The following arguments may be used to include a subset of the mass–storage devices:

> slot bus target–id

The "slot" argument specifies a mass–storage controller, the "bus" argument specifies a SCSI bus, and the "id" argument specifies a SCSI target id number. All arguments are in decimal. A missing argument or "*" will select all slots, buses, or ids corresponding to the argument.

`SHOW CONFIGURATION MASS-STORAGE switches`

Switches should be entered after all other arguments.

`/CAPACITY`

Print the capacity of direct–access devices. The /LONG switch is set implicitly by this switch.

`/HOME-BLOCKS`

Print the home block information used by TDBoot for direct–access devices. The /LONG switch is set implicitly by this switch.

`/LONG`

Print a detailed configuration.

`/SHORT`

Print a minimum configuration.

`SHOW CONFIGURATION MEMORY`

Show only the configuration of memory modules.

`SHOW CONFIGURATION MEMORY` **`arguments`**

The optional decimal slot number argument (0–15) indicates a specific MEMORY to include. A missing argument or "*" will include all memories in the system.

`SHOW CONFIGURATION MEMORY` **`switches`**

Switches should be entered after all other arguments.

`/LONG`

Print a detailed configuration.

`/SHORT`

Print a minimum configuration.

`SHOW CONFIGURATION NETWORK`

Show only the configuration of network controllers.

`SHOW CONFIGURATION NETWORK` **`arguments`**

The following arguments may be used to include a subset of the network ports:

slot port

The "slot" argument specifies a network controller, and the "port" argument specifies a network port. A missing argument or "*" will select all slots or ports corresponding to the argument.

`SHOW CONFIGURATION NETWORK` **`switches`**

Switches should be entered after all other arguments.

`/LONG`

Print a detailed configuration.

`/SHORT`

Print a minimum configuration.

`SHOW CONFIGURATION SYSTEM`

Show only the system ID and options, and the backplane serial number and options.

`SHOW CONFIGURATION *`

Show the configuration of all devices in the system. This is the default if no arguments are specified after the SHOW CONFURATION command.

`/LONG`

Print a detailed configuration.

`/SHORT`

Print a minimum configuration.

`SHOW DAYLIGHT-SAVINGS`

Print the current DAYLIGHT–SAVINGS setting.

`SHOW HOME-BLOCKS`

This command prints the home blocks of all or selected direct–access devices. This is similar to the "SHOW CONFIGURATION MASS–STORAGE ... /HOME–BLOCKS" command, except that it includes only direct–access devices.

`SHOW HOME-BLOCKS arguments`

The following arguments may be used to include a subset of the mass–storage devices:

slot bus target–id

The "slot" argument specifies a mass–storage controller, the "bus" argument specifies a SCSI bus, and the "id" argument specifies a SCSI target id number. All arguments are in decimal. A missing argument or "*" will select all slots, buses, or ids corresponding to the argument.

`SHOW MEMORY-STATUS [slot]`

Print all or selected memory status. The optional decimal slot number argument (0–15) indicates a specific memory controller whose status should be printed. No argument indicates that status for all memory controllers should be printed. The printout includes both the status and error–summary registers. Note: reading the error–summary register clears the "Parity Error Detected" bit in the status register and releases the error–summary register to capture another error.

`SHOW PAGE-FAIL`

Print the data saved by the most recent page fail(s) in a user readable form. If this was the result of a recursive page fail, the preceding page fail information is also printed. Be careful that there are no intervening page fails or you will not get the results you expect.

`SHOW PAGE-FAIL CONSOLE`

Show page fail data which preceded the most recent recursive page failure which attempted to trap via the console.

`SHOW PAGE-FAIL IO`

Show page fail data which preceded the most recent recursive page failure which attempted to trap via the EPT.

`SHOW PAGE-FAIL LAST`

Show page fail data from the most recent page fail.

`SHOW PAGE-FAIL ROM`

Show page fail data which preceded the most recent recursive page failure which attempted to trap via the ROM.

`SHOW POWER-STATUS`

Print the current power status.

`SHOW PROGRAM`

Print information associated with the current program in memory.

`SHOW PROGRAM ENTRY-VECTOR`

Print the current program's entry–vector location and length.

`SHOW PROGRAM MEMORY-MAP`

Print all or some of the contents of the current program's memory map. By default, all executive pages (those mapped via the EPT) will be printed.

`/EXEC`

Limit information printout to executive pages (those mapped via the EPT).

`/NOPAGE-INFO`

Suppress printout of information on pages in each section. With this switch, only information about sections is printed.

`/SECTION:n`

Limit printout to the specified section. The argument is an octal section number (0–7776).

`/USER`

Limit information printout to user pages (those mapped via the UPT).

`SHOW PROGRAM PDVAS`

Print the current program's program data vector addresses (PDVAs).

`SHOW SCSI-IDS`

Print the dynamic initiator target ID number for each SCSI bus on each mass–storage controller in the system.

`SHOW STRUCTURE str:`

Print information about the units in the specified structure. The argument is a 1–6 character structure name followed by a colon (":").

`SHOW TIMEZONE`

Print the current local TIMEZONE offset.

`SHOW VERSION`

Print the version numbers of the BOOT ROM program and the current CPU microcode.

`SHOW ZONE-TABLE slot bus target [lun]`

Print the zone table for the specified disk drive. The first argument is the decimal slot number (1–15). The second argument is the bus number (0–3). The third arugment is the decimal target ID (0–15). The optional fourth argument is the logical unit number (LUN, 0–7, default 0).

`SHUTDOWN`

Set the shutdown request flag (AP%SHT) in the processor flags accessible by RDAPR and continue the program. A program can use this to receive shutdown requests by enabling an interrupt when this bit is set or by sampling the bit periodically.

`S[TART]` Set the program counter and continue the program. The optional argument can specify the absolute virtual address to load into the PC or a "+" followed by an entry–vector offset. If no argument is present, entry–vector offset 0 is used. The program flags and context are zeroed before starting, which starts in EXEC mode. If you do not wish the flags and context to be zeroed, consider using the "DEPOSIT FLAGS <n>" and "CONTINUE <pc>" commands. The optional switch /DEBUG will set CF%DBG in the control flags (WCTRLF/RCTRLF), and can be used to tell a program (such as the monitor) to run in debug mode.

`TEST` Perform testing of specific processor components and/or devices. A slot number or a keyword may be entered.

`TEST slot`

The "slot" argument is a decimal slot number (0–15) indicating the single slot to test. Slot 0 refers to this CPU. The slot number may be followed by switches appropriate to the type of device in that slot.

`TEST BUS`

Perform device specific tests on each slot except this CPU.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`TEST CACHE`

Perform the cache tests on this CPU.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`TEST CPU`

Perform only CPU tests. This currently includes the cache and pager tests.

`TEST CPU arguments`

The optional decimal slot number argument (0–15) indicates a specific CPU to test (0 is a synonym for this CPU). A missing argument or "*" will test all CPUs in the system.

`TEST CPU switches`

Switches should be entered after all other arguments.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`TEST INTERVAL-TIMER`

Run the interval timer at various interval values and compute the actual interval using RDTIME. Prints the interval being tested followed by the mean and variance for that interval.

`/INTERVAL:n`

Select a specific interval to test. The argument specifies the interval from 256 to 32768 microseconds. It will be rounded to the nearest multiple of 128.

`/LOOP-ON-ERROR`

Continue testing after printing an error.

`/QUIET`

Suppress informational output during testing. You will still get output if an error occurs.

`/SAMPLES:n`

The argument specifies the number of samples to take before computing average and variance. The default value is 100.

`TEST MASS-STORAGE`

Perform the mass–storage controller tests on all or selected mass–storage controllers and SCSI devices.

`TEST MASS-STORAGE arguments`

The following arguments may be used to select a subset of the mass–storage devices:

slot bus target–id unit

The "slot" argument specifies a mass–storage controller, the "bus" argument specifies a SCSI bus, the "id" argument specifies a SCSI target id number, and the "unit" argument specifies a SCSI logical unit number. A missing argument or "*" will select all slots, buses, ids, or units corresponding to the argument.

`TEST MASS-STORAGE switches`

Switches should be entered after all other arguments.

`/BYTE-OFFSET:n`

Specify the byte offset to use for data transfers. This can be used to test transfers which end in the middle of a word. (Byte–offset is effective only during tests of the industry–compatible data format.)

`/COMPARE`

During data tests, do write, read, and compare.

`/CORE-DUMP`

Test core dump (40–bit) data format.

`/DEVICE-BUFFERS`

Test the SCSI BUS. This consists of a WRITE BUFFER followed by a READ BUFFER and verify to each selected SCSI device which supports these commands.

`/DRAM`

Test mass–storage controller DRAM (internal buffer memory). Data is written from system memory into the DRAM and then read back and compared. If none of /DEVICE–BUFFERS, /DRAM, /MEDIA–TEST, /TIMING, or /VERIFY are specified, then /DEVICE–BUFFERS and /DRAM are the default.

`/ENDING-OFFSET:n`

Specify the ending word offset to use for data transfers. The argument is the offset backward from the end of the cache line of the last word to transfer (e.g., /ENDING–OFFSET:7 would transfer only the first word of the cache line at the end of the transfer). This can be used to test transfers which end in the middle of a cache line.

`/FLOATING-ZEROS`

Test using a floating–zeros data pattern.

`/FLOATING-ONES`

Test using a floating–ones data pattern.

`/FONES`

Test using a floating–ones data pattern.

`/FZEROS`

Test using a floating–zeros data pattern.

`/HIGH-DENSITY`

Test high–density (36–bit) data format.

`/INDUSTRY-COMPATIBLE`

Test industry–compatible (32–bit) data format.

`/JOHNSON-COUNTER`

Test using a Johnson counter data pattern.

`/MEDIA-TEST`

Test the medium on the SCSI device by reading each block into system memory. For direct–access devices (disks), the entire medium is read.

`/NOCOMPARE`

During data tests, do write and read, but no compare.

`/NOLONG-TRANSFERS`

Do not use long transfers during media test. This switch is meaningful only in conjunction with the /MEDIA–TEST switch.

/ONES

Test using a data pattern of all ones.

/REPEAT:n

Repeat the specified tests. The argument is the number of times to repeat.

/RETRIES:n

Retry after errors when establishing the communications region. The argument is the number of times to retry before giving up.

/RONLY

During data tests, do read, but no write or compare.

/STARTING-OFFSET:n

Specify the starting word offset to use for data transfers. The argument is the offset forward from the beginning of the cache line of the first word to transfer (e.g., /BEGINNING–OFFSET:7 would begin the transfer with the last word of the cache line). This can be used to test transfers which start in the middle of a cache line.

/TIMING

Test various timing parameters of selected SCSI devices.

/VERIFY

Test the medium on the SCSI device by issuing the SCSI VERIFY command. Since not all devices support this command, it may fail. If so, use the /MEDIA–TEST option, which will read each block into memory. For direct–access devices (disks), the entire medium is read.

/WONLY

During data tests, do write, but no read or compare.

/ZEROS

Test using a data pattern of all zeros.

TEST MEMORY

Test system memories.

TEST MEMORY arguments

The optional decimal slot number argument (0–15) indicates a specific memory to test. A missing argument or "*" will test all memories in the system.

TEST MEMORY switches

Switches should be entered after all other arguments.

/ADDRESS

Select the memory address test. If no memory tests are selected, all are performed. An “A” is printed to indicate that the address test is being initialized. During initialization, each word is written with its bus address word (BAW). An “a” is printed to indicate that the address–test compare has started. Each word of memory is checked to verify that it still contains the correct data.

`/COMBINED`

Select the combined memory tests. If no memory tests are selected, all are performed. A “C” is printed to indicate combined testing is in progress. A sequence of data patterns and addresses is used which will exercise the cache refill, writeback, and flush operations.

`/DATA`

Select the memory data test. If no memory tests are selected, all are performed. A “D” is printed to indicate that the data test is being initialized. During initialization, each word is written with the negative of its bus address word (BAW) minus two. A “d” is printed to indicate that the data–test compare has started. Each word of memory is checked to verify that it still contains the correct data.

`/NOCACHING`

Disable use of the cache to accelerate the operation. This will force direct memory accesses even though they are slower. Use this if you think there are problems with the cache. Caching will automatically be disabled if the CACHE–TEST startup parameter is disabled, or if there were initialization errors. Since using the cache requires using the pager, caching is implicitly disabled if paging is disabled (PAGER–TEST is disabled, or /NOPAGING was specified).

`/NOPAGING`

Disable use of paging to accelerate the operation. This will force the use of PMOVEM even though it is slower. Use this if you think there are problems with the pager. Paging will automatically be disabled if the PAGER–TEST startup parameter is disabled, or if there were initialization errors. Since using the cache requires using the pager, caching is implicitly disabled if paging is disabled.

`/PROCEED-ON-ERROR`

Causes the memory test to proceed despite errors. Errors will continue to be printed, and a failing memory will not be taken off line.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`/R[EADONLY]`

Perform a READONLY memory test. This test is performed before any others to avoid changing the contents of memory. A “R” is printed to indicate readonly testing is in progress. Each word of memory is read to determine if it has good parity. This test makes use of the pager and cache to accelerate operation. See the /NOPAGING and /NOCACHING switches for more information.

`/STATIC`

Select the static memory tests. If no memory tests are selected, all are performed. An “S” is printed to indicate static testing is in progress. Each static test is performed on a cache line in memory. Data patters of all ones, all zeros, a floating–one, and a floating–zero are written and compared in each word of the line. Cache line addresses of all zeros and all ones are performed first. This is followed by writing and detecting bad parity in cache line zero. Finally, cache line addresses first of a floating–one and then of a floating–zero are tested.

`TEST NETWORK`

Perform diagnostic tests on the specified network controllers.

`TEST NETWORK arguments`

The following arguments may be used to test a subset of the network ports:

slot port

The “slot” argument specifies a network controller, and the “port” argument specifies a network port. A missing argument or “*” will select all slots or ports corresponding to the argument.

`TEST NETWORK switches`

Switches should be entered after all other arguments.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`/TIMING`

Measure the amount of time it takes to reset the network controller.

`TEST PAGER`

Test this CPU’s pager.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`TEST POWER-FAIL`

Loop checking power status and reporting changes. Type ^C to terminate.

`TEST TIME-BASE`

Check RDTIME timebase for monotonicity and no large changes. Prints elapsed time approximately every 3 seconds.

`/LOOP-ON-ERROR`

Continue testing after printing an error.

`/QUIET`

Suppress informational output during testing. You will still get output if an error occurs.

`TEST *`

Test this CPU and each other system slot. This is the default if no arguments are specified after the TEST command.

`/REPEAT:n`

Repeat the specified tests. The argument is the number of times to repeat.

`UNLOAD scsi-device-specifier`

Unload the specified SCSI device (usually a tape or other removable medium). The device is specified using one of the following formats:

`# slot bus target [unit]`	To specify an explicit SCSI nexus
`MTAnnn:`	To specify a SCSI tape by its sequential nexus position
`channel, controller, unit`	Operating system tape specifier

`VDIRECTORY [path]`

This is a shorthand for the "DIRECTORY args /LONG" command. See the "DIRECTORY" command for a full description of the arguments.

E.2 Micro–console commands

A command beginning with a "." is actually processed by the micro–console. Help for the micro–console commands is provided here as a convenience to the user. Micro–console commands consist of the initial "." followed by one or more letters and a variable number of numeric arguments. All numeric arguments are in octal. The following is a summary of the available commands:

Command	Description
`.B offset`	Start TDBoot
`.C`	Continue program
`.D l [adr] data`	Deposit things
`.E [l [adr]]`	Examine things
`.H`	Halt program
`.I`	Initialize microcode
`.M`	Enable macro–console
`.R cnt cmd`	Repeat command
`.S adr`	Start program
`.T l [args]`	Test things
`.U`	Disable macro–console
`.V`	Type version
`.W data`	Write control flags

`.B` Start TDBoot as a normal program. The TDBoot code runs out of ROM and parses its own commands. When running in this mode, micro–console commands are not available. To get to the micro–console, type control–backslash (^\). Terminate this mode of TDBoot by typing "EXIT" or control–Z (^Z).

`.C` Continue the current program if the PC is valid. After this command, the program will run until it executes a HALT instruction or is halted by a console command.

`.D` Deposit things in various parts of the machine. The next letter specifies where to deposit. The last letter used is remembered and is used as the default thing to examine in the next ".E" command. The following is a summary of the deposit commands:

Command	Description
`.D A adr data`	Deposit MemA
`.D D baw data`	Deposit device–control
`.D F data`	Deposit flags and context
`.D I data`	Deposit PIR
`.D M baw data`	Deposit physical memory
`.D N adr data`	Deposit NVram
`.D O om adr data`	Diag write, om –> cache mode; adr –> E; data –> IABUS
`.D P data`	Deposit PC
`.D R reg data`	Deposit current–context register
`.D S adr data`	Deposit system virtual memory
`.D U adr data`	Deposit user virtual memory

`.D A address data`

Deposit into MemA. The first argument is the 13–bit address (0–17777). The second argument is the 36–bit data to deposit.

`.D D baw data`

Perform a device–control cycle on the system bus (this is also known as a write to I/O space). The first argument is the bus address word (BAW). The second word is the data to write. Bit 0 of the address controls cache access. It should never be set when accessing I/O space.

`.D F data`

Store into the internal processor flags and context word. The argument is a 36–bit number specifying the processor flags in bits 0–12, the current AC block in bits 18–20, the previous AC block in bits 21–23, and the previous section in bits 24–35.

`.D I data`

Store into the internal priority–interrupt register (as read by RDPI). Changing these values may cause certain inconsistencies in the internal machine state; use with great care.

`.D M baw data`

Perform a word write on the system bus (this is also know as a write to memory space). The first argument is the bus address word (BAW). The second word is the data to write. Bit 0 of the address controls cache access. If bit 0 is clear, the cache will be bypassed. If bit 0 is set, the cache will be flushed and/or loaded as needed and the data written into the proper cache line. Care should be taken to use the same cache access as is currently being used for the specified address; otherwise cache inconsistencies may occur.

`.D N address data`

Write one byte into the nonvolatile RAM (NVRAM). The first argument is a 13–bit address (0–17777). The second argument is the 8–bit data (0–377) to be written. Use care when changing NVRAM, because it controls various pieces of system configuration.

`.D O mode address data`

Perform a diagnostic write cycle. The first argument is loaded into the cache mode. The second argument is loaded into the internal DPM address register. The third argument is loaded into the IABUS latches. A DPM write cycle is performed.

`.D P address`

Store into the macro PC. This is similar to the .S command, but the macro program is not continued.

`.D R n data`

Store into the registers selected by the current AC block context setting. The first argument is the 4–bit register number. The second argument is the 36–bit data to write.

`.D S address data`

Write to a system virtual address. The first argument is the 30–bit address. The second argument is the 36–bit data to write. A system virtual address is one which is mapped starting at EPT offset 540 plus the super–section number.

`.D U address data`

Write to a user virtual address. The first argument is the 30–bit address. The second argument is the 36–bit data to write. A user virtual address is one which is mapped starting at UPT offset 540 plus the super–section number.

`.E` Examine things in various parts of the machine. The next letter specifies what to examine. The following is a summary of the examine commands:

Command	Description
`.E`	Repeats previous examine with next address or examines the same address as the previous deposit, whichever occured most recently.
`.E A adr`	Examine MemA
`.E D baw`	Examine device–control
`.E F`	Examine flags and context
`.E I`	Examine PIR and HW interrupt register
`.E L adr`	Examine lookup ROM (<0:12>, <31:35>, or <18:35>)
`.E M baw`	Examine physical memory
`.E N adr`	Examine NVram
`.E O A om adr`	Diag read, om -> cache mode; adr -> E; print IABUS
`.E O D om adr`	Diag read, om -> cache mode; adr -> E; print DPM
`.E O N om adr`	Diag read, om -> cache mode; adr -> E; no print
`.E P`	Examine PC
`.E R reg`	Examine current–context register
`.E S adr`	Examine system virtual memory
`.E U adr`	Examine user virtual memory

`.E A address`

Examine locations in MemA. The argument is the 13–bit address (0–17777).

`.E D baw`

Perform a status request cycle on the system bus (also know as an I/O space read). The argument is the bus address word (BAW) to read. Bit 0 controls cache access. Bit 0 should not be set with this command.

`.E F`

Examine the internal processor flags and context. The processor flags are in bits 0–12, the current AC block is in bits 18–20, the previous AC block is in bits 21–23, and the previous section number is in bits 24–35.

`.E I`

Examine the priority–interrupt register and the hardware interrupt register. The priority–interrupt register contains the same data as read with RDPI. The hardware interrupt register has hardware bits available to the microcode.

`.E L address`

Examine the internal lookup ROM used for byte instructions. The argument specifies the address in the ROM to examine. First, if bits 0–17 are zero, copy bits 18–30 to bits 0–12. The address is then formed from bits 0–12 and bits 31–35.

`.E M baw`

Perform a word–read cycle on the system bus (also know as a memory–space read). The argument is the bus address word (BAW) to read. Bit 0 controls cache access. If bit 0 is clear, the cache is bypassed. If bit 0 is set, the cache will be examined to see if the specified address is cached there. If not, the corresponding cache line will be loaded from system memory into the cache and the desired will be word returned.

`.E N address`

Examine a byte of nonvolatile RAM (NVRAM). The argument is the 13–bit address of the desired byte.

`.E O`

Perform a diagnostic–read. The first argument is a letter specifying what to type out upon completion. The second argument is loaded into the cache mode. The third argument is loaded into the internal DPM address register. The following is a summary of the diagnostic–read commands:

Command	Description
`.E O A om adr`	Diag read, om –> cache mode, adr –> E, print IABUS
`.E O D om adr`	Diag read, om –> cache mode, adr –> E, print DPM
`.E O N om adr`	Diag read, om –> cache mode, adr –> E, no print

`.E O A mode address`

Perform a diagnostic–read cycle and print the resulting contents of IABUS. The first argument is loaded into the cache mode; the second argument is loaded into the internal DPM address register.

`.E O D mode address`

Perform a diagnostic–read cycle and print the resulting contents of DPM. The first argument is loaded into the cache mode; the second argument is loaded into the internal DPM address register.

`.E O N mode address`

Perform a diagnostic–read with no output. The first argument is loaded into the cache mode; the second argument is loaded into the internal DPM address register. diagnostic–read with no output is sometimes useful for scoping a processor problem.

`.E P`

Print the current macro PC.

`.E R n`

Examine the registers selected by the current AC block context setting. The argument is the 4–bit register number (0–17).

`.E S address`

Read a system virtual address. The first argument is the 30–bit address. A system virtual address is one which is mapped starting at EPT offset 540 plus the super–section number.

`.E U address`

Read a user virtual address. The first argument is the 30–bit address. A user virtual address is one which is mapped starting at UPT offset 540 plus the super–section number.

`.H` Halt the currently running program. The PC remains valid, and the program can be continued where it left off using the .C command.

`.I` Restart the CPU microcode at the beginning. The CPU cannot tell the difference between this and starting from power–on.

`.M` Initialize the macro–console by starting it at its entry–vector offset 3. Whenever the micro–console passes a command to the macro–console, it clears the macro–console enable bits stored in MemA. If the macro–console terminates without setting this bit again, it is effectively disabled (i.e., the micro–console will not pass commands to it). The .M command gives the macro–console a chance to re–enable itself. (The macro–console is also disabled by the .U command or by installing option jumper J0.)

`.R n command`

Repeat any other micro–console command a specified number of times. The argument following .R is an number specifying the number of times to repeat the command. A value of zero will cause command to repeat indefinitely. Following the repeat count, you should give the command you wish repeated, but without the initial dot (“.”). For example, the following commands examine all 16 (decimal) current–context registers:

```
.E R 0
.R 17 E
```

The first command examines register 0, while the second command examines the next 17 (octal) registers in sequence.

`.S address`

Start the processor executing instructions. The argument is a 30–bit address specifying the intial PC.

`.T` Perform various low–level tests which are difficult or impossible to perform using macro–code, or when macro–code execution is not operational. The following test commands are available:

Command	Description
`.T A d`	W/R alternate MemA locs with "d" and "d" with its low order bit complemented, respectively.
`.T B`	Read BP power status and ROMs
`.T C b`	Test cache interaction with memory at BAW "b"
`.T M b s p c`	Test system memory: start at BAW "b", for "s" words, with pattern "p". "c" has control bits.
`.T S d s m`	Rotate "d" by "s", AND with "m"
`.T U b`	Test cache invalidate/flush functions
`.T X h1 l1 h2 l2`	Do 72–bit add of h1,l1 and h2,l2

`.T A data`

Test MemA. The argument is a data pattern. Two locations in MemA are used. The data is written to the first location. Then the data written to the second location with the low–order bit complemented. This changes the parity bit. Both locations are then read to verify parity. This process repeats until a character is entered on the console.

`.T B`

Read the backplane power status, serial number and system ID ROMs. Then read the CPU serial number ROM into predefined locations in MemA. Print the power status (it appears in the hi–order 8 bits of value printed). The bits are interpreted as follows:

Bit	Mask	Description
0	400000000000	AC fault
1	200000000000	Thermal fault
2	100000000000	Battery bad
3	040000000000	Battery low
4	020000000000	Need power asserted on backplane
5	010000000000	Not used, should be zero
6	004000000000	Backplane wedged
7	002000000000	Always set

`.T C baw`

Perform a variety of cache/memory interaction tests. The argument is the base bus address word (BAW) of the memory locations to use. The addresses used are BAW through BAW+37777777. Since the page offsets are fixed for mapped addresses, the BAW should have bits <27:35> set to zero. If a test fails, the address, actual, and expected data are printed and the test is terminated. The test can also be terminated by typing on the console.

`.T M baw count pattern control`

Perform a variety of system memory tests. The first argument is the base bus address word (BAW) of the locations to be tested. The second argument is the count of locations to test (0 will count for a long time). The third argument is a data pattern. The fourth argument is the function, encoded as follows:

Bits	Mask	Function
`33:35`	0	Use the data pattern unchanged
	1	Complement the pattern on each pass
	2	Increment the pattern on each pass
	3	Rotate the pattern left one bit on each pass
18	400000	Continue on error
15:17	0000000	Init pass only (places pattern in memory)
	1000000	Init only, pattern XOR'd with BAW
	2000000	Write/read/compare using pattern
	3000000	Write/read/compare, pattern XOR'd with BAW
	4000000	Write only (no read or compare)
	5000000	Read only (no write or compare)
	6000000	Read/write (no compare)
	7000000	XOR/read/write (no compare)

When comparing and stopping on errors (bit 18 clear), an error will terminate testing with a printout of the address, actual data, and expected data. The test can also be terminated at the end of a pass by typing on the console.

`.T S data rotate mask`

Test the internal shifter/masker hardware. The first argument is the data. The second argument is the shift amount. The third argument is the mask.

`.T U baw`

Perform the invalidate and flush tests on the CPU cache. The argument is the base bus address word (BAW) of the memory locations to use. The addresses used are BAW through BAW+377777. Since this test uses the right half of the BAW as a counter, it should have bits <18:35> set to zero. If a test fails, the address, actual, and expected data are printed and the test is terminated. The test can also be terminated by typing on the console.

`.T X h1 l1 h2 l2`

Test the internal 72–bit adder. The first argument is the high–order word loaded into the B register. The second argument is the low–order word loaded into the B register. The third argument is the high–order word loaded into the A register. The fourth argument is the low–order word loaded into the A register. The two registers are added and the two word result is printed, high–order word first.

`.U` Disable the macro–console. This may be useful to prevent inadvertent running of macro–code by entering a command without an initial dot ("."). Use the .M command to re–enable the macro–console. (The macro–console is also disabled by installing option jumper J0.)

`.V` Print the processor description string and microcode version number.

`.W data` Perform a WCTRLF instruction using the argument. This is provided to allow manipulation of the auxiliary port when the macro–console may not be running. The "data" argument contains the WCTRLF bits. The bits which control the auxiliary port are:

Name	Value	Function
`CF%SET`	1B0	Set selected bits
`CF%CLR`	1B1	Clear selected bits
`CF%DTR`	200	Auxiliary Data Terminal Ready
`CF%APE`	10	Auxiliary Port Enable
`CF%RTS`	2	Auxiliary Request To Send

To enable the auxiliary port, type ".W400000000212"; to disable the auxiliary port, type ".W200000000212".

Appendix F

XKL-1 Processor Arcana

This appendix describes details of the XKL-1 processor that are so implementation–specific as to be **excluded** from the architectural specification of the TOAD-1 System.

Specific locations in MemA and NVRAM identified in the manual are included for convenience of the authors of the processor microcode, TDBoot, and diagnostics. **This material is subject to change.**

F.1 MemA Specific Locations

Note that special instructions have been provided to change particular locations in MemA. When such an instruction has been provided, it may cause side–effects that are necessary for the proper operation of the system. For example, although the user base register and the executive base register are implemented as locations in MemA, those locations should not be addressed via the AMOVEM instruction, because changing these elements requires that the Pager Translation Buffer be invalidated; such is the effect of the WREBR and WRUBR instructions.[1]

0–177	Fast–memory (AC) blocks 0–7. Fast–memory block number n starts at address $20\times n$. `AM%AB0==:0` ... `AM%AB7==:160`
200–217	`AM%LPN==:200` 16 words, indexed by physical slot number, containing the first linear page number of the memory in the given slot. For memory devices, bit 0 will be set. If the pager refill code encounters an entry in which bit 0 is clear, the CST update will be skipped. For (the non–existent) slot 0, the data is the system total memory capacity, in pages. (This table is used by the LDLPN instruction, as well as by the pager refill microcode.)
220–237	`AM%DVT==:220` 16 words, indexed by physical slot number, containing the corresponding device's response to a "Device Status" request directed to its address zero. This data can be interpreted to show the system hardware configuration.

---

[1] The RDEBR and RDUBR instructions should be used in preference to the corresponding AMOVE instructions, for compatibility with future systems.

240–257	`AM%MCP==:240` 16 words, indexed by physical slot number, containing the capacity of this slot's memory device, in pages, or zero if the device is not a memory.
264	`AM%CNF==:264` Memory configuration flags. Bit 0 (`VALCN%==:1B0`) set means the configuration is valid. Bit 1 (`REVCN%==:1B1`) set means that memory is configured in reverse: the memory at the highest slot number is mapped to the lowest linear address. Bit 2 (`FORCN%==:1B2`) means that the configuration was forced.
266-267	`AM%OFL==:266`, `AM%OPC==:267` Flags and Context, and PC at latest trap, MUUO, or interrupt.
300	`AM%EBR==:00300` Executive base register. Use WREBR to change this value.
301	`AM%UBR==:00301` User base register. Use WRUBR to change this value.
302	`AM%CTX==:00302` Process context word: current AC block, previous AC block, and previous–context PC section. Change this by means of WRCTX.
303	`AM%SPB==:00303` SPT base address. Change this via WRSPB.
304	`AM%CSB==:00304` CST base address. Change this via WRCSB.
305	`AM%PUR==:00305` CST process use register (Data Word). Use WRPUR to change this value.
306	`AM%CSM==:00306` CST mask word. Use WRCSTM to change this value.
307	`AM%ADB==:00307` Address–break register. Use WRADB to set this value.
310–311	`AM%TIM==:00310` Time–base double word.
312	`AM%CTI==:00312` Most recent CTY input character.
313	`AM%CTS==:00313` Console status. (Kept by microcode). This is read by RDCTYS.
314	`AM%HPM==:00314` Hard page-failure mask. When a hard page–failure occurs, if EPT 500 AND this mask is nonzero, the hard page–failure will trap to the macro–console by entering it at offset 6 from its starting address.
320–322	`AM%SY0==:320` These locations hold data equivalent to that reported by APRID. These locations are readable by other devices via device status requests to addresses 0–2, respectively.
323–327	These locations (`AM%SY3==:323`—`AM%SY7==:327`) are readable by other processors via device status requests addressed to locations 3–7, respectively. These are used for inter–processor synchronization before the memory and operating system are fully functional (§3.10).
334–343	These locations are reserved for the Macro console. `AM%MBT==:334` Macro console state, shared by TDBoot and the microcode: • `MS%VAL==:1` Macro PC is valid. • `MS%RUN==:2` Macro code is running (`MS%VAL` will be set).

- `MS%UCA==:100000` Micro–console is active. This can be cleared by the macro console.
- `MS%MCA==:200000` Macro–console is active. This is cleared by PI reset.
- `MS%MCE==:400000` Macro–console is enabled. (This flag is cleared when the macro console is entered and it is set when the macro console completes a command; if the macro console halts without setting this flag, the macro console is disabled.)

`AM%MFG==:335` Saved macro flags.

`AM%MPC==:336` Saved macro PC.

`AM%MEB==:337` Saved macro EBR.

`AM%MUB==:340` Saved macro UBR.

`AM%MPI==:341` Saved macro highest priority level being held.

`AM%MCM==:342` Pointer to the macro console's command string in MemA.

`AM%MCS==:343` Saved macro CSB.

500–507	This region, which starts at `AM%PFN==:500` is used to record hard page–fail data. This is the same information as is stored in Executive Process Table location 500 (`UP.PFB`). Because some hard page–fail situations result from failure to access main memory, the data is recorded here also.
500	`AM%PFB==:500` Implementation-specific hardware page–fail bits.
501	`AM%PFD==:501` This location contains data copied out of the processor's "D to D" latch. This information may be of use to engineers in tracking down the precise nature of the page–failure. This is the same information as is stored in Executive Process Table location 501 (`UP.PFD`).
502–503	`AM%PF0==:502`, `AM%PF1==:503` Most recent page–fail double word. This is the same information as is stored in User Process Table locations 502–503 (`UP.PF0`, `UP.PF1`), respectively. However, while TDBoot is using the ROM–based vestigial UPT, this information can not be written in memory, but it can be found here.
504–505	`AM%POF==:504`, `AM%POP==:505` Flags and Context, and PC of the most recent page–fail trap. This is the same information as is stored in User Process Table or Executive Process Table locations 505–506 (`UP.POF`, `UP.POP`), respectively. These locations are used by TDBoot while using the ROM–based vestigial EPT/UPT. (These location are used regardless of whether the trap is "hard" or "soft".)
506–507	Reserved. These locations correspond to User Process Table or Executive Process Table locations 506–507, the new Flags and Context, and PC words (`UP.PNF`, `UP.PNP`), respectively.
510-517	`AM%PD0==:510` This block of 8 locations contains implementation–specific pager and cache diagnostic data.
520-537	`AM%PFI==:520` Copy of 500–517 at I/O Page Fail.
540-557	`AM%PFR==:540` Copy of 500–517 at ROM–fallback Page Fail.

560-577 `AM%PFC==:560` Copy of 500–517 at Console–fallback Page Fail.

1000–1777 `AM%PFL==:1000` This is the page fail logging region, which contains 1000 (`PFL.SZ==:1000`) locations. The first word contains a count of words in use. This is followed by logging blocks. The first word of a logging block contains a block type in the left–half word and the block size in the right–half word. The defined block types are

- `PFL.CB==:1` A cache block. Each entry is three words, as supplied by DRDCSH.
- `PFL.PB==:2` A pager block. Each entry is three words, as supplied by DRDPTB.
- `PFL.MB==:3` A memory block. Each entry is two words, a BAW and data.

2033 Keep–Alive counter. Set from the value specified in the WRKPA instruction. If this cell is not zero when the 16–bit time–base overflows (approximately every 32.8 milliseconds, while the machine is running), the processor will decrement the value stored here. When the processor decrements this cell to zero, the processor performs a Keep–Alive interrupt. (See §3.8.3.)

2053–2055 These three consecutive locations contain the User map cache. This data is used to shorten the pager refill process. Specifically, this data is the section numbers of last two User Virtual sections for which sucessful refills have been done, and pointers to the page maps for those sections. On each User mode page refill, if virtual address bits 6–17 match the either of the values recorded here, the refill process takes a shortcut to the given page map and permissions. This data is cleared by WREBR and CLRPT.

2056-2060 Three consecutive locations, the Executive map cache.

F.2 NVRAM Specific Locations

The following are some of the parameters stored in NVRAM:

- Device–specific initial parameters. A block of thirty–two (40 octal, `NV%DVS==:40`) consecutive locations is provided for each device on the backplane bus. For each slot, the block starts at address *Slot* × 40. Locations 40–777 are allocated in this way.

  Among other things, these locations store the system's internet (IP) address on each of the networks to which it is connected, and the SCSI device identification numbers of the XRH-1 initiators.

Because there is no slot numbered 0, locations 0–37 are used for other purposes:

- `NV%DVT==:0` The bus configuration region, sixteen locations indexed by slot number. One byte of device–type information is stored per slot. Location 0 stores the slot number of this CPU board.
- `NV%DLY==:21` The number of seconds to delay in TDBOOT before attempting to become the master processor (in a multiprocessor system). This allows the system manager to bias the selection in favor of a particular processor.

- `NV%BPM==:22` Parameters for TDBoot.
- `NV%ATP==:23` Parameters for the auxiliary terminal port.
- `NV%ABD==:24` The length of time, in seconds, to delay in TDBoot prior to performing the automatic boot function.
- `NV%TZH==:25` The local time zone, expressed as the number of hours east of GMT. (Locations west of GMT are represented as negative numbers.) This item, along with the next two items, affect how TDBoot converts "universal time" when converting it to a human–readable format.
- `NV%TZM==:26` The minutes component of the local time zone, if needed to express a fractional hour.
- `NV%DST==:27` This value governs TDBoot's understanding of Daylight Savings Time and how to apply it to the conversion of dates and times. The value 0 (`.DSTAU`) directs TDBoot to apply daylight savings time automatically to applicable dates in April through October. The value 1 (`.DSTNV`) means to apply daylight savings time never, and the value 2 (`.DSTAL`) means to apply daylight savings time always. The latter two values allow for the manual control of daylight savings time when the automatic algorithm does not correspond to local custom.
- `NV%SOL==:30; and 31` The value held in these two locations define which slots have been declared off–line. TDBoot will not attempt to test or use a device in any slot that has been set off–line.

Some addresses at the high end of NVRAM also have assigned meanings:

- `NV%MA0==:17376`, `NV%MA1==:17377` Two locations containing "magic numbers" to signify that the NVRAM has been initialized by TDBoot.
- `NV%BPA==:17500` Default path names for the `Boot` command. Sixty–four locations are allocated for the default path name for the `Boot` command, and the default path name for the `Dump` command (`NV%NML==:100`).
- `NV%DPA==:17600` Default path name for the `Dump` command.
- `NV%UCR==:17700` Sixty–four locations reserved for the microcode. The processor microcode uses location `NV%FLG==:17777` to determine whether the NVRAM battery is functional or not.

F.3 XKL-1 Board Option Jumpers

There are three sets of jumpers. The locations of the jumpers are described assuming a normal orientation of the CPU board: top edge up, component side facing you.

Set J1: Boot ROM size. These jumpers are located below the leftmost Boot ROM. There are three pins, numbered 3, 2, and 1, with number 1 being at the right. Connect 2–1 for 256K ROMs; connect 3–2 for 512K ROMs. This is manufactured with a soldered wire, because the change is not to be done casually.

Set J2: Options. These are readable by the processor in the APRID instruction. J2 is located near the auxiliary console connector. J2–0 is at the top.

J2–0, if installed, is interpreted by the microcode to disable the macro–console.

- J2–1, reserved.
- J2–2, reserved.
- J2–3, reserved.

Jumper J3: Enable automatic restart on microcode parity error, when installed. This jumper is located below the R–Bus connector (the diagnostic connector), near the left edge of the board.

Appendix G

Non–existent Appendices

Further appendices pertaining to the processors built by Digital Equipment Corporation have not yet been incorporated in this manual.

These vestigal sections are present to satisfy textual cross–references.

This manual continues at the Glossary.

G.1 Timing

G.2 Processor Operation

G.3 Handling Memory

Appendix H

Glossary

A: the accumulator field of a instruction word.

AC: an accumulator number in the range 0 to 17_8.

Address Break: a trap that occurs when the processor references the address specified in the address break register.

Address Failure: see Address Break.

AFI: Address Failure Inhibit. A processor flag, which when set, allows the next instruction to be executed without an an address break (address failure) trap. Customarily, this flag is set by a JRSTF or XJRSTF to allow an instruction that previously caused an address break to be continued past. This flag is cleared when an instruction (other than a JRSTF) completes.

Alignment: in a byte, the number of bits at the left–end of the word to the left of as many bytes of this size and alignment that can be fit into a word. Mathematically, $(36 - P) \bmod S$, where P is the position of the byte (measured in bits to the right of the right–most bit in the byte) and S is the byte size.

APR: the Arithmetic Processor. In the KL10 and earlier systems, the device address of the

BAW: Bus Address Word. A 36–bit quantity that specifies a slot number, an in–module address, and whether to address the slot as a device or as a memory.

CAC: Current context AC block. The AC block in use by the program that is currently executing.

CD: Carrier Detect. A signal from a DCE to a DTE signifying that a connection between two modems has been establised.

CSB: CST Base Register. This register contains the physical address (PAW) of the CST and a flag to determine whether or not the CST is cacheable.

CST: Core (memory) Status Table. An array, consisting of one word for each page of physical memory, indexed by the LPN. The CST contains such data as the page age, page state, the cacheability of the page, and whether the page has been modified since last read from disk into memory.

CSTM: CST Mask Word.

CTY: Console Teletype. Now the console terminal, or the auxiliary console terminal, or the communication port through which either is connected.

DCE: Data Communications Equipment. A modem, or a device wired as a modem. Contrast to DTE, to which a DCE connects.

DTE: Data Terminal Equipment. A terminal, computer, or other device wired as a terminal. Contrast to DCE, to which a DTE connects. To connect two DTEs together, e.g., a computer and a terminal, a cable wired as a “null modem” is required.

DTR: Data Terminal Ready. A signal from a DTE to a DCE signifying that the DTE is ready to communicate.

E: Effective Address.

EA: Effective Address.

EBR: Executive Base Register. This contains the address (a PAW) of the EPT.

Effective Address: The numeric result of a computation performed for every instruction; the result may be used as a number (in an immediate instruction), as a shift factor, or as a memory address.

EPT: Executive Process Table. A data structure that describes the Executive address–space, trap words for the Executive, etc.

Flags: individual bits that represent the state and previous condition of the program.

Interrupt: an asynchronous break in the usual flow of a program’s execution. Peripheral devices cause interrupts when they need attention from the Monitor. The interval timer causes interrupts so the Monitor can obtain control, periodically, from compute–bound programs.

Linear Page Number: A numeric index by which each page of memory can be identified.

LPN: Linear Page Number.

LSB: Least Significant Bit. The rightmost bit in an arithmetic operand.

LUUOs: Local unimplemented user operation. An operation code, not implemented by the processor, but reserved for user–controlled program traps. An LUUO is, in effect, another kind of subroutine call.

Mass–Storage Control Block: A data structure by which the system software communicates to the XRH-1 and vice–versa.

MCB: Message Control Block. A data structure by which the system software communicates to the XNI-1 and vice–versa.

MemA: processor private memory. These locations include the accumulator blocks.

MSB: Most Significant Bit. The leftmost bit in an arithmetic operand that differs from the sign bit.

MSCB: Mass–Storage Control Block.

MUUO: Monitor unimplemented user operation: an instruction code that causes a trap to the Monitor, because either a Monitor call was intended or the program has blundered.

XNI-1: The network interface.

No–op: No operation. An instruction that has no overt effect. Note that some such instructions have side effects that may be significant. For example, SKIP reads memory and MOVES both reads and writes memory.

NVRAM: Non–Volatile Random Access Memory. A RAM that remembers data even when the power is turned off.

PAC: Previous context AC block. The AC block number used by the previous context program. This is where the target of a PXCT instruction will look for data when an AC is addressed as a memory operand.

Page–Failure: an exception condition during the execution of an instruction. Most often a page–failure represents an inability to translate a virtual address to a physical address. Page–failure is also used to signal other problems and conditions.

PAW: Page Address Word. A Bus Address Word shifted right by 9 bits. It specifies a slot number and an in–module page number.

PC: Program Counter. The location (an address) of the next instruction to execute.

PCS: Previous Context Section. The section in which the previous context program was operating. This is value is supplied for the section when the target of a PXCT instruction specifies a local address.

PCU: Previous Context User. This flag bit, set in an exec mode PC, signifies that the previous context was user mode. The setting of this bit affects the operation of PXCT.

PI: Priority Interrupt.

PTB: Page Translation Buffer. A two–way associative memory by which the pager translates virtual addresses to physical addresses.

PUR: Process Use Register.

RAM: Random Access Memory.

RI: Ring Indicate. A signal from a DCE to a DTE signifying that the telephone is ringing.

RTS: Request to Send.

SCSI: Small Computer System Interface a standard that specifies the electrical and command format interfaces for peripheral devices.

Section: A 256 Kw (512 page) region of memory identified by a constant value in address bits 6–17. A section may be physical, in which case the address bits referred to are physical address bits. A section may be virtual, with the addresses being virtual. In the unextended processors, the virtual address space of any process is limited to one section, section zero, in which the address bits 6–17 are all zero.

Supersection: A XKL-1–specific division of the virtual address space, into regions of 512 sections, in which address bits 6–8 are held constant. At the present time, operating system software on the XKL-1 supports only supersection zero, in which address bits 6–8 are zero.

SPB: SPT Base Register. This contains the physical address (a BAW) for the SPT.

SPT: Special Page–Address Table. Each entry in this table contains a PAW that specifies the address of a page table. SPT entries are used in the evaluation of shared and indirect page pointers.

Trap: A synchronous break in the usual flow of a a program's execution. Traps are used to detect arithmetic overflow and stack overflow conditions. Unimplemented instructions are also said to "trap", which means they are executed as MUUOs.

UART: Universal Asynchronous Receiver and Transmitter. A device that translates characters to serial data for transmission and that assembles serial data into characters for reception. The interface to a serial line.

UBR: User Base Register. This contains the address (a PAW) of the UPT.

UPT: User Process Table. A data structure that describes the user address–space, trap locations, page–fail handlers, etc.

XRH-1: The SCSI IO interface.

Index of Instructions

Index