digital

decsystem10

assembly language handbook

third edition

system reference

macro

monitor calls

link-10

ddt

utilities

decsystem10 handbook series

digital

decsystem10

assembly language handbook

third edition

Additional copies of this handbook may be ordered from:
Software Distribution Center, DEC, Maynard, Mass. 01754. Order Code: DEC-10-NRZC-D.

decsystem10 handbook series

The material in this handbook, including but not limited to instruction times and operating speeds, is for information purposes and is subject to change without notice.

Copyright © 1967, 1968, 1969, 1970, 1971, 1972, 1973 by
Digital Equipment Corporation

Actual distribution of the software described in this specification will be subject to terms and conditions to be announced at some future date by Digital Equipment Corporation.

DEC assumes no responsibility for the use or reliability of its software on equipment which is not supplied by DEC.

The software described in this manual is furnished to purchaser under a license for use on a single computer system and can be copied (with inclusion of DEC's copyright notice) only for use in such system, except as may otherwise be provided in writing by DEC.

The following are trademarks of
Digital Equipment Corporation, Maynard, Massachusetts:

DEC	PDP
FLIP CHIP	FOCAL
DIGITAL	COMPUTER LAB

tab index

FOREWORD

This handbook is a collection of documents and sections of documents taken from the DECsystem-10 SOFTWARE NOTEBOOKS (DEC-10-SYZB-D). It is intended to be used by experienced programmers interested in writing and operating assembly-language programs on the DECsystem-10. The material in this handbook is aimed at providing the information needed for user-mode programming.

Most documents in this handbook are reprinted without change from the DECsystem-10 Software Notebooks. However, the first document in the handbook, the System Reference Manual, is an excerpt from the System Reference Manual in the Notebook set. This excerpt contains only the user-mode programming information needed by most assembly language programmers, and does not cover the documentation related to the various peripheral devices. If additional information is required, the reader is referred to the complete System Reference Manual in the Software Notebooks. All DECsystem-10 installations have two copies of the notebook set for reference.

The documents contained in this handbook reflect the following hardware and versions of the software:

System Reference Manual – KA10 and KI10 processors
MACRO – Version 47
Monitor Calls – 5.06 release
DDT – Version 34
LINK-10 – Version 1
CREF – Version 47
FILCOM – Version 20
FUDGE2 – Version 15
GLOB – Version 5A

The Assembly Language Handbook is one in the set of DECsystem-10 handbooks. The other handbooks comprising this series are:

(1) the COBOL Language Handbook.

(2) the Mathematical Languages Handbook, which covers FORTRAN, BASIC and ALGOL.

(3) the DECsystem-10 Users Handbook, which includes an introductory section, the operating system commands, TECO, and PIP.

In addition to the above-mentioned handbooks, the following documentation is also available:

(1) the COBOL Users Guide, which is aimed at COBOL users who wish to become familiar with COBOL on the DECsystem-10.

(2) the System Reference Card, which includes the word formats, instructions, and conversion tables for the DECsystem-10.

(3) the Operating System Commands Reference Card, which describes the commands, along with their formats, that are a part of the operating system.

(4) the Monitor Calls Reference Card, which covers the programmed operators (UUOs), and their formats, that can be used with the monitor.

(5) the BASIC Language Reference Card, which includes the statements, intrinsic functions, and edit and control commands of the DECsystem-10 BASIC Language.

The handbooks, Users Guide, and reference cards may be ordered from:

Software Distribution Center
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

DECsystem-10

System Reference Manual

ORDER NO. DEC-10-HGAD-D FROM PROGRAM LIBRARY, MAYNARD, MASSACHUSETTS PRICE $5.00

DIRECT COMMENTS CONCERNING THIS MANUAL TO SOFTWARE QUALITY CONTROL, MAYNARD, MASSACHUSETTS

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

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

December 1971

Copyright © 1968, 1969, 1971
by Digital Equipment Corporation

First edition, May 1968
Three printings

Second edition, December 1971

This edition has been expanded to provide system reference information for a DECsystem-10 with KA10 or KI10 central processors. The KI10 material has been incorporated into the text throughout.

Manufactured in the United States of America

Contents

1.	INTRODUCTION	5
1.1	Number System	11
	Floating point arithmetic	12
1.2	Instruction Format	14
	Effective address calculation	15
1.3	Memory	16
	KI10 memory allocation	18
	KA10 memory allocation	18
1.4	Programming Conventions	19
2.	CENTRAL PROCESSOR	23
2.1	Half Word Data Transmission	24
2.2	Full Word Data Transmission	31
	Move instructions	32
	Pushdown list	34
2.3	Byte Manipulation	37
2.4	Logic	39
	Shift and rotate	46
2.5	Fixed Point Arithmetic	48
	Arithmetic shifting	52
2.6	Floating Point Arithmetic	53
	Scaling	55
	Number conversion	56
	Single precision with rounding	58
	Single precision without rounding	60
	Double precision operations	64
2.7	Arithmetic Testing	67
2.8	Logical Testing and Modification	73
2.9	Program Control	80
2.10	Unimplemented Operations	91
2.11	Programming Examples	93
	Double precision floating point	95

2.12	Input-Output	96
	Readin mode	101
	Console-program communications	102
2.13	Priority Interrupt	103
2.14	Trapping and Processor Conditions	111
	Overflow trapping	111
	KI10 processor conditions	112
	KA10 processor conditions	115
2.15	KI10 Modes	117
	Paging	118
	Page failure	122
	Monitor programming	124
	Executive XCT	127
2.16	KA10 Modes	129
	User programming	131
	Monitor programming	131
2.17	Real Time Clock DK10	132
2.18	KA10 Operation	135
	Indicators	136
	Operating keys	138
	Operating switches	140
2.19	KI10 Operation *(Not available at this time)*	
	APPENDICES	
A	Instruction and Device Mnemonics	147
	Numeric listing	149
	Alphabetic listing	152
	Device mnemonics	156
	Algebraic representation	157
B	In-out Codes	167
	Teletype code	168
	Card codes	172
C	Timing	175
	KA10 timing	176
	KI10 timing *(Not available at this time)*	
D	KA10 Algorithms	179
	Fixed point algorithms	180
	Floating point algorithms	185
E	Processor Compatibility *(Not available at this time)*	
F	Indicator Panels *(Available in the System Reference Manual published as a part of the DECsystem-10 Software Notebooks.)*	
G	Bit Assignments	191

1

Introduction

The DECsystem-10 is a general purpose, stored program computing system that includes at least one PDP-10 central processor, a memory, and a variety of peripheral equipment such as paper tape reader and punch, teletypewriter, card reader and punch, line printer, DECtape, magnetic tape, disk, drum, display and data communications equipment. Each central processor is the control unit for an entire large-scale subsystem, in which it is connected by an in-out bus to its own peripheral equipment and by a memory bus to one or more memory units in a main memory, some of whose units may be shared by several processors. Within the subsystem the central processor governs all peripheral equipment, sequences the program, and performs all arithmetic, logical and data handling operations. Besides central processors, there are also 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 to the in-out bus of some central processor, to which it appears as an in-out device; the direct-access processor is also connected to memory by its own memory bus, and to its peripheral equipment by a device bus. The DECsystem-10 may also contain peripheral subsystems, such as for data communications, which are themselves based on small computers; such a subsystem in toto is connected to a PDP-10 in-out bus and is treated by the PDP-10 as a peripheral device. Unless otherwise specified, the words "processor" and "central processor" refer to the large-scale PDP-10 central processor, and "in-out bus" refers to the bus from the central processor to its peripheral equipment. A direct-access processor and the bus to its peripheral equipment are all always referred to by their names, *eg* the DF10 data channel and its channel bus (often a direct-access processor and device control are a single unit).

Confusion could result only in a chapter dealing with a small-computer subsystem. Here the small processor is usually referred to by its name (PDP-8, PDP-11) and the words "computer" and "memory" refer to the small computer. To differentiate, the PDP-10 is referred to by its name or as the "DECsystem-10 central processor", and the large scale memory connected to the PDP-10 memory bus is referred to as "DECsystem-10 main memory".

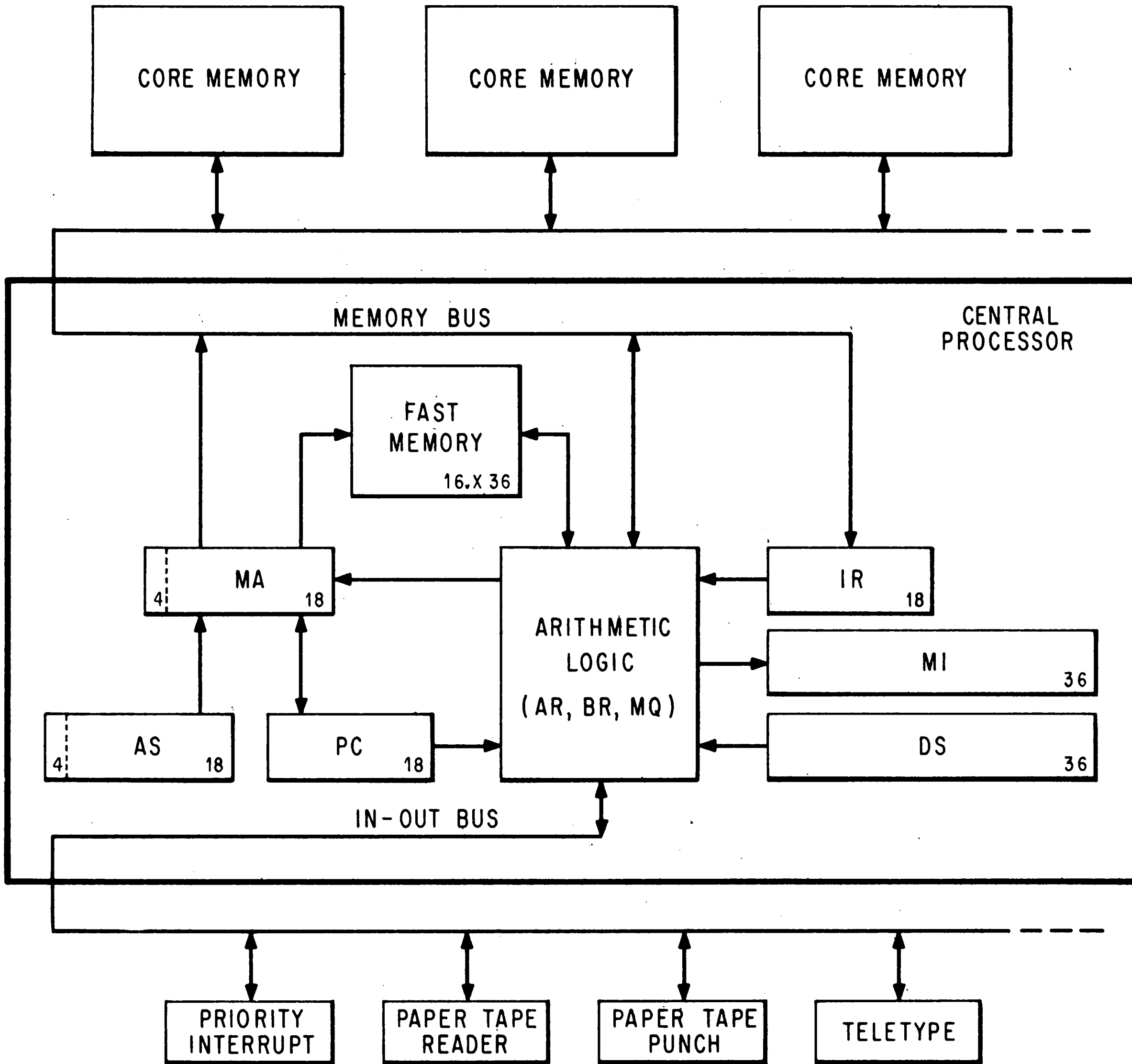
At present there are two types of PDP-10 central processors, the KA10 and the KI10. The latter is faster and more powerful, having a somewhat larger instruction repertoire including double precision floating point. Both processors handle words of thirty-six bits, which are stored in a memory whose maximum capacity depends upon the addressing capability of the processor. Internally both processors use 18-bit addresses and can thus reference 262,144 word locations in memory. This is the total addressing capability of the KA10, but in the KI10 it is only the virtual address space available to a single program. Paging hardware supplies four additional address bits to map pages in the program virtual address space into pages anywhere in a physical memory that is sixteen times as large. Thus for a number of different programs, the processor actually has access to a

physical memory with a capacity of 4,194,304 words. Storage in memory is usually in the form of 37-bit words, the extra bit producing odd parity for the word. 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 handle the words. The processor can handle half words, wherein the left half comprises bits 0–17, the right half, bits 18–35. There is also hardware for byte manipulation – a byte is any contiguous set of bits within a word. KA10 registers that hold addresses have eighteen bits, numbered 18–35 according to the position of an address in a word. KI10 internal address registers have eighteen bits, but a register that must supply a complete address to physical memory has twenty-two bits (numbered 14–35). Words are used either as computer instructions in the program, as addresses, or as operands (data for the program).

Of the internal registers shown in the illustration on the next page, only PC, the 18-bit program counter, is directly relevant to the programmer. The processor performs a program by executing instructions retrieved from the locations addressed by PC. At the beginning of each instruction PC is incremented by one so that it normally contains an address one greater than the location of the current instruction. 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. Also of importance to the programmer are the sense switches and the 36-bit data switch register DS on the processor console: through these switches the program can read information supplied by the operator. The processor also contains flags that detect various types of errors, including several types of overflow in arithmetic and pushdown operations, and provide other information of interest to the programmer.

The processor has other registers but the programmer is not usually concerned with them except when manually stepping through a program to debug it. 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 through AS the operator can supply a starting address for the program. Through the memory indicators MI the program can display data for the operator. The instruction register IR contains the left half of the current instruction word, *ie* all but the address part. The memory address register MA supplies the address for every memory access. The heart of the processor is the arithmetic logic, principally the 36-bit arithmetic register AR. This register takes part in all arithmetic, logical and data handling operations; all data transfers to and from memory, peripheral equipment and console are made via AR. Associated with AR are an extremely fast full adder, a buffer register BR that holds a second operand in many arithmetic and logical instructions, a multiplier-quotient register MQ that serves primarily as an extension of AR for handling double length operands, and smaller registers that handle floating point exponents and control shift operations and byte manipulation. In the KI10, AR and the adder each have a 28-bit left extension for handling double precision floating point numbers.

From the point of view of the programmer however the arithmetic logic can be regarded as a black box. It performs almost all of the operations



DECSYSTEM-10 SIMPLIFIED

necessary for the execution of a program, but it never retains any information from one instruction to the next. Computations performed in the black box either affect control elements such as PC and the flags, or produce results that are always sent to memory and must be retrieved by the processor if they are to be used as operands in other instructions.

An instruction word has only one 18-bit address field for addressing any location throughout all of the virtual address space. But 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 or 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 the 18-bit memory address (a zero index register address specifies no indexing). Although all computations on both operands and addresses are performed in the single arithmetic register AR, 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 memory locations are not actually in core memory, but are rather in a fast solid state 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, gaining faster execution for a short but oft-repeated subroutine.

The KI10 actually has four fast memory blocks, but only one of these is available to a program at any given time.

Besides the registers that enter into the regular execution of the program and its instructions, the processor has a priority interrupt system and equipment to facilitate time sharing. The interrupt system facilitates processor control of the peripheral equipment by means of a number of priority-ordered channels 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 channel or doing some special operation specified by the device (such as incrementing the contents of a memory location). Assignment of channels to devices is entirely under program control. One of the devices to which the program can assign a channel is the processor itself, allowing internal conditions such as overflow or a parity error to signal the program.

Time Sharing. Inherent in the basic 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. But above this fundamental level, the time share hardware provides for different modes of processor operation and establishes certain instruction restrictions 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 processor, but 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. 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, in other words with only one user, is equivalent to

The KI10 allows unrestricted in-out with a limited number of devices for special real time applications.

operation in executive mode (specifically kernel mode in the KI10).

The KA10 has the two modes discussed above, user and executive. It also has protection and relocation hardware to confine the user virtual address space within a particular range, and to relocate user memory references to the appropriate area in physical core. A user ordinarily has access to two separate core areas, one of which may be write-protected, *ie* the user cannot alter its contents.

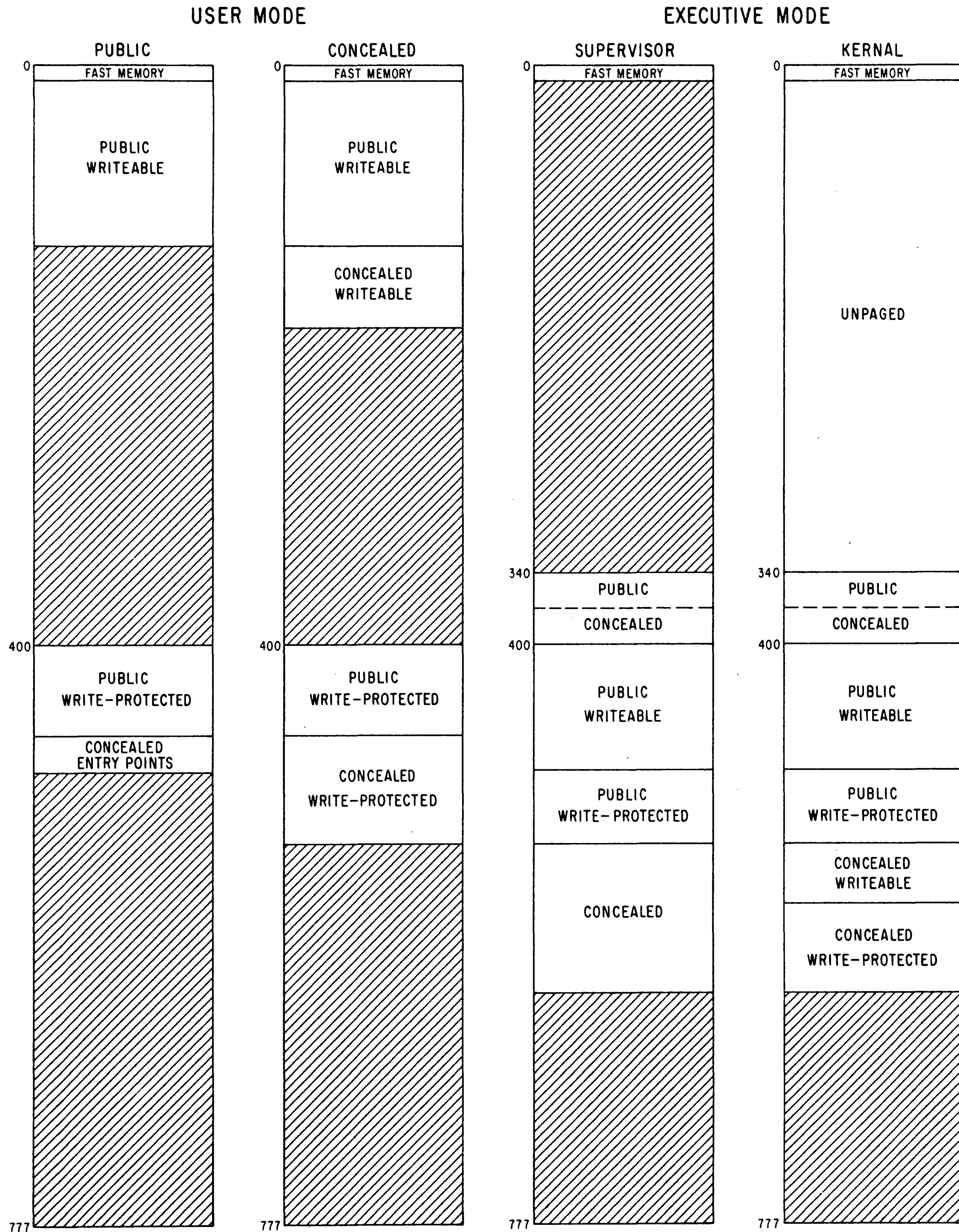
The KI10 has paging hardware for the mapping of pages from the limited virtual address space into pages anywhere in physical memory. A page map for each program specifies not only the correspondence from virtual address to physical address, but also whether an individual page is accessible or not, alterable or not, and public or concealed. Both user and executive modes are subdivided according to whether the program is running in a public area or a concealed area. Within user mode these are the public and concealed modes; within executive mode, the supervisor and kernel modes. A program in concealed mode can reference all of 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 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 and the program can even address some of memory directly (*ie* unpaged); in the paged address space, individual pages may be restricted as inaccessible or write-protected, but it is the kernel mode 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 mode 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 alter (even though the information is not write-protected from the kernel program). In either mode the Monitor automatically uses fast memory block 0 (the hardware requires this). The kernel program is responsible for assigning fast memory blocks to the various user programs: ordinarily blocks 2 and 3 are for special real time applications, and block 1 is assigned to all other users.

The illustration on the next page shows a typical layout of the virtual address space for the various modes. The space is 256K, made up of 512 pages numbered 0–777 octal. Any program can address locations 0–17 as these are in a fast memory block and are completely unrestricted (although the same addresses may be in different blocks for different programs). The public mode 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 mode 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.

The supervisor mode program is confined within the paged area of the address space, pages 340 and above. Part of the public area in this space may



TYPICAL VIRTUAL ADDRESS SPACE CONFIGURATION

be write-protected, but the program can read information in the concealed areas – it cannot alter any location in a concealed area whether that area is write-protected or not. 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. The kernel mode program can access all of the unpaged area without restriction and can reference all of the accessible paged area, both public and concealed, with the usual restriction that it cannot alter a write-protected area. As in the case of concealed user mode, fetching an instruction from a public area returns control to supervisor mode.

1.1 NUMBER SYSTEM

The program can interpret a data word as a 36-digit, unsigned binary number, or the left and right halves of a word can be taken as separate 18-bit numbers. The PDP–10 repertoire includes instructions that effectively 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 standard arithmetic instructions in the PDP–10 use twos complement, fixed point conventions 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 35 bits are the magnitude in ordinary binary notation. The negative of a number is obtained by taking its twos complement. If x is an n-digit binary number, its twos complement is $2^n - x$, and its ones complement is $(2^n - 1) - x$, or equivalently $(2^n - x) - 1$. Subtracting a number from $2^n - 1$ (*ie*, from all 1s) is equivalent to performing the logical complement, *ie* changing all 0s to 1s and all 1s to 0s. Therefore, to form the twos complement one takes the logical complement (usually referred to merely as the complement) of the entire word including the sign, and adds 1 to the result. In a negative number the sign bit is 1, and the remaining bits are the twos complement of the magnitude.

$+153_{10} = +231_8 =$ | 000 000 000 000 000 000 000 000 000 010 011 001 |
0 35

$-153_{10} = -231_8 =$ | 111 111 111 111 111 111 111 111 111 101 100 111 |
0 35

Zero is represented by a word containing all 0s. Complementing this number produces all 1s, and adding 1 to that produces all 0s again. Hence there is only one zero representation 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 in 1 (a

number all 1s represents −1). But since there are the same number of positive and negative numbers and zero is positive, there is one more negative number than there are nonzero positive numbers. This is the most negative number and it cannot be produced by negating any positive number (its octal representation is $400000\,000000_8$ and its magnitude is one greater than the largest positive number).

If ones complements were used for negatives one 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 one can read a negative number by attaching significance to the rightmost 1 and attaching significance to the 0s at 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.

Multiplication produces a double length product, and 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 null.

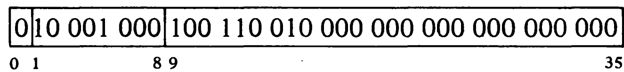
The computer does not keep track of a binary point — the programmer must adopt a point convention and shift the magnitude of the result to conform to the convention used. Two common conventions are to regard a number as an integer (binary point at the right) or as a proper fraction (binary point at the 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 0. The range for double length integers and proper fractions is thus -2^{70} to $2^{70} - 1$ and -1 to $1 - 2^{-70}$.

Floating Point Arithmetic. The KI10 has hardware for processing single and double precision floating point numbers; the KA10 can generally process only single precision numbers, although the hardware does include features that facilitate double precision arithmetic by software routines. 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. But the contents of bits 9−35 are now interpreted only as a binary fraction, and the contents of bits 1−8 are interpreted as an integral exponent in excess 128 (200_8) code. Exponents from −128 to +127 are therefore represented by the binary equivalents of 0 to 255 ($0{-}377_8$). Floating point zero and negatives are represented in exactly the same way as in fixed point: zero by a word containing all 0s, a negative by the twos complement. A negative number has a 1 for its sign and the twos complement of the fraction, but since every fraction must ordinarily contain a 1 unless the entire number is zero (see

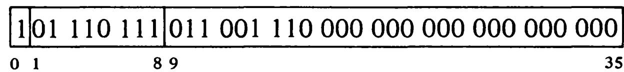
below), it has the ones complement of the exponent code in bits 1–8. Since the exponent is in excess 128 code, an actual exponent x is represented in a positive number by $x + 128$, in a negative number by $127 - x$. The programmer, however, need not be concerned with these representations as the hardware compensates automatically. *Eg,* 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.

$+153_{10} = +231_8 = +.462_8 \times 2^8 =$



0|10 001 000|100 110 010 000 000 000 000 000 000
0 1 8 9 35

$-153_{10} = -231_8 = -.462_8 \times 2^8 =$



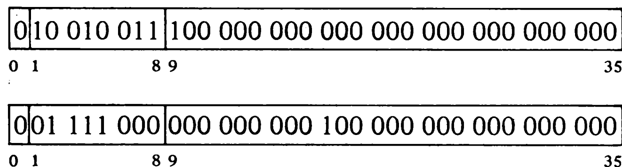
1|01 110 111|011 001 110 000 000 000 000 000 000
0 1 8 9 35

Except in special cases the floating point instructions assume that all nonzero operands are normalized, and they normalize a nonzero result. A floating point number is considered normalized if the magnitude of the fraction is greater than or equal to ½ and less than 1. The hardware may not give the correct result if the program supplies an operand that is not normalized or that has a zero fraction with a nonzero exponent.

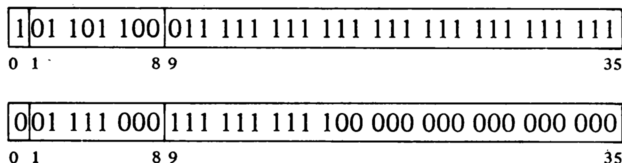
Single precision floating point numbers have a fractional range in magnitude of ½ to $1 - 2^{-27}$. Increasing the length of a number to two words does not significantly change the range but rather increases the precision; in any format the magnitude range of the fraction is ½ to 1 decreased by the value of the least significant bit. In all formats the exponent range is −128 to +127.

The precaution about truncation given for fixed point multiplication applies to most floating point operations as they produce extra length results; but here the programmer may request rounding, which automatically restores the high order part to twos complement form if it is negative. In single precision division the two words of the result are quotient and remainder, but in the other operations they form a double length number which is stored in two accumulators if the instruction is executed in "long" mode. (Long mode division uses a double length dividend.) A double length number used by the single precision instructions is in software double precision format. As such it 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. *Eg* the number $2^{18} + 2^{-18}$ has this two-word representation in software

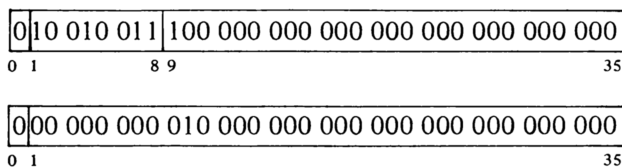
double precision format:



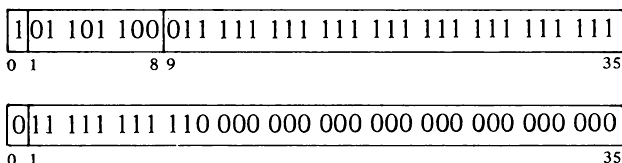
whereas its negative is



The double precision floating point instructions use a more straightforward double length format with greater precision than is allowed by the software format. For these instructions all operands and results are double length, and all instructions except division calculate a triple length answer, which is rounded to double length with the appropriate adjustment for a twos complement negative. In hardware 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. Bit 0 is ignored. The number used above as an example of software double precision format has this representation in hardware format:



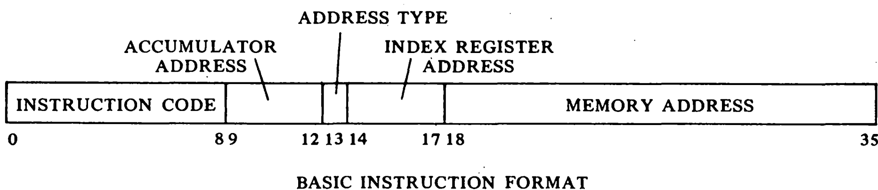
and its negative is



1.2 INSTRUCTION FORMAT

In all but the input-output instructions, the nine high order bits (0–8) specify the operation, and bits 9–12 usually address an accumulator but are sometimes used for special control purposes, such as addressing flags. The

rest of the instruction word usually 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. The instruction codes

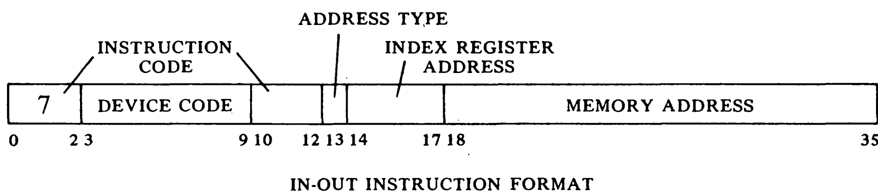


BASIC INSTRUCTION FORMAT

that are not assigned as specific instructions are performed by the processor as so-called "unimplemented operations".

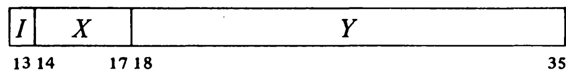
Among the unimplemented operations are some that are specified as "unimplemented user operations" or UUOs (a mnemonic that means nothing to the assembler). Half of these are for the local use of a program (LUUOs) and the other half are for communication with the Monitor (MUUOs). In general, unassigned codes act like MUUOs.

An input-output instruction is designated by three 1s in bits 0–2. 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.



IN-OUT INSTRUCTION FORMAT

Effective Address Calculation. 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 if the instruction must reference memory, bits 18–35 are the memory address Y. The effective address E of the instruction depends on the values of I, X and Y. If X is nonzero, the contents of index register X are added to Y to produce a modified address. If I is 0, addressing 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 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 bit 13; the 18-bit number calculated from the X and Y parts of this location is the effective address E.

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 instruc-

On the other hand, please note that this calculation is carried

out only for words indicated in the text as having the format shown. Do not assume that the procedure is used for any miscellaneous pointer simply because it happens to contain an address [*see page G2*].

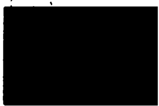
tion word is 0 and no memory reference is necessary, then Y is not an address. It may be a mask in some kind of test instruction, conditions to be sent to an in-out device, or part of it may be the number of places to shift in a shift or rotate instruction or the scale factor in a floating scale instruction. Even when modified by an index register, bits 18–35 do not contain an address when I is 0. 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 shift number, or effective scale factor. Many of the instructions that usually reference memory for an operand even 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 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 or conditions, or only part of it as in a shift number or scale factor.

1.3 MEMORY

The internal timing for each in-out device and each memory is entirely independent of the central processor. Because core memory readout is destructive, every word read must be written back in unless new information is to take its place. But the processor need never wait the entire cycle time. To read, it waits only until the information is available and then continues its operations while the memory performs the write portion of the cycle; to write, it waits only until the data is accepted, and the memory then performs an entire cycle to clear and write. To save time in an instruction that fetches an operand and then writes new data into the same location, the memory executes a read-modify-write cycle in which it 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 memory that may be needed by some other processor. Such instructions instead request separate read and write access. The KI10 further increases the speed of memory operation by overlapping memory cycles. *Eg* it can start one memory to read a word before receiving a word previously requested from a different memory.

Access times for the accumulator-index register locations are decreased considerably by substitution of a fast memory (contained in the processor) for the first sixteen core locations. Readout is nondestructive, so the fast memory has no basic cycle: the processor reads a word directly, but to write it must first clear the location and then load it.



The following table gives the characteristics of the various memories. 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 usually considerably shorter. The size of the MD10 can be increased in units of 32K up to 128K.

	Read Access	*Write Access*	*Cycle*	*Modify Completion*	*Size*
161 Core Memory	2.5	.49	4.7	2.69	16K
163 Core Memory	.94	.49	1.8	1.33	16K
164 Core Memory / MB10 Core Memory	.60*	.20*	1.65*	.97	16K
MA10 Core Memory	.61	.20	1.00	.57	16K
MD10 Core Memory	.83	.33	1.8	1.23	32–128K
ME10 Core Memory	.61	.20	1.00	.65	16K
KA10 Fast Memory	.21	.21			16
KI10 Fast Memory					16

*Add .1 in a multiprocessor system.

From the simple hardware addressing point of view, the entire memory is a set of contiguous 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 KA10 address is octal 777777, decimal 262,143; the largest KI10 address is 17777777, decimal 4,194,303. (Addresses are always in octal notation unless otherwise specified.) But the whole memory would usually be made up of a number of core memories of different capacities as listed above. Hence a given address actually selects a particular memory and a specific location within it. For a 16K memory with 18-bit addressing, the high order four address bits select the memory, the remaining fourteen bits address a single location in it; selecting a 32K memory takes three bits, leaving fifteen for the location. The times given above assume the addressed memory is idle when access is requested. To avoid waiting for a previously requested memory cycle to end, the program can make consecutive requests to different memories by taking instructions from one memory and data from another. All memories can be interleaved in pairs in such a way that consecutive addresses actually alternate between the two memories in the pair (thus increasing the probability that consecutive references are to different memories). Appropriate switch settings at the memories interchange the least significant address bits in the memory selection and location parts, so that in any two memories numbered n and $n + 1$ where n is even, all even addresses are locations in the first memory, all odd addresses are locations in the second. Hence memories 0 and 1 can be interleaved as can 6 and 7, but not 3 and 4 or 5 and 7. Some memories can be interleaved in contiguous groups of four, where the number of the first memory in the

group is divisible by four (*eg* memories 0–3 or 14–17). In this case all addresses ending in 0 or 4 reference the first memory in the group, all ending in 1 or 5 reference the second, and so forth.

In terms of the virtual address space (the addresses that can be specified within the limits of the instruction format) or the subset of it that is accessible to a user, the situation may be quite different. In the KA10 the user program has a continuous address space beginning at 0, or two continuous spaces beginning at 0 and 400000. In the KI10 the possible program address space is the set of all 18-bit addresses just as in the KA10, but which addresses a program can actually use depends entirely upon which of the 512 virtual pages (512 words per page) are accessible to it. For a so-called "small user", the accessible space must lie within the ranges 0–37777 and 400000–437777. In any event all programs have access to fast memory, whether as accumulators, index registers or ordinary memory references (*ie* addresses 0–17 are never restricted or relocated).

The kernel mode program can always address locations 0–337777 as these are unpaged. Virtual pages 340 and above are mapped.

KI10 Memory Allocation. The KI10 hardware defines the use of certain memory locations, but almost all of these are relative to pages whose physical location is specified by the Monitor. The only physical locations uniquely defined by the hardware are those in fast memory, whose addresses are the same for all programs: location 0 holds a pointer word during a bootstrap readin, 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 within a page specified by the Monitor for a particular user (including itself). For each user the Monitor keeps a process table, which must begin at location 0 of some page. The locations used by the hardware for the page map, traps, etc. of a given user are all in the first page of the table for that user. The parts of a user process table not used by the hardware may be used by the Monitor to keep accumulators (when the user is not running), a pushdown list that the Monitor uses for the job, and various user statistics such as running time, memory space, billing information, and job tables. The detailed configuration of the hardware-defined parts of the process tables (user and executive) is given in §2.15.

The Monitor keeps a user process table for each user program and one executive process table for itself for each KI10 processor. In the text, the phrase "the user process table" refers to the process table currently specified by the Monitor as the one for the user, even if that user is not currently running. The Monitor must also specify the whereabouts of the executive process table for the processor under consideration.

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

0	Holds a pointer word during a bootstrap readin
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

The initial control word address for the DF10 Data Channel must be less than 1000.

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.

1.4 PROGRAMMING CONVENTIONS

The computer has five instruction classes: data transmission, logical, arithmetic, program control and in-out. The instructions in the in-out class control the peripheral equipment, and also control the priority interrupt and time sharing, control and read the processor flags, and communicate with the console. The next chapter describes all instructions mentioned above, presents a general description of input-output, and describes the effects of the in-out instructions on the processor, priority interrupt and time share hardware. Effects of in-out instructions on particular peripheral devices are discussed with the devices.

The MACRO–10 assembly program 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), which are six bits in in-out instructions, otherwise nine or thirteen bits. *Eg* the mnemonic

MOVNS

assembles as 213000 000000, and

MOVNS 2570

The assembler translates every statement into a 36-bit word, placing 0s in all bits whose values are unspecified.

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.

The initial symbol @ preceding a memory address places a 1 in bit 13 tc 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 an 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 correspond to the octal digits in the word (000–774). Mnemonics are however available 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 addressing 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, *eg*

```
        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. *Eg*

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

and

```
        ADD     12,A
```

```
            .
            .
            .
A:          7256004
```

are equivalent.

Anything written at the right of a semicolon is commentary that explains the program but is not part of it.

2

Central Processor

This chapter describes all PDP–10 instructions but does not discuss the effects of those in-out instructions that address specific peripheral devices. In the description of each instruction, the mnemonic and name are at the top, 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 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; *eg* 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. Following the description is a table giving the mnemonics and octal codes (bits 0–8) for the various modes.

Letters representing modes are suffixes, which produce new mnemonics that are recognized as distinct symbols by the assembler.

In a description E refers to the effective address, half word operand, mask, 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 zero in the other, and is represented either as $0,E$ or $E,0$ depending upon whether E is in the right or left half.

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. If an instruction uses two accumulators, these have addresses A and $A+1$, where the second address is 0 if A is 17. In some cases an instruction uses an accumulator only if A is nonzero: a zero address in bits 9–12 specifies no accumulator.

The instructions are described in terms of their 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 core. 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. But for the details of programming in such special situations the reader must look elsewhere. In particular, §2.13 describes the priority interrupt, §2.14 discusses trapping, and §§2.15 and 2.16 describe the special effects and restrictions associated with the various machine modes in the KI10 and the KA10 respectively.

To minimize processor execution time the programmer should minimize the number of memory references and the number of shifts and other 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 that 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 more than the time saved).

No execution times are given with the instruction descriptions as the time may vary greatly depending upon circumstances. At the outset the time depends upon which processor performs the instruction, the mode the processor is in, and the speeds of the memories used for fetching the instruction, fetching its operands, and storing its results. Beyond this the time depends in many cases on the configuration of the operands and the number of iterative steps specified by the programmer as in a shift. Lastly 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, *ie* the sequence defined by the program counter (which 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 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 instruction executed or the hardware feature itself changes PC.

Some simple examples are included with the instruction descriptions, but more complex examples using a variety of instructions are given in §2.11.

2.1 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 determined 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.

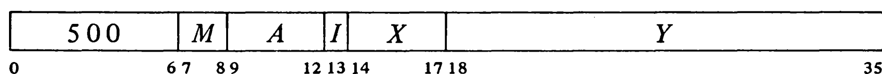
Operation	*Suffix*	*Effect on Other Half of Destination*
Do nothing		None
Zeros	Z	Places 0s in all bits of the other half
Ones	O	Places 1s in all bits of the other half
Extend	E	Places the sign (the leftmost bit) of the half word moved in all bits of the other half. This action extends a right half word 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*	AC
Memory	M	AC	*E*
Self	S	*E*	*E*, but full word result also goes to AC if *A* is nonzero

Note that selecting the left half of the source in immediate mode merely clears the selected half of the destination.

HLL Half Word Left to Left

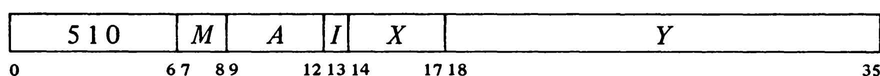


Move the left half of the source word specified by *M* to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination left half are lost.

HLLI merely clears AC left. If A is zero, HLLS is a no-op, otherwise it is equivalent to MOVE.

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

HLLZ Half Word Left to Left, Zeros



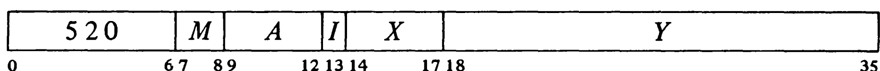
510	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

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.

HLLZI merely clears AC. If A is zero, HLLZS merely clears the right half of location E.

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

HLLO Half Word Left to Left, Ones



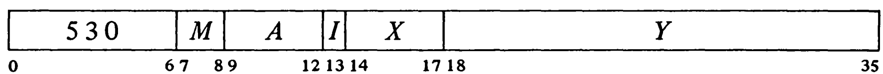
520	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

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.

HLLOI sets AC to all 0s in the left half, all 1s in the right.

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

HLLE Half Word Left to Left, Extend



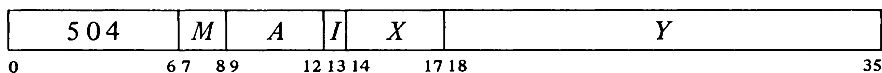
530	M	A	I	X	Y
0 6	7 8	9 12	13	14 17	18 35

Move the left half of the source word specified by M to the left half of the specified destination, and 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

HLLEI is equivalent to HLLZI (it merely clears AC).

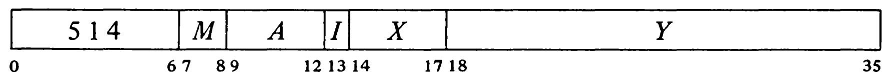
HRL Half Word Right to Left



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

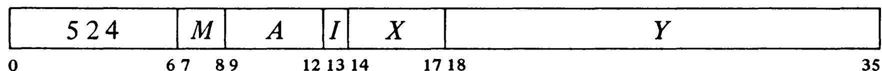


Move the right half of the source word specified by M to the left half of the specified destination, and clear the destination right half. The source is unaffected, the original contents of the destination are lost.

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

HRLZI loads the word E,0 into AC.

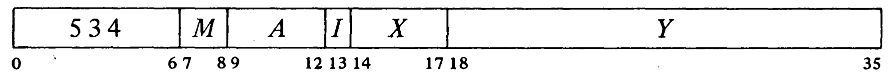
HRLO Half Word Right to Left, Ones



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

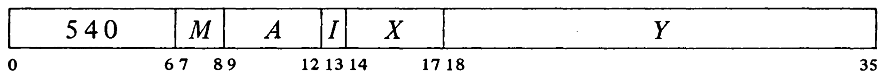
HRLE Half Word Right to Left, Extend



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

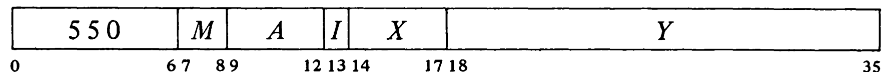


Move the right half of the source word specified by M to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

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

If A is zero, HRRS is a no-op; otherwise it is equivalent to MOVE.

HRRZ Half Word Right to Right, Zeros



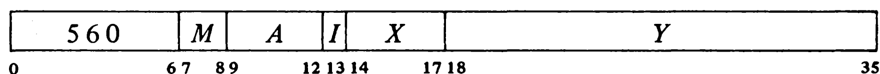
Move the right half of the source word specified by M to the right half of the

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

HRRZI loads the word $0,E$ into AC. If A is zero, HRRZS merely clears the left half of location E.

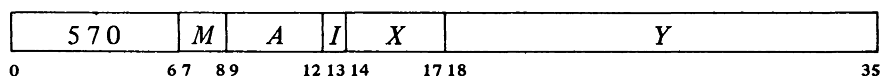
HRRO Half Word Right to Right, Ones



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

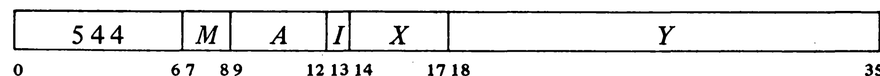
HRRE Half Word Right to Right, Extend



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



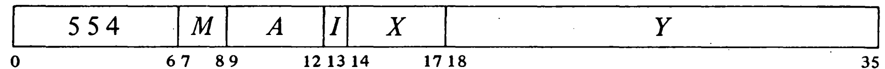
Move the left half of the source word specified by M to the right half of the

specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

HLRI merely clears AC right.

HLR	Half Left to Right	544
HLRI	Half Left to Right Immediate	545
HLRM	Half Left to Right Memory	546
HLRS	Half Left to Right Self	547

HLRZ Half Word Left to Right, Zeros

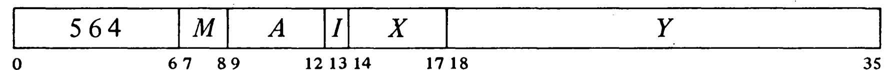


Move the left half of the source word specified by M to the right half of the specified destination, and clear the destination left half. The source is unaffected, the original contents of the destination are lost.

HLRZI merely clears AC and is thus equivalent to HLLZI.

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

HLRO Half Word Left to Right, Ones

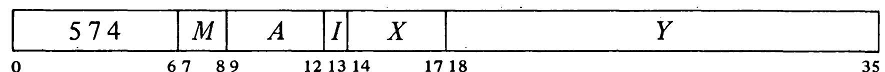


Move the left half of the source word specified by M to the right half of the specified destination, and set the destination left half to all 1s. The source is unaffected, the original contents of the destination are lost.

HLROI sets AC to all 1s in the left half, all 0s in the right.

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

HLRE Half Word Left to Right, Extend



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

HLREI is equivalent to HLRZI (it merely clears AC).

EXAMPLES. 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. *Eg* this pair of instructions loads the 18-bit numbers *M* and *N* into the left and right halves respectively of an accumulator that is addressed symbolically as XR.

```
HRLZI   XR,M
HRRI    XR,N
```

Of course the source program must somewhere define the value of the symbol XR as an octal number between 1 and 17.

Suppose that at some point we wish to use the two halves of XR independently as operands (taken as 18-bit positive numbers) for computations. We can begin by moving XR left to the right half of another accumulator AC and leaving the contents of XR right alone in XR.

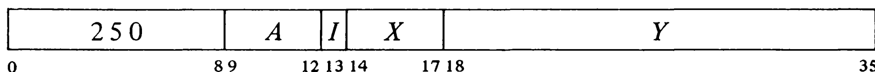
```
HLRZM   XR,AC
HLLI    XR,          ;Clear XR left
```

It is not necessary to clear the other half of XR when loading the first half word. But any instruction that modifies the other half is faster than the corresponding instruction that does not, as 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.)

2.2 FULL WORD DATA TRANSMISSION

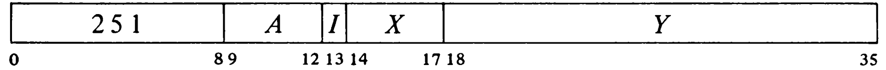
These are the instructions whose basic purpose is to move one or more full words of data from one place to another, usually from an accumulator to a memory location or vice versa. In a few cases instructions may perform minor arithmetic operations, such as forming the negative or the magnitude of the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

EXCH Exchange



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

BLT **Block Transfer**



Beginning at the location addressed by AC left, move words to another area of memory beginning at the location addressed by AC right. Continue until a word is moved to location E. The total number of words in the block is thus $E - AC_R + 1$.

Caution

Priority interrupts are allowed during the execution of this instruction, following the processing of each word. If an interrupt occurs, 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, unless the interrupt system is inactive, A and X must not address the same register as this would produce a different effective address calculation upon resumption should an interrupt occur; and the program must not attempt to load an accumulator addressed either by A or X unless it is the final location being loaded. Furthermore, the program cannot assume that AC is the same after the BLT as it was before.

Examples. This pair of instructions loads the accumulators from memory locations 2000–2017.

```
HRLZI   17,2000      ;Put 2000 000000 in AC 17
BLT     17,17
```

But to transfer the block in the opposite direction requires that one accumulator first be made available to the BLT:

```
MOVEM   17,2017      ;Move AC 17 to 2017 in memory
MOVEI   17,2000      ;Move the number 2000 to AC 17
BLT     17,2016
```

If at the time the accumulators were loaded the program had placed in location 2017 the control word necessary for storing them back in the same block (2000), the three instructions above could be replaced by

```
EXCH    17,2017
BLT     17,2016
```

Move Instructions

Besides the move instructions for single words there are also

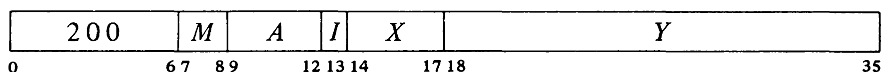
Each of these instructions moves a single word, which may be changed in the process (*eg* its two halves may be swapped). There are four instructions,

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 nonzero

four transmission instructions that handle double length operands (operands of two adjacent words). These are available, however, only in the KI10; and since they are principally for use in hardware double precision floating point operations, they are described with the floating point instructions in §2.6

MOVE **Move**

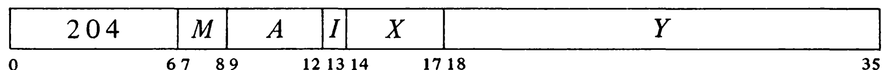


Move one word from the source to the destination specified by *M*. The source is unaffected, the original contents of the destination are lost.

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

MOVEI loads the word 0,*E* into AC and is thus equivalent to HRRZI. If *A* is zero, MOVES is a no-op; otherwise it is equivalent to MOVE.

MOVS **Move Swapped**

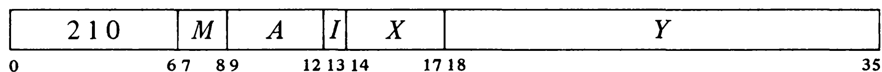


Interchange the left and right halves of the word from the source specified by *M* and move it to the specified destination. The source is unaffected, the original contents of the destination are lost.

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

Swapping halves in immediate mode loads the word *E*,0 into AC. MOVSI is thus equivalent to HRLZI.

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

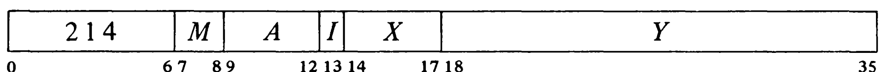
In the KI10 a move executed as an interrupt instruction can set no flags.

Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} 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. Setting Overflow also sets the Trap 1 flag in the KI10.

MOVNI loads AC with the negative of the word $0,E$ and can set no flags.

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

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 Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} sets the flags, but this is not a normalized number.) The source is unaffected, the original contents of the destination are lost. Setting Overflow also sets the Trap 1 flag in the KI10.

In the KI10 a move executed as an interrupt instruction can set no flags.

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

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

An example at the end of the preceding section demonstrates the use of a pair of immediate-mode half word transfers to load an address and a control count into an accumulator. The same result can be attained by a single move instruction. This saves time but still requires two locations. *Eg* if the number 200 001400 is stored in location M, the instruction

MOVE AC,M

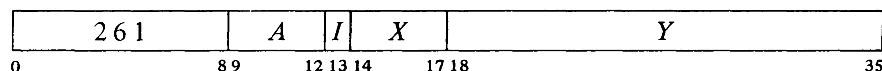
loads 200 into AC left and 1400 into AC right. If the same word, or its negative, or with its halves swapped, must be loaded on several occasions, then both time and space can be saved as each transfer requires only a single move instruction that references M.

Pushdown List

These two instructions insert and remove full words in a pushdown list. The address of the top item in the list is kept in the right half of a pointer in AC,

and the program can keep a control count in the left half. There are also two subroutine-calling instructions that utilize a pushdown list of jump addresses [§2.9].

PUSH **Push Down**

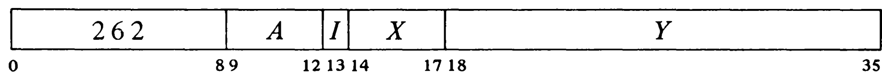


Add one to each half of AC, then move the contents of location E to the location now addressed by AC right. If the addition causes the count in AC left to reach zero, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. The contents of E are unaffected, the original contents of the location added to the list are lost.

Note: The KA10 increments the two halves of AC by adding 1000001_8 to the entire register. In the KI10 the two halves are handled independently.

In the KI10 a PUSH executed as an interrupt instruction cannot set Trap 2.

POP **Pop Up**



Move the contents of the location addressed by AC right to location E, then subtract one from each half of AC. If the subtraction causes the count in AC left to reach -1, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. The original contents of E are lost.

Because of the order in which the operands are stored, the instruction POP AC,AC would load the contents of the location addressed by AC right into AC on top of the pushdown count, destroying it.

Note: The KA10 decrements the two halves of AC by subtracting $1\,000001_8$ from the entire register. In the KI10 the two halves are handled independently.

In the KI10 a POP executed as an interrupt instruction cannot set Trap 2.

In the KA10, incrementing and decrementing both halves of AC together is effected by adding and subtracting $1\,000001_8$. Hence a count of -2 in AC left is increased to zero if $2^{18}-1$ is incremented in AC right, and conversely, 1 in AC left is decreased to -1 if zero is decremented in AC right.

A 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 pushdown list. We can deposit data in *a, b, c, d,* then read

d, then write in *d* and *e*, then read *e*, *d*, *c*, etc.

Note that by trapping or checking overflow and 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 words 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.

Pushdown storage is very convenient for a program that can use data stored in this manner as the pointer is initialized only once and only one accumulator is required for the most complex pushdown operations. To initialize a 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
```

Of course the programmer must define BLIST elsewhere 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 pseudoinstruction

```
IOWD J,K
```

is replaced by a word containing $-J$ in the left half and $K - 1$ in the right. Elsewhere there would 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.

In the PDP-10 the pushdown list is kept in a random access core memory, so the restrictions on order of entry and removal of items actually apply only to the standard addressing by the pointer in pushdown instructions – other addressing methods can reference any item at any time. The most convenient way to do this is to use the right half of the pointer as an index register. To move the last entry to accumulator AC we need simply give

```
MOVE   AC,(P)
```

Of course 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 the additive factor. Thus we can retrieve the next to last item by giving

```
MOVE   AC,-1(P)
```

and so forth. Similarly

```
PUSH   P,-3(P)
```

adds the third to last item to the end of the list;

```
        POP     P,-2(P)
```

removes the last item and inserts it in place of the next to last item in the shortened list.

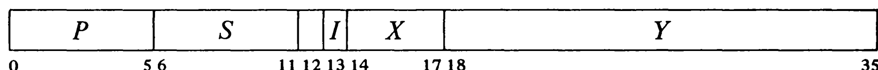
Note that E is calculated before the contents of P are changed.

2.3 BYTE MANIPULATION

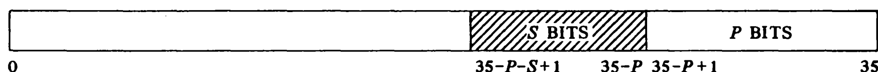
In a KA10 without byte manipulation hardware, all of the instructions presented in this section are trapped as unassigned codes [§2.10].

This set of five 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. The pointer has the format



where S is the size of the byte as a number of bits, and P is its position as the number of bits remaining at the right of the byte in the word (*eg* 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. Thus the pointer aims at a word whose format is



where the shaded area is the byte.

To facilitate processing a series of bytes, several of the byte instructions increment the pointer, *ie* modify it so that it points to the next byte position in a set of memory locations. Bytes are processed from left to right in a word, so 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 one 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.

Caution (KA10 only)

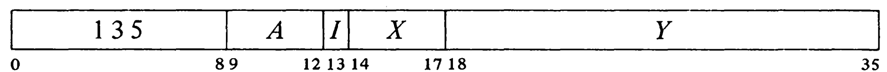
Do not allow Y to reach maximum value. The whole pointer is incre-

In the KI10, incrementing maximum Y produces a zero address without affecting X.

mented, 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 a priority interrupt should occur before the calculation is complete, the incremented X is used when the instruction is repeated.

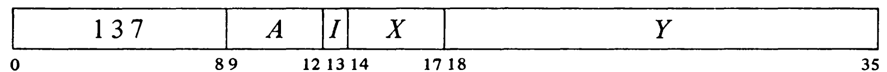
Among these five instructions one simply increments the pointer, the others load or deposit a byte with or without incrementing.

LDB Load Byte



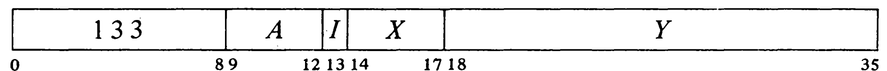
Retrieve a byte of S bits from the location and position specified by the pointer contained in location E, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected, the original contents of AC are lost.

DPB Deposit Byte



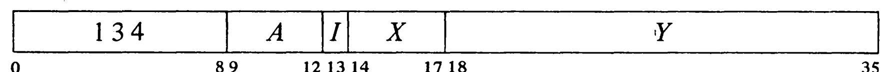
Deposit the right S bits of AC into the location and position specified by the pointer contained in location E. The original contents of the bits that receive the byte are lost, AC and the remaining bits of the deposit location are unaffected.

IBP Increment Byte Pointer



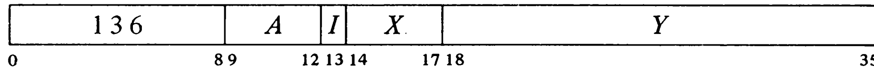
Increment the byte pointer in location E as explained above.

ILDB Increment Pointer and Load Byte



Increment the byte pointer in location E 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**



Increment the byte pointer in location E 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 them 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. At any time that the program might inspect the pointer during execution of a series of ILDBs or IDPBs, it points to the last byte processed (this may not be true when the pointer is tested from an interrupt routine [§2.13]).

Special Considerations. If S is greater than P and also greater than 36, incrementing produces a new P equal to $100-S$ rather than $36-S$. For $S>36$ the byte is at most the entire word; for $P\geqslant 36$ no byte is processed (loading merely clears AC). 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.

2.4 LOGIC

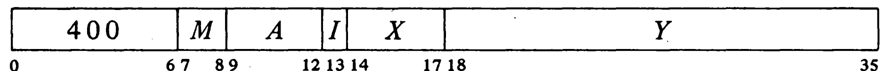
For logical operations the PDP-10 has instructions for shifting and rotating as well as for performing the complete set of sixteen Boolean functions of two variables (including those in which the result depends on only one or neither variable). The Boolean functions operate bitwise on full words, so 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 on page 2-23 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*	AC
Memory	M	*E*	*E*
Both	B	*E*	AC and *E*

SETZ Set to Zeros

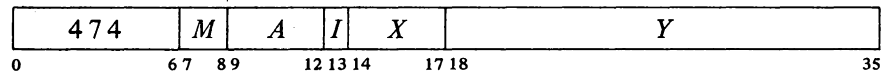


Change the contents of the destination specified by *M* to all 0s.

SETZ and SETZI are equivalent (both merely clear AC). MACRO also recognizes CLEAR, CLEARI, CLEARM and CLEARB as equivalent to the set-to-zeros mnemonics.

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

SETO Set to Ones

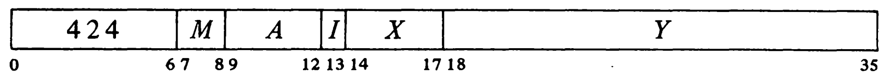


Change the contents of the destination specified by *M* to all 1s.

SETO and SETOI are equivalent.

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

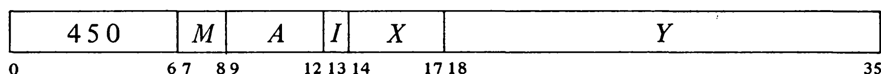
SETA Set to AC



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

SETA	Set to AC	424	SETA and SETAI are no-ops. SETAM and SETAB are both equivalent to MOVEM (all move AC to location E).
SETAI	Set to AC Immediate	425	
SETAM	Set to AC Memory	426	
SETAB	Set to AC Both	427	

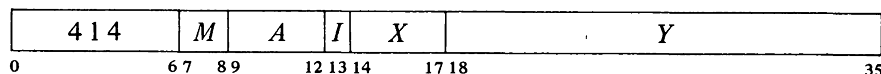
SETCA Set to Complement of AC



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

SETCA	Set to Complement of AC	450	SETCA and SETCAI are equivalent (both complement AC).
SETCAI	Set to Complement of AC Immediate	451	
SETCAM	Set to Complement of AC Memory	452	
SETCAB	Set to Complement of AC Both	453	

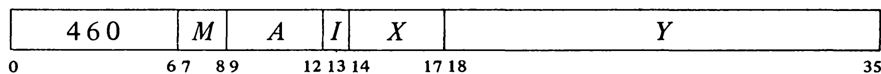
SETM Set to Memory



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

SETM	Set to Memory	414	SETM and SETMB are equivalent to MOVE. SETMI moves the word $0,E$ to AC and is thus equivalent to MOVEI. SETMM is a no-op that references memory.
SETMI	Set to Memory Immediate	415	
SETMM	Set to Memory Memory	416	
SETMB	Set to Memory Both	417	

SETCM Set to Complement of Memory

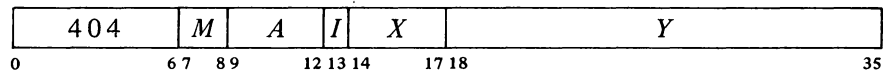


Change the contents of the destination specified by M to the complement of the specified operand.

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

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

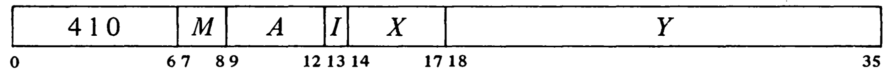
AND And with AC



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

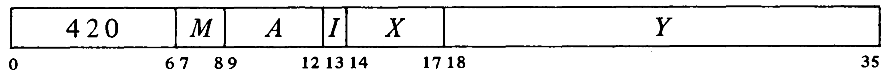
ANDCA And with Complement of AC



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

ANDCM And Complement of Memory with AC

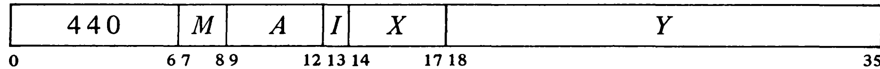


Change the contents of the destination specified by M to the AND function of the complement of the specified operand and AC.

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

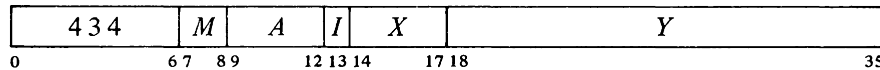
ANDCB And Complements of Both



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

IOR Inclusive Or with AC

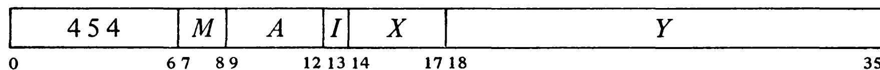


Change the contents of the destination specified by M to the inclusive OR function of the specified operand and AC.

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

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

ORCA Inclusive Or with Complement of AC



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

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

XOR **Exclusive Or with AC**

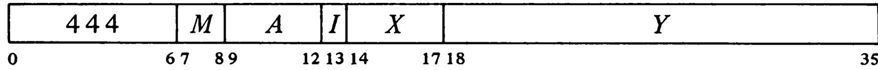
430	*M*	*A*	*I*	*X*	*Y*
0	6 7	8 9	12 13	14 17	18 35

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, *ie* by taking the exclusive OR of the remaining operand and the result.

EQV **Equivalence with AC**



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
EQVB	Equivalence to Both	447

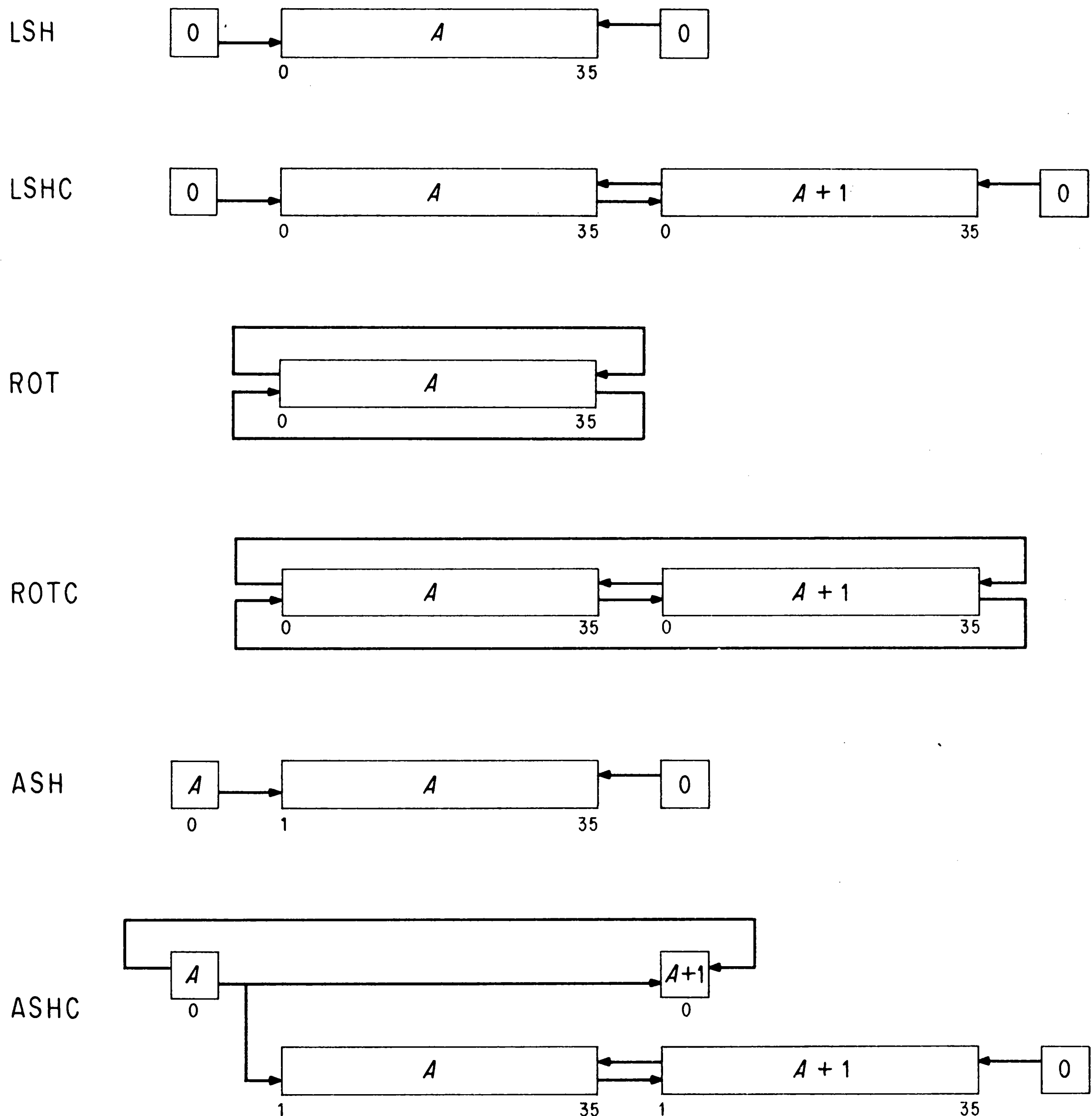
The original contents of the destination can be recovered except in EQVB, where both operands are replaced by the result. In the other three modes the replaced operand is restored by repeating the instruction in the same mode, *ie* by taking the equivalence function of the remaining operand and the result.

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	0	1	0	1
Mode Specified Operand	0	0	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

Shift and Rotate

The remaining logical instructions shift or rotate right or left the contents of AC or the contents of two accumulators, A and $A+1$ (mod 20_8), concatenated into a 72-bit register with A on the left. The illustration below shows the movement of information these instructions produce in the accu-

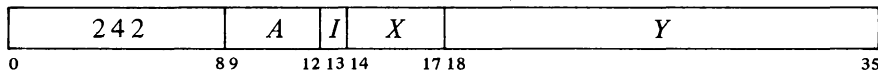


ACCUMULATOR BIT FLOW IN SHIFT AND ROTATE INSTRUCTIONS

mulators. In a (logical) shift the contents of a register are moved bit-to-bit with 0s brought in at the end being vacated; information shifted out at the other end is lost. [*For a discussion of arithmetic shifting see* §2.5.] In rotation the contents are moved cyclically such that information rotated out at one end is put in at the other.

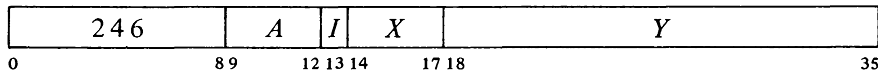
The number of places moved is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective shift E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the shift directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the shift. A positive E produces motion to the left, a negative E to the right. In the KA10, maximum movement is 255 places. The KI10 eliminates redundant movement of the operand by shifting E mod 72 places, for a maximum of 71.

LSH Logical Shift



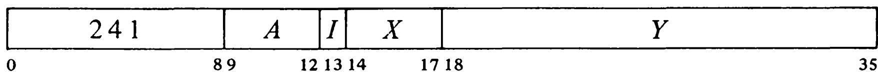
Shift AC the number of places specified by E. If E is positive, shift left bringing 0s into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right bringing 0s into bit 0; data shifted out of bit 35 is lost.

LSHC Logical Shift Combined



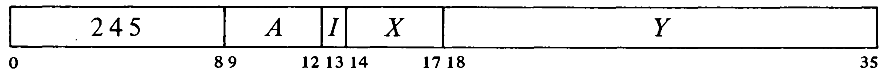
Concatenate accumulators A and $A+1$ with A on the left, and shift the 72-bit combination the number of places specified by E. If E is positive, shift left bringing 0s into bit 71 (bit 35 of AC $A+1$); bit 36 is shifted into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right bringing 0s into bit 0; bit 35 is shifted into bit 36; data shifted out of bit 71 is lost.

ROT Rotate



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**



Concatenate accumulators A and $A+1$ with A on the left, and rotate the 72-bit combination the number of places specified by E. If E is positive, rotate left; bit 0 is rotated into bit 71 (bit 35 of AC $A+1$) 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.

2.5 FIXED POINT ARITHMETIC

For fixed point arithmetic the PDP−10 has instructions for arithmetic shifting (which is essentially multiplication by a power of 2) as well as for performing addition, subtraction, multiplication and division of numbers in fixed point format [§1.1]. In such numbers the position of the binary point is arbitrary (the programmer may adopt any point convention). The add and subtract instructions involve only single length numbers, whereas multiply supplies a double length product, and divide uses a double length dividend. The high and low order words respectively of a double length fixed point number are in accumulators A and $A+1$ (mod 20_8), where the magnitude is the 70-bit string in bits 1−35 of the two words and the signs of the two are identical. 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 (of course they can be used for small fractions as well provided the programmer keeps track of the binary point). For convenience in the following, 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 actually 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 [§2.2], and the arithmetic test instructions that increment or decrement the test word [§2.7]. In these instructions an incorrect result is indicated – and the Overflow flag set – if the carries are different, *ie* if there is a carry into the sign but not out of it, or vice versa. 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 [§2.6], and loss of significant bits in left arithmetic shifting. In the KI10 any condition that sets Overflow also sets the Trap 1 flag.

Overflow is determined directly from the carries, not from the carry flags, as their states may reflect events in previous instructions.

In the KI10 an arithmetic instruction executed as an interrupt instruction can set no flags.

These flags can be read and controlled by certain program control instructions [§§2.9, 2.10]. In the KA10, Overflow is available as a processor

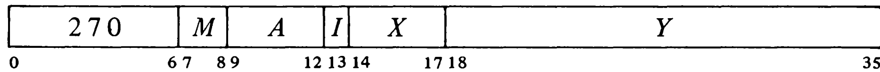
condition (via an in-out instruction) that can request a priority interrupt if enabled, whereas KI10 overflow is handled by trapping through the setting of Trap 1 [*both subjects are discussed in* §2.14]. The conditions detected can only set the arithmetic flags and the hardware does not clear them, so 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.

All but the shift 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*

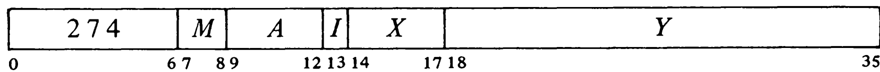
ADD Add



Add the operand specified by M to AC and place the result in the specified destination. If the sum is $\geqslant 2^{35}$ set 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 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 summands are negative, or their signs differ and their magnitudes are equal or 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

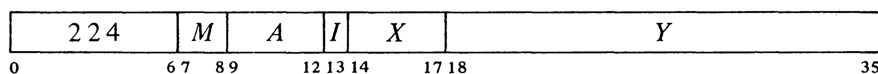


Subtract the operand specified by M from AC and place the result in the specified destination. If the difference is $\geqslant 2^{35}$ set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{35}. If the difference is $< -2^{35}$ set 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 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

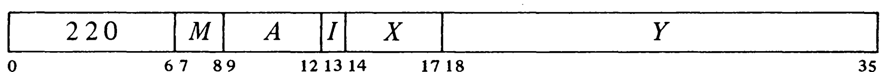
MUL **Multiply**



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 accumulator $A+1$. If both operands are -2^{35} set 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

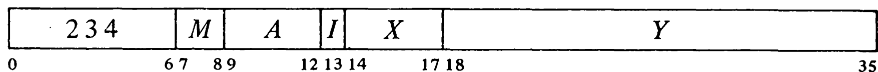
IMUL **Integer Multiply**



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 Overflow if the product is $\geqslant 2^{35}$ or $< -2^{35}$ (*ie* 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**

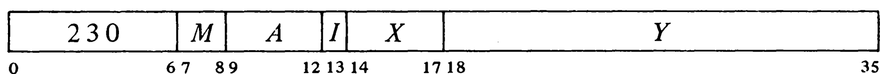


If the magnitude of the number in AC is greater than or equal to that of the

operand specified by M, set 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 double length number contained in accumulators A and $A+1$ 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 accumulator $A+1$.

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

IDIV **Integer Divide**



If the operand specified by M is zero, set 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 accumulator $A+1$.

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

EXAMPLE. The integer multiply and divide instructions are very useful for computations on addresses or character codes, or performing any integral operations in which the result is small enough to be accommodated in a single register.

As an example suppose we wish to determine the parity of the 8-bit character *abcdefgh*, where the letters represent the bits of the character. Assuming the character is right-justified in AC, we first duplicate it twice to the left producing

abc def gha bcd efg hab cde fgh

where the bits (in positions 12–35) are grouped corresponding to the octal digits in the word. Anding this with

001 001 001 001 001 001 001 001

retains only the least significant bit in each 3-bit set, so we can represent the result 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:

```
                    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
```

Since any digit is at most 1, there can be no carry out of any column with fewer than eight digits unless there is a carry into 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 (with the character right-justified in AC):

```
        IMULI   AC,200401
        AND     AC,ONES
        IMUL    AC,ONES
          .
          .
          .
ONES:   11111111
```

where the parity is indicated by AC bit 14. Of course, following the IMUL would be a test instruction to check the value of the bit.

Arithmetic Shifting

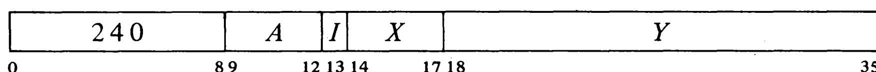
These two instructions produce an arithmetic shift right or left of the number in AC or the double length number in accumulators A and $A+1$. Shifting is the movement of the contents of a register bit-to-bit. The operation discussed here is similar to logical shifting [*see* §2.4 *and the illustration on page 2-24*], but in an arithmetic shift only the magnitude part is shifted – the sign is unaffected. In a double length number the 70-bit string made up of the magnitude parts of the two words is shifted, but the sign of the low order word is made equal to the sign of the high order word.

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.

The number of places shifted 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 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. In the KA10, maximum movement is 255 places. The KI10 eliminates redundant movement of the operand by shifting E mod 72 places, for a maximum of 71.

ASH **Arithmetic Shift**



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 Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative one). 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.

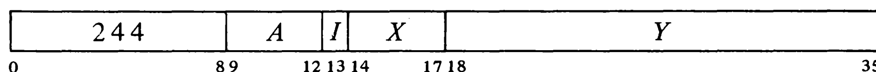
An arithmetic right shift truncates a negative result differently from IDIV if 1s are shifted out. The result of the shift is more negative by one than the quotient of IDIV.

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
```

This takes 5–6 μs as opposed to about 16 μs for IDIVI.

ASHC **Arithmetic Shift Combined**



Concatenate the magnitude portions of accumulators A and $A+1$ with A on the left, and shift the 70-bit combination in bits 1–35 and 37–71 the number of places specified by E. Do not shift AC bit 0, but make bit 0 of AC $A+1$ equal to it if at least one shift occurs (*ie* if E is nonzero). If E is positive, shift left bringing 0s into bit 71 (bit 35 of AC $A+1$); bit 37 (bit 1 of AC $A+1$) is shifted into bit 35; data shifted out of bit 1 is lost; set Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative one). 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.

2.6 FLOATING POINT ARITHMETIC

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)

In a KA10 without floating point hardware, all of the instructions presented in this section are trapped as unassigned codes [§2.10].

and negating double length numbers (software format) as well as performing addition, subtraction, multiplication and division of numbers in single precision floating point format. Moveover the KI10 has instructions for performing the four standard arithmetic operations on floating point numbers in hardware double precision format, for moving double precision numbers (with the option of taking the negative) between a pair of accumulators and a pair of memory locations, and for converting single precision numbers from fixed format to floating and vice versa. Except for the conversion instructions and the simple moves, all instructions treated here interpret all operands as floating point numbers in the formats given in § 1.1, and generate results in those formats. The reader is strongly advised to reread § 1.1 if he does not remember the formats in detail.

For the four standard arithmetic operations in single precision, the program can select whether or not the result shall be rounded. Rounding produces the greatest consistent precision using only single length operands. Instructions without rounding have a "long" mode, which supplies a two-word result for greater precision; the other modes 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" shall 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). Long mode is likely to retain information that would otherwise be lost, but in any given mode the significance of the result depends on the relative values of the operands. Even when both operands contain twenty-seven significant bits, a long addition may store two words that together contain only one significant bit. 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; long mode retains the remainder.

A subtraction involving two like-signed numbers whose exponents are equal and whose fractions differ only in the LSB gives a result containing only one bit of significance.

Among the floating point instructions available only in the KI10, those that convert between number types operate only on single words. The instruction that converts 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. The instructions for the four standard operations using double precision have no modes. In division the processor always calculates a two-word quotient that is normalized if the original operands are normalized, but rounding is not available. 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 limited normalization and then for rounding, which is automatic.

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 nonzero operands are normalized. Any condition that sets Overflow in the KI10 also sets the Trap 1 flag. These flags can be read and controlled by certain program control instructions [§§2.9, 2.10]. In the KA10, Overflow and Floating Overflow are available as processor conditions (via an in-out instruction) that can request a priority interrupt if enabled, whereas KI10 overflow is handled by trapping through the setting of Trap 1 [*both of these subjects are discussed in* §2.14]. 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.

In the KI10 an arithmetic instruction executed as an interrupt instruction can set no flags.

The floating point hardware functions at its best if given operands that are either normalized or zero, and except in special situations the hardware normalizes a nonzero result. An operand with a zero fraction and a nonzero exponent can give wild answers in additive operations because of extreme loss of significance; *eg* 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. This unnormalized number can produce an incorrect answer in any operation. Use of other unnormalized operands simply causes loss of significant bits, except in division where they can prevent its execution because they can satisfy a no-divide condition that is impossible for normalized numbers.

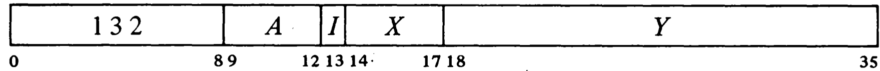
The processor normalizes the result by shifting the fraction and adjusting the exponent to compensate for the change in value. Each shift and accompanying exponent adjustment thus multiply the number both by 2 and by ½ simultaneously, leaving its value unchanged.

Note that 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 ½ gives a result of ¼. 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. A subtraction involving two like-signed 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.

Scaling

One floating point instruction is in a category by itself: it changes the exponent of a number without changing the significance of the fraction. In other words it multiplies the number by a power of 2, and is 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 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 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.

FSC **Floating Scale**



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.

This instruction can be used to float a fixed number with 27 or fewer significant bits. To float an 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 ($233_8 = 155_{10} = 128 + 27$).

NOTE

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

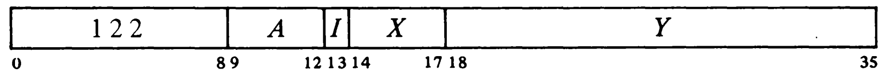
If the exponent after normalization is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

Number Conversion

In the KA10 these instructions are trapped as unassigned codes.

Although FSC can be used to float a fixed point number, the KI10 has 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.

FIX **Fix**



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.

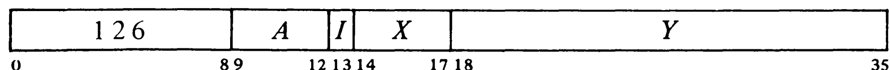
This overflow test checks for a value $\geqslant 2^{35}$ assuming the operand is normalized.

Otherwise replace the exponent X in the word from location E with bits equal to the sign of the fraction, and shift the (now fixed) extended fraction $N = X - 27$ places to the correct position for its order of magnitude with 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.

This is the standard Fortran truncation ("fixation"). For it, the processor drops the

Truncation produces the integer of largest magnitude less than or equal to the magnitude of the original number. *Eg* a number $> +1$ but $< +2$ becomes $+1$; a number < -1 but > -2 becomes -1.

FIXR **Fix and Round**



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 X in the word from location E with bits equal to the sign of the fraction, and shift the (now fixed) extended fraction $N = X - 27$ places to the correct position for its order of magnitude with 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.

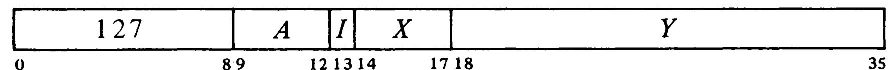
Rounding is in the positive direction: the magnitude of the integral part is increased by one if the fractional part is $\geqslant \frac{1}{2}$ in a positive number but $> \frac{1}{2}$ in a negative number. *Eg* +1.4 (decimal) is rounded to +1, whereas +1.5 and +1.6 become +2; but with negative numbers,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

fractional part in a positive number, but adds one to the integral part (as required by twos complement format) if any bits of significance are shifted out in a negative number.

This overflow test checks for a value $\geqslant 2^{35}$ assuming the operand is normalized.

This is the Algol standard for real to integer conversion. For it the processor adds one to the integral part if the fractional part is $\geqslant \frac{1}{2}$ in a positive number or (as required by twos complement format) is $\leqslant \frac{1}{2}$ in a negative number.

FLTR **Float and Round**



Shift the magnitude part of the fixed point integer from location E right eight places, insert the exponent 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 standard floating point arithmetic instructions [*see below*].

Since the largest fixed point magnitude (without considering sign) is $2^{35} - 1$, a floating point number with exponent greater than 35 (and assumed normalized) cannot be converted to fixed point. There is no limit in the opposite direction, but precision can be lost as floating point format provides fewer significant bits. A fixed integer greater than $2^{27} - 1$ cannot be represented exactly in floating point unless all its significant bits are clustered within a group of twenty-seven.

Single Precision with Rounding

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 $<$ ½LSB. Moreover an extra single-step renormalization occurs if the rounded word is no longer normalized.

There are four instructions that use only one-word operands and store a single-length rounded result. Rounding is away from zero: 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.

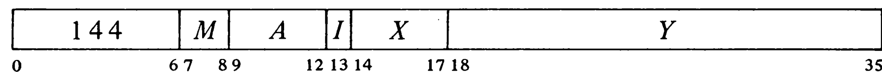
The rounding instructions have four modes that determine the source of the non-AC operand and the destination of the result. These modes are like those of logic and fixed point arithmetic, including an immediate mode that allows the instruction to carry an operand with it.

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 Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

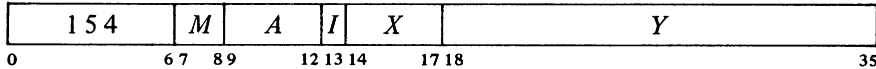
FADR **Floating Add and Round**



Floating add the operand specified by *M* to AC. 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

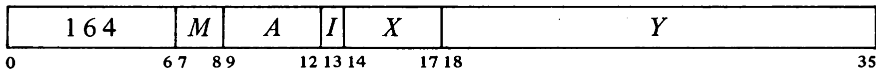
FSBR **Floating Subtract and Round**



Floating subtract the operand specified by M from AC. 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

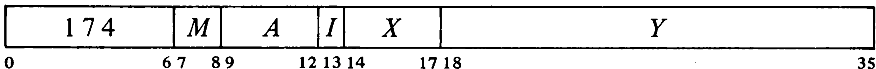
FMPR **Floating Multiply and Round**



Floating Multiply AC by the operand specified by M. 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**



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 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 if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If the division can be performed, floating divide AC 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, and 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

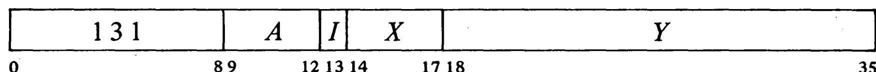
Single Precision without Rounding

Instructions that do not round are faster for processing floating point numbers with fractions containing fewer than 27 significant bits. On the other hand the long mode provides double precision (software format) or allows the programmer to use his own method of rounding. Besides the four usual arithmetic operations with normalization, there are two nonnormalizing instructions that facilitate software double precision arithmetic [§2.11 *gives examples of double precision floating point routines*]. These two instructions have no modes.

Usually the double length number is in two adjacent accumulators, and E equals $A+1$.

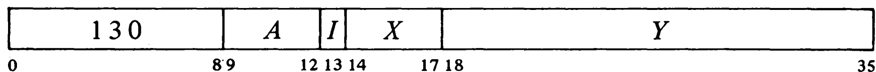
Note that this instruction can be used to negate numbers in software double precision format only, for the KI10 hardware double precision format, the program must use the double moves.

DFN **Double Floating Negate**



Negate the 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.

UFA **Unnormalized Floating Add**



Floating add the contents of location E to AC. If the double length fraction in the sum is zero, clear accumulator $A+1$. Otherwise normalize the sum only if the magnitude of its fractional part is $\geqslant 1$, and place the high order part of the result in AC $A+1$. The original contents of AC and E are unaffected.

NOTE

The result is placed in accumulator $A+1$. This is the only arithmetic instruction that stores the result in a second accumulator, leaving the original operands intact.

If the exponent of the sum following the one-step normalization is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one.

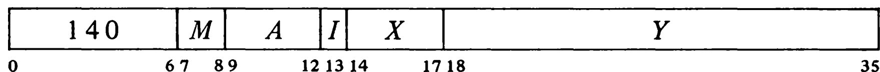
The exponent of the sum is equal to that of the larger summand unless addition of the fractions overflows, in which case it is greater by 1. Exponent overflow can occur only in the latter case.

The remaining single precision floating point instructions perform the four standard arithmetic operations with normalization but without rounding. All use AC and the contents of location E as operands and have four modes.

Mode	*Suffix*	*Effect*
Basic		High order word of result stored in AC.
Long	L	In addition, subtraction and multiplication, the two-word result (in the double length format described in §1.1) is stored in accumulators A and $A+1$. In division the dividend is the double length word in A and $A+1$; the single length quotient is stored in AC, the remainder in AC $A+1$.
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 Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < −128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

FAD Floating Add



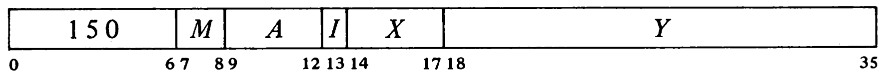
Floating add the contents of location E to AC. If the double length fraction in the sum is zero, clear the destination specified by M, clearing both accumulators in long mode. Otherwise normalize the double length sum bringing 0s into bit positions vacated at the right, test for exponent overflow or

underflow as described above, and place the high order word of the result in the specified destination.

In long mode if the exponent of the sum is $> 154\ (127+27)$ or $< -101\ (-128+27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the sum in bits 1–8, and the low order part of the fraction in bits 9–35.

FAD	Floating Add	140
FADL	Floating Add Long	141
FADM	Floating Add to Memory	142
FADB	Floating Add to Both	143

FSB Floating Subtract

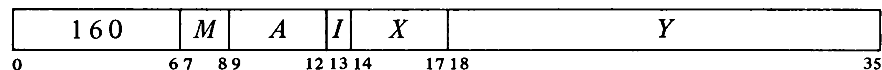


Floating subtract the contents of location E from AC. If the double length fraction in the difference is zero, clear the destination specified by M, clearing both accumulators in long mode. Otherwise normalize the double length difference bringing 0s into bit positions vacated at the right, test for exponent overflow or underflow as described above, and place the high order word of the result in the specified destination.

In long mode if the exponent of the difference is $> 154\ (127+27)$ or $< -101\ (-128+27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the difference in bits 1–8, and the low order part of the fraction in bits 9–35.

FSB	Floating Subtract	150
FSBL	Floating Subtract Long	151
FSBM	Floating Subtract to Memory	152
FSBB	Floating Subtract to Both	153

FMP Floating Multiply



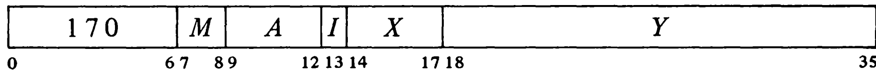
Floating multiply AC by the contents of location E. If the double length fraction in the product is zero, clear the destination specified by M, clearing both accumulators in long mode. Otherwise normalize the double length

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.

In long mode 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 $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the product in bits 1–8, and the low order part of the fraction in bits 9–35.

FMP	Floating Multiply	160
FMPL	Floating Multiply Long	161
FMPM	Floating Multiply to Memory	162
FMPB	Floating Multiply to Both	163

FDV Floating Divide



If the magnitude of the fraction in AC is greater than or equal to twice that of the fraction in location E, set 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 if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If division can be performed, floating divide the AC operand by the contents of location E. In long mode the AC operand (the dividend) is the double length number in accumulators A and $A+1$; in other modes it is the single word in AC. Calculate a quotient fraction of 27 bits. If the fraction is zero, clear the destination specified by M, clearing both accumulators in long mode if the double length dividend was zero. A quotient with a nonzero 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 the specified destination.

In long mode a nonzero unnormalized dividend whose entire high order fraction is zero produces a zero quotient. In this case the second AC receives rubbish.

In long mode 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 > 127 or < -128 or the fraction is zero, clear AC $A+1$. Otherwise place the floating point remainder (exponent and fraction) with the sign of the dividend in AC $A+1$.

FDV	Floating Divide	170
FDVL	Floating Divide Long	171
FDVM	Floating Divide to Memory	172
FDVB	Floating Divide to Both	173

Double Precision Operations

In the KA10 these instructions are trapped as unassigned codes.

Although double precision floating point arithmetic can be done by routines using the single precision instructions and the software double length format, the KI10 has instructions specifically for handling double length operands in the hardware double precision format described in § 1.1. Four of the instructions use two double length operands, perform the standard arithmetic operations, and store double length results. The other four instructions each move one double length operand between the accumulators and memory, either unchanged or negated.

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) just as they do for the double length operands used in some shift, rotate and single precision arithmetic instructions. The memory locations have addresses E and $E+1$ (mod 2^{18}), where the second address is 0 if E is 777777.

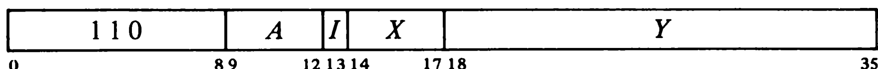
For the two instructions that simply move a pair of words without altering them, the format of those words is actually irrelevant. The other six instructions process each word pair as a double length number in the hardware floating point format. Hence they ignore bit 0 in the low order word of every operand and clear that bit in the result.

The four nonmove instructions perform the standard arithmetic operations. All use two double length operands in the hardware double precision format, one from the accumulators and one from memory. Addition and subtraction always normalize the result; in multiplication and division the result is guaranteed to be normalized only if the original operands are normalized. In all cases the result, rounded except in division, is placed in the accumulators. 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).

An arithmetic instruction executed as an interrupt instruction can set no flags.

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 Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one. Setting Overflow also sets the Trap 1 flag.

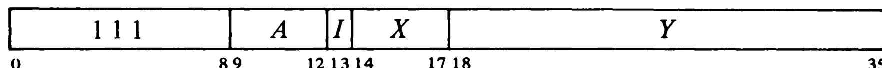
DFAD **Double Floating Add**



Floating add the operand of locations E and $E+1$ to the operand of accumulators A and $A+1$. If the high order 70 bits of the fraction in the

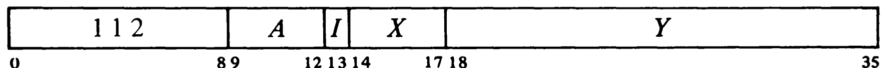
sum are zero, clear A and $A+1$. Otherwise normalize the triple length sum bringing 0s in at the right, round the high order double length part, test for exponent overflow or underflow as described above, and place the result in ACs A and $A+1$.

DFSB Double Floating Subtract



Floating subtract the operand of locations E and $E+1$ from the operand of accumulators A and $A+1$. If the high order 70 bits of the fraction in the difference are zero, clear A and $A+1$. 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 ACs A and $A+1$.

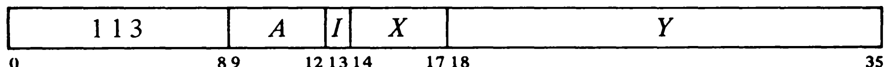
DFMP Double Floating Multiply



Floating multiply the operand of accumulators A and $A+1$ by the operand of locations E and $E+1$. If the high order 70 bits of the fraction in the product are zero, clear A and $A+1$. Otherwise, if there are any bits of significance among the high order 35, 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 ACs A and $A+1$.

The 35-bit shift can be done only if the original operands are unnormalized.

DFDV Double Floating Divide



If the magnitude of the fraction in the operand of accumulators A and $A+1$ is greater than or equal to twice that of the fraction in the operand of locations E and $E+1$, set Overflow, Floating Overflow, No Divide and Trap 1, and go immediately to the next instruction without affecting the original AC or memory operands in any way.

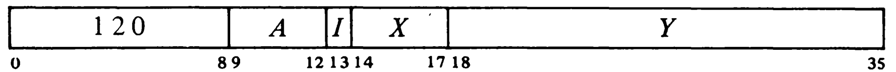
Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If the division can be performed, floating divide the AC operand by the memory operand, calculating a quotient fraction of 62 bits. If the fraction

A nonzero quotient is normalized if the original operands are normalized.

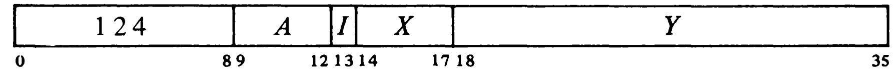
is zero, clear A and $A+1$. Otherwise test for exponent overflow or underflow as described above, and place the double length quotient part of the result in ACs A and $A+1$ (the remainder is lost).

DMOVE Double Move



Move the contents of locations E and $E+1$ respectively to accumulators A and $A+1$. The memory locations are unaffected, the original contents of the ACs are lost.

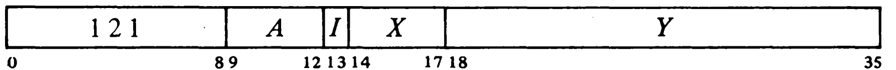
DMOVEM Double Move to Memory



Do not use the instruction DMOVEM AC,AC+1. At present the processor places AC in both AC+1 and AC+2, but this result is not guaranteed.

Move the contents of accumulators A and $A+1$ respectively to locations E and $E+1$. The ACs are unaffected, the original contents of the memory locations are lost.

DMOVN Double Move Negative

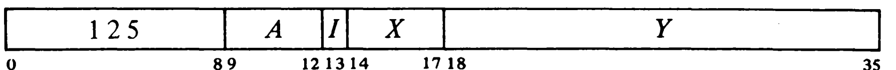


Note that these two instructions can be used to negate numbers in hardware double precision format only; for software double precision, the program must use DFN.

Note also that there is no overflow test, as negating a correctly formatted floating point number cannot cause overflow.

Negate the double length floating point number taken from locations E and $E+1$, and move it to accumulators A and $A+1$. The memory locations are unaffected, the original contents of the ACs are lost.

DMOVNM Double Move Negative to Memory



Do not use the instruction DMOVNM AC,AC+1. At pre-

Negate the double length floating point number taken from accumulators A and $A+1$, and move it to locations E and $E+1$. The ACs are unaffected, the original contents of the memory locations are lost.

Although the configuration of the operands is irrelevant in DMOVE and DMOVEM, none of the above instructions is available in the KA10. Therefore unless a program is actually doing floating point arithmetic in the hardware double precision format, it is recommended that the double moves not be used in KI10 programs so they will be compatible with the KA10. Simply to move a two-word operand unaltered requires two one-word moves. To negate a two-word operand that is actually in the hardware format requires a somewhat longer substitution; *eg* this sequence is equivalent to DMOVN AC,E.

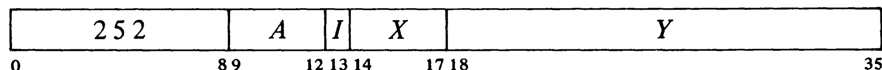
```
SETCM  AC,E             ;Take ones complement of high word
MOVN   AC+1,E+1         ;Take twos complement of low word
TDNN   AC+1,[377777777777]     ;If low part of fraction is
ADDI   AC,1             ;zero, change high word to twos com-
                        ;plement
```

sent the processor places the negative of AC (the complement, if AC+1 originally contains zero) into AC+1, and the negative of that into AC+2, but this result is not guaranteed.

2.7 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. Two of the instructions have no modes.

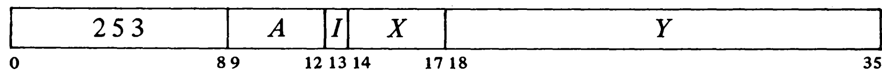
AOBJP Add One to Both Halves of AC and Jump if Positive



Add one to each half of AC and place the result back in AC. If the result is greater than or equal to zero (*ie* 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.

Note: The KA10 increments the two halves of AC by adding $1\ 000001_8$ to the entire register. In the KI10 the two halves are handled independently.

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



Add one to each half of AC and place the result back in AC. If the result is less than zero (*ie* 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.

Note: The KA10 increments the two halves of AC by adding $1\ 000001_8$ to the entire register. In the KI10 the two halves are handled independently.

In the KA10, incrementing both halves of AC together is effected by adding $1\ 000001_8$. A count of −2 in AC left is therefore increased to zero if $2^{18} - 1$ is incremented in AC right.

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 accounted for
```

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

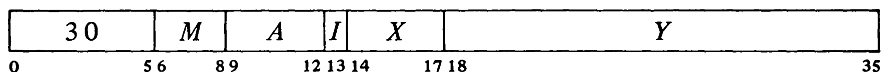
Mode	*Suffix*
Never	
Less	L
Equal	E
Less or Equal	LE
Always	A
Greater or Equal	GE
Not Equal	N
Greater	G

Instructions with one operand compare AC or the contents of location E with zero, those with two compare AC with E or the contents of location E. The processor always makes the comparison even though the result is used in only six of the modes. If the mnemonic has no suffix there is never any program control function, and the instruction may be a no-op; an A suffix produces an unconditional jump or skip – the action is always taken regardless of how the two quantities compare.

In the KI10 an arithmetic instruction executed as an interrupt instruction can set no flags.

The last four of these instructions perform arithmetic operations, which are checked for overflow. In the KI10 any condition that sets Overflow also sets the Trap 1 flag.

CAI Compare AC Immediate and Skip if Condition Satisfied

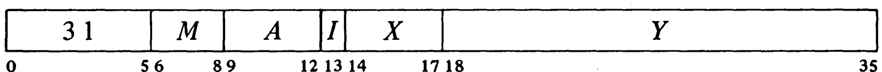


Compare AC with E (*ie* with the word $0, E$) and skip the next instruction in sequence if the condition specified by M is satisfied.

CAI	Compare AC Immediate but 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 but 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

CAI is a no-op.

CAM Compare AC with Memory and Skip if Condition Satisfied

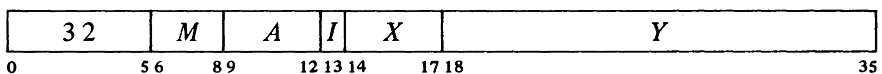


Compare AC with the contents of location E and skip the next instruction in sequence if the condition specified by M is satisfied. The pair of numbers compared may be either both fixed or both normalized floating point.

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

CAM is a no-op that references memory.

JUMP Jump if AC Condition Satisfied



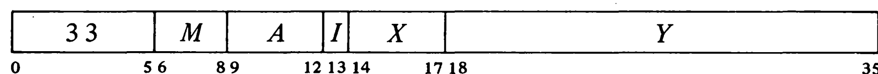
Compare AC (fixed or floating) with zero, and if the condition specified by

JUMP is a no-op (instruction code 320 has this mnemonic for symmetry).

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

SKIP Skip if Memory Condition Satisfied



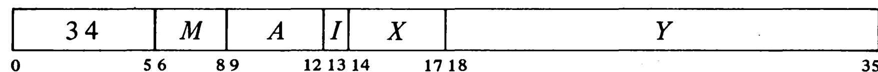
If *A* is zero, SKIP is a no-op; otherwise it is equivalent to MOVE. (Instruction code 330 has mnemonic SKIP for symmetry.)

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 nonzero also place the contents of location *E* in AC.

SKIP	Do Not Skip	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

SKIPA is a convenient way to load an accumulator and skip over an instruction upon entering a loop.

AOJ Add One to AC and Jump if Condition Satisfied

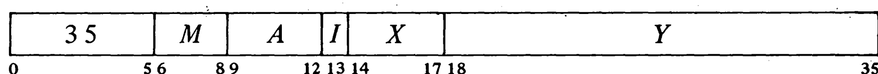


Increment AC by one and place the result back in AC. Compare the result with zero, and if the condition specified by *M* is satisfied, take the next instruction from location *E* and continue sequential operation from there. If AC originally contained $2^{35} - 1$, set the Overflow and Carry 1 flags;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

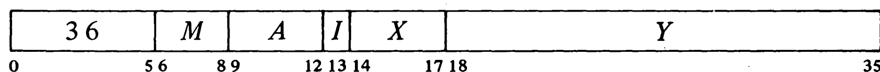
AOS Add One to Memory and Skip if Condition Satisfied



Increment the contents of location *E* by one and place the result back in *E*. 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 the Overflow and Carry 1 flags; if -1, set Carry 0 and Carry 1. If *A* is nonzero also place the result in AC.

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

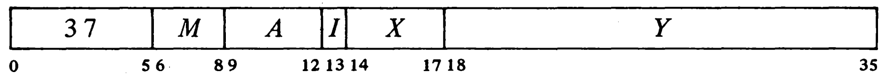


Decrement AC by one and place the result back in AC. Compare the result with zero, and if the condition specified by *M* is satisfied, take the next instruction from location *E* and continue sequential operation from there. If AC originally contained -2^{35}, set the Overflow and Carry 0 flags; if any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



Decrement the contents of location E by one and place the result back in E. 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 the Overflow and Carry 0 flags; if any other nonzero number, set Carry 0 and Carry 1. If A is nonzero also place the result in AC.

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 multiprocessor system. Suppose memory contains a routine that must be available to two processors but cannot be used by both at once. When one processor finishes the routine it sets location LOCK to −1. Either processor 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.

This procedure is invalid in the KA10 if the programmer

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

is making use of the drum split feature (which is not used by any DEC equipment).

Since it takes several days to count to 2^{36}, it is alright to keep testing the lock.

2.8 LOGICAL TESTING AND MODIFICATION

These eight instructions use a mask to modify and/or test selected bits in AC. The bits are those that correspond to 1s in the mask and they are referred to as the "masked bits". The programmer chooses the mask, the way in which the masked bits are to be modified, and the condition the masked bits must satisfy 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

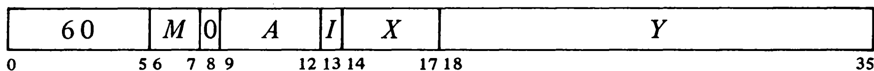
An additional letter may be appended to indicate the mode, which specifies the condition the masked bits must satisfy to produce a skip.

These 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.

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)

If the mnemonic has no suffix there is never any skip, and 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. Note that the skip condition must be satisfied by 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

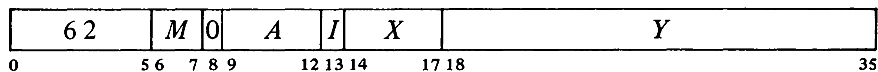


If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

TRN is a no-op.

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

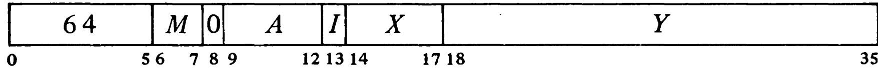
TRZ Test Right, Zeros, and Skip if Condition Satisfied



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 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

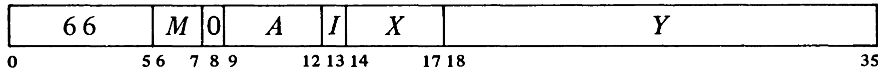
TRC **Test Right, Complement, and Skip if Condition Satisfied**



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

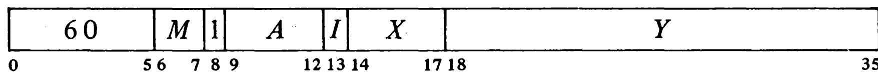
TRO **Test Right, Ones, and Skip if Condition Satisfied**



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	666

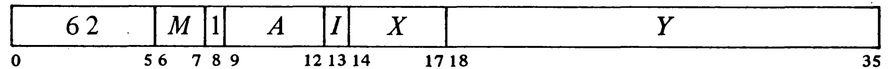
TLN **Test Left, No Modification, and Skip if Condition Satisfied**



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	TLN is a no-op.
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	

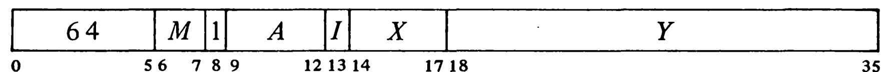
TLZ **Test Left, Zeros, and Skip if Condition Satisfied**



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

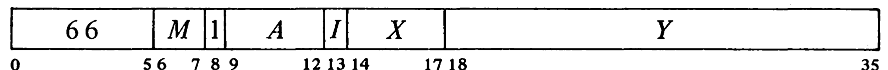
TLC **Test Left, Complement, and Skip if Condition Satisfied**



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

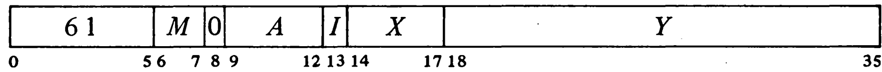
TLO **Test Left, Ones, and Skip if Condition Satisfied**



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**

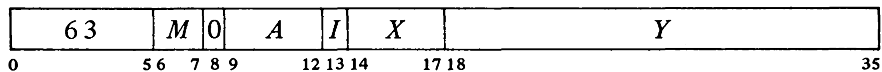


If the bits in AC corresponding to 1s in the contents of location E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

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

TDN is a no-op that references memory.

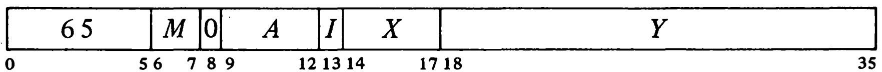
TDZ **Test Direct, Zeros, and Skip if Condition Satisfied**



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

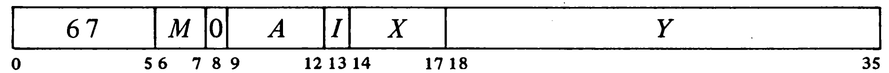
TDC **Test Direct, Complement, and Skip if Condition Satisfied**



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

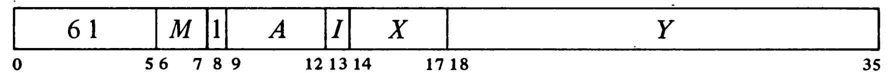
TDO Test Direct, Ones, and Skip if Condition Satisfied



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

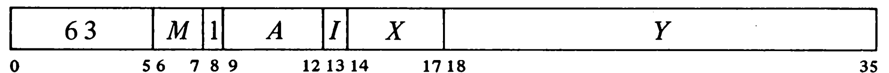


If the bits in AC corresponding to 1s in the contents of location E with its left and right halves swapped satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

TSN is a no-op that references memory.

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

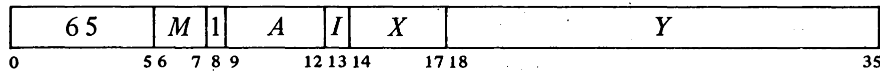
TSZ Test Swapped, Zeros, and Skip if Condition Satisfied



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

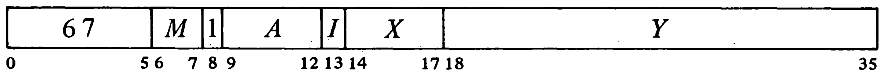
TSC **Test Swapped, Complement, and Skip if Condition Satisfied**



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**



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 right 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 we wish to skip if both bits 34 and 35 are 1 in location L. The following suffices.

```
SETCM   F,L
TRNE    F,3
```

We 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:

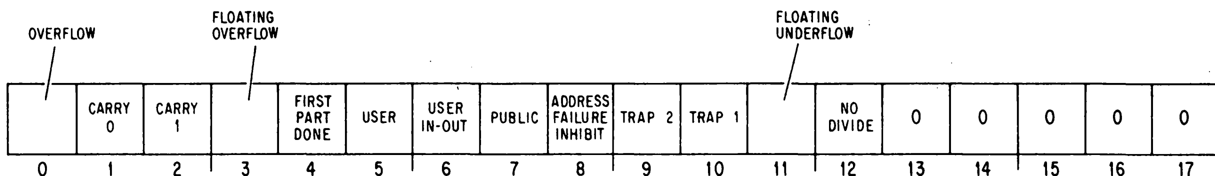
```
TRC     F,X+Y     ;Complement flags X and Y
TRCE    F,X+Y     ;Test both and restore original states
...               ;Do this if not both on
...               ;Skip to here if both on
```

2.9 PROGRAM CONTROL

The program control class of instructions includes the unimplemented user operations [*discussed in the next section*] and the arithmetic and logical test instructions. Some instructions in this class are no-ops, as are a few of the instructions for performing logical operations. The most commonly used no-op is JFCL, which is discussed below. No-ops among the instructions previously discussed are SETA, SETAI, SETMM, CAI, CAM, JUMP, TRN, TLN, TDN, TSN. Of these, SETA, SETAI, CAI, JUMP, TRN and TLN do not use the calculated effective address to reference memory. Hence in these instructions one can store any information in bits 18–35 without fear of attempting to address a location outside a user block or in a memory that does not exist.

The present section treats all program control instructions other than those mentioned above and in-out instructions that test input conditions [§2.12]. All but one of these 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. Also, all but two of the jumps are unconditional; one exception tests various flags, the other tests an accumulator.

Several of the jump instructions save the current contents of the program counter PC in the right half of an accumulator or memory location and save the states of various flags in the left half. The bits saved in the left half of



this PC word in KI10 user mode are as shown here. In the KA10, bits 7–10 are not used. In KI10 executive mode, bit 6 receives the same flag although it has a different meaning, and bit 0 receives a different flag altogether [*see below*]. In either processor all unused bit positions are cleared.

The following lists the left PC-word bit positions that receive information and explains the meaning of the flags at the time they are saved. Certain instructions can set up these flags to restore them to their original states following an interruption or to control specific situations. The explanations assume the flags reflect normal circumstances – not arbitrary rigging. In the following an *X* in a mnemonic indicates any letter (or none) that may appear in the given position to specify the mode, *eg* ADD*X* comprises ADD, ADDI, ADDM, ADDB.

Note that nothing is stored in bits 13–17, so when the PC word is addressed indirectly it can produce neither indexing nor further indirect addressing.

Bit *Meaning of a 1 in the Bit*

0 Overflow – any of the following has occurred:

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 in a negative number.

An MUL*X* has multiplied -2^{35} by itself (product 2^{70}).

An IMUL*X* has multiplied two numbers with product $\geqslant 2^{35}$ or $< -2^{35}$.

An FIX or FIXR has fetched an operand with exponent > 35.

Floating Overflow has been set (bit 3).

No Divide has been set (bit 12).

In user mode, bit 0 reflects the state of Overflow. But when the flags are saved in KI10 executive mode, bit 0 represents the Disable Bypass flag, which the Monitor uses to control certain aspects of the execution of an instruction by an executive XCT [*see below and* §2.15]. Although these are two separate flags that are read in different circumstances, when a PC word is used to restore or set up the flags, bit 0 conditions both of them.

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 ADD*X* has added two negative numbers with sum $< -2^{35}$.

An SUB*X* has subtracted a positive number from a negative number with difference $< -2^{35}$.

An SOJ*X* or SOS*X* has decremented -2^{35}.

But if set with Carry 1, indicates that one of these nonoverflow events has occurred:

In an ADD*X* both summands were negative, or their signs differed and their magnitudes were equal or the positive one was the greater in magnitude.

In an SUB*X* 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 AOJ*X* or AOS*X* has incremented -1.

An SOJ*X* or SOS*X* has decremented a nonzero number other than -2^{35}.

An MOVN*X* has negated zero.

Remember [§2.5], overflow is determined directly from the carries, not from the flags. The carry flags give meaningful information only if no more than one instruction that can set them occurs between clearing and reading them.

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 ADD*X* has added two positive numbers with sum $\geqslant 2^{35}$.

An SUB*X* has subtracted a negative number from a positive number with difference $\geqslant 2^{35}$.

An AOJ*X* or AOS*X* has incremented $2^{35} - 1$.

An MOVN*X* or MOVM*X* has negated -2^{35}.

But if set with Carry 0, indicates that one of the nonoverflow events listed under Carry 0 has occurred.

3 Floating Overflow – any of the following has set Overflow:

In a floating point instruction other than FLTR, DMOVN, DMOVNM or DFN, the exponent of the result was > 127.

Floating Underflow (bit 11) has been set.

No Divide (bit 12) has been set in an FDV*X*, FDVR*X* or DFDV.

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. *Eg* if an ILDB or IDPB is interrupted after the processing of the pointer but before the processing of the byte, the pointer now points not to the last byte, but rather to the byte that should be handled at the return [§2.13]. Thus when the processor restarts the instruction, it must retrieve the pointer but *not* increment it.

Although this flag is set upon completion of the first part of every interruptable two-part instruction, it is seldom relevant to the programmer as it is always cleared by the completion of the second part. The flag is seen only in an interruption, and its effect on the repeated first part is automatic provided only that it is properly restored at the return.

Besides indicating a priority interrupt in the middle of a byte instruction, the KI10 First Part Done indicates a page failure in the processing of a byte, in the transfer of the second (low order) word in a DMOVEM or DMOVMN, or in a noninterrupt data IO instruction that results from a block IO instruction (following the processing of the pointer [§2.12]).

5 User – the processor is in user mode [§§2.15, 2.16].

6 User In-out – even with the processor in user mode, there are no instruction restrictions (but memory restrictions still apply).

In the KA10, User In-out is applicable only to user mode [§2.16]. In the KI10 this flag has the stated effect when the processor is in user mode, but is used in executive mode to control certain aspects of the execution of an instruction by an executive XCT [*see below and* §2.15].

7 Public (KI10 only) – the last instruction performed was fetched from a public area of memory, *ie* the processor is in user mode public or executive mode supervisor.

8 Address Failure Inhibit (KI10 only) – an address failure cannot occur during the next instruction [§2.15].

9 Trap 2 (KI10 only) – if bit 10 is not also set, arithmetic overflow has occurred. If traps are enabled, the setting of this flag immediately causes one [§2.14]. At present, bits 9 and 10 cannot be set together by any hardware condition.

10 Trap 1 (KI10 only) – if bit 9 is not also set, pushdown overflow has occurred. If traps are enabled, the setting of this flag immediately causes one [§2.14]. At present, bits 9 and 10 cannot be set together by any hardware condition.

11 Floating Underflow – in a floating point instruction other than FLTR, DMOVN, DMOVNM or DFN, the exponent of the result was < -128 and Overflow and Floating Overflow have been set.

12 No Divide – any of the following has set Overflow:

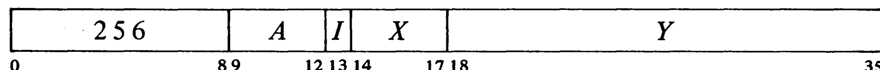
In a DIV*X* the dividend was greater than or equal to the divisor.

In an IDIV*X* the divisor was zero.

In an FDV*X*, FDVR*X* or DFDV 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.

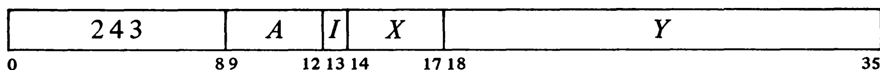
XCT Execute



Execute the contents of location *E* as an instruction. 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 jump (no matter how many XCTs precede a jump instruction, when PC is saved it contains an address one greater than the location of the first XCT in the chain).

A user XCT or any KA10 XCT acts as described here, and the *A* portion of the instruction is ignored. But in KI10 executive mode this instruction performs as stated only when *A* is zero. Nonzero *A* results in a so called "executive XCT", whose ramifications are far more widespread than indicated here [*for details refer to* §2.15].

JFFO Jump if Find First One



If AC contains zero, clear AC *A*+1 and go on to the next instruction in sequence.

If AC is not zero, count the number of leading 0s in it (0s to the left of the leftmost 1), and place the count in AC *A*+1. Take the next instruction from location *E* and continue sequential operation from there.

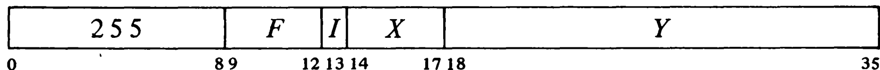
In either case AC is unaffected, the original contents of AC *A*+1 are lost.

Note that when AC is negative, the second accumulator is cleared, just as it would be if AC were zero.

To left-normalize an integer in AC:

```
JFFO   AC,.+1
ASH    AC,-1(AC+1)
```

JFCL Jump on Flag and Clear



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

This instruction can be used simply to clear the selected flags by having the jump address point to the next consecutive location, as in

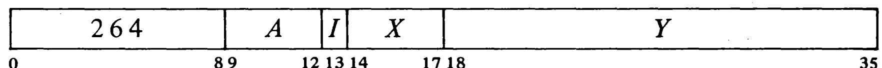
JFCL 17,.+1

which clears all four flags without disrupting the normal program sequence. A JFCL that selects no flag is the fastest no-op as it neither fetches nor stores an operand, and bits 18–35 of the instruction word can be used to store information.

To select one or a combination of these flags (which are among those described above) the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits, but MACRO recognizes mnemonics for 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 Carry 1	25510
JCRY	JFCL 6,	Jump on Carry 0 or 1	25530
JFOV	JFCL 1,	Jump on Floating Overflow	25504

JSR **Jump to Subroutine**

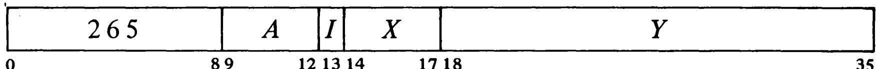


The A portion of this instruction is ignored.

Place the current contents of the flags (as described above) in the left half of location E and the contents of PC in the right half (at this time PC contains an address one greater than the location of the JSR instruction). Take the next instruction from location $E + 1$ and continue sequential operation from there. 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, bit 5 of the PC word stored is 1 and the processor leaves user mode, clearing Public. (In the KI10 an interrupt that is not dismissed automatically returns control to kernel mode.)

JSP **Jump and Save PC**

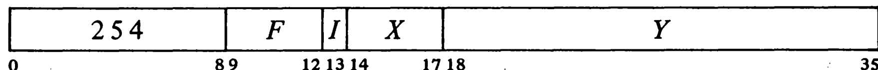


Place the current contents of the flags (as described above) in AC left and

the contents of PC in AC right (at this time PC contains an address one greater than the location of the JSP instruction). Take the next instruction from location *E* and continue sequential operation from there. 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, bit 5 of the PC word stored is 1 and the processor leaves user mode, clearing Public. (In the KI10 an interrupt that is not dismissed automatically returns control to kernel mode.)

JRST Jump and Restore



Perform the functions specified by *F*, then 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*

9 Restore the channel on which the highest priority interrupt is currently being held [§2.13].

Unless the User In-out flag is set, this function cannot be performed in a user program. Instead of restoring the channel, it acts just like an MUUO [§2.10].

10 Halt the processor. When it stops, the MA lights on the console display an address one 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).

Unless the User In-out flag is set, this function cannot be performed in a user program. Instead of halting the processor, it acts just like an MUUO [§2.10].

MA actually displays the address of the location that would have been executed next had the JRST been replaced by a no-op. So except for a JRST in a priority interrupt, MA points to the location one beyond that containing the instruction that caused the halt. This instruction is ordinarily the JRST or perhaps an XCT, but could even be a UUO.

11 Restore the flags listed above from the left half of the word in the last location referenced in the effective address calculation. Hence to restore flags requires that the JRST instruction use indexing or indirect addressing.

Restoration of all but the user and Public flags is directly according to the contents of the corresponding bits as given above: a flag is set by a 1 in the bit, cleared by a 0. A 1 in bit 5 sets User but a 0 has no effect, so the Monitor can restart 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 JRST is being performed by the Monitor, *ie* if User is clear. A 1 in bit 7 sets Public, but a 0

By manipulating the contents of the left half word used to restore the flags, the programmer can set them up in any desired way except that a user program cannot clear User or set User In-out, and no public program can clear

Public for itself. As an example, setting First Part Done prevents incrementing in the next ILDB, IDPB or noninterrupt KI10 block IO instruction provided there is no intervening JSR, JSP or PUSHJ. Note that if overflow traps are enabled, setting a trap flag immediately causes one.

clears it only if the JRST is being performed in executive mode with a 1 in bit 5 (*ie* User is being set). These conditions imply that the processor is entering user mode: hence the user cannot enter concealed mode by clearing Public; and although the supervisor can place the processor in user mode concealed, it cannot use this procedure to enter kernel mode.

12 *KA10.* Enter User mode. The user program starts at relocated location *E*.

KI10. The instruction is simply a jump except when fetched from a nonpublic area, in which case it clears Public. In other words a location containing a JRST 1, is a valid entry to a nonpublic area and the instruction places the processor in concealed or kernel mode.

To produce one or a combination of these functions the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits, but MACRO recognizes mnemonics for the most important 13-bit instruction codes (bits 0–12).

JRST	JRST 0,	Jump	25400
	JRST 10,	Jump and Restore Interrupt Channel	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 channel and restoring the flags for the interrupted program.

In a JRSTF or JEN the flags are restored from bits 0–12 of the final word retrieved in the effective address calculation; hence any JRST with a 1 in bit 11 must use indirect addressing or indexing, which takes extra time. If the PC word was stored in AC (as in a JSP), a common procedure is to use AC to index a zero address (*eg*, JRSTF (AC)), so its right half becomes the effective (jump) address. If the PC word was stored in core (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.

CAUTION

Giving a JRSTF or JEN without indexing or indirect addressing restores the flags from the instruction code itself.

While the KA10 is in user mode, if this instruction is executed as an interrupt instruction or by an MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode.

JFCL is the only jump that can test any of the flags directly. In fact it is the only basic program control instruction that can do so – several of the flags can be tested as processor conditions by in-out instructions, but these are ordinarily illegal in user programs anyway. But JFCL can test only four of the flags, 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. *Eg*, it may test the carry flags simply to take appropriate action in a double precision fixed point routine.

JSR and JSP are regularly used to call subroutines. They 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
JRST    .+2           ;Faster than skipping
JSR     OVRFLO        ;Jump over this if Overflow clear
  .
  .
  .
```

The fastest skip is CAIA in the KA10, TRNA in the KI10.

If we wish to go to the DIVERR routine when No Divide is set, we must first read the flags into a test accumulator T and then use a test instruction.

```
JSP     T,.+1         ;Store flags but continue in sequence
TLNE    T,40          ;40 left selects bit 12
JSR     DIVERR        ;Skip this if No Divide clear
  .
  .
  .
```

A subroutine called by a JSR must have its entry point reserved for the PC word. Hence it is nonreentrant: the JSR modifies memory so the subroutine cannot be shared with other programs. 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. But there are two ways to get back from a JSP. We can address the PC word indirectly with a JRST @AC (or JRSTF @AC if the flags must be restored), but we can also get it by addressing AC as an index register: JRST (AC). By using the second return method we 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).

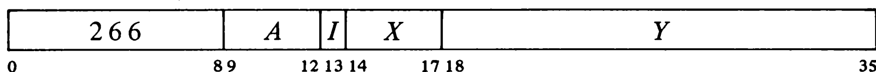
Suppose we wish to call a print subroutine and supply the words from which the characters are to be taken. Our main program would contain the following:

```
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 which 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
        ILDB    CH,T        ;Increment pointer and load byte
        JUMPE   CH,1(T)     ;Upon reaching zero character return
                            ;to one beyond last data word
        .                   ;Print routine
        .
        .
        JRST    PRINT+1     ;Get next character
```

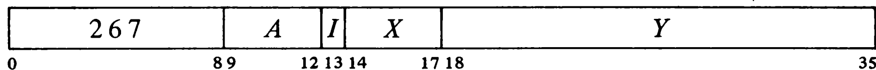
JSA **Jump and Save AC**



Place AC in location E, the effective address E in AC left, and the contents of PC in AC right (at this time PC contains an address one greater than the location of the JSA instruction). Take the next instruction from location $E+1$ and continue sequential operation from there. The original contents of E are lost.

While the KA10 is in user mode, if this instruction is executed as an interrupt instruction or by an MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode.

JRA **Jump and Restore AC**



Place the contents of the location addressed by AC left into AC. Take the next instruction from location E and continue sequential operation from there.

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 a 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

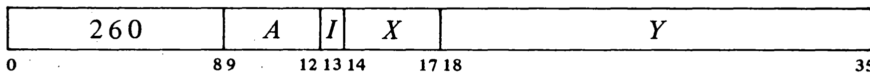
without tying up an accumulator to the extent a JSP would.

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, as shown by this example.

```
        .                       ;Main program
        .
        JSA     17,S1           ;Call to first subroutine (A)
        .
        .
S1:     0                       ;First subroutine starts here
        .
        .
        JSA     17,S2           ;Call to second subroutine (B)
        .
        .
        JRA     17,(17)         ;Return to A + 1 in main program
S2:     0                       ;Second subroutine starts here
        .
        .
        JSA     17,S3           ;Call to third subroutine (C)
        .
        .
        JRA     17,(17)         ;Return to B + 1 in first subroutine
S3:     0                       ;Third subroutine starts here
        .
        .
        JRA     17,(17)         ;Return to C + 1 in second subroutine
```

To call the next deeper subroutine at any level, a JSA places E and PC in the left and right of AC 17, saves the previous contents of AC 17 in E (the first subroutine location), and jumps to $E + 1$. To return to the next higher level, a JRA restores the previous contents of AC 17 from the location addressed by AC 17 left (the first subroutine location) and jumps to the location addressed by AC 17 right (the location following the JSA in the higher subroutine). If N lines of data for the next subroutine follow a JSA, the return to the location following the data is made by giving a JRA 17,N(17).

PUSHJ **Push Down and Jump**



260	A	I	X	Y
0–8	9–12	13	14–17	18–35

Add one to each half of AC and place the result back in AC. If the addition causes the count in AC left to reach zero, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. Then place the current contents of the flags (as described above) in the left half of the location now addressed by AC right and the contents of PC in the right half of that location (at this time PC contains an address one greater than the location of the PUSHJ instruction). Take the next instruction from location E and continue sequential operation from there.

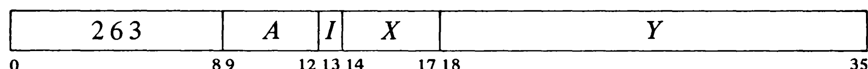
In the KI10 a PUSHJ executed as an interrupt instruction cannot set Trap 2.

The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared. However, pushdown overflow overrides the Trap 2 clear, so if the list overflows, Trap 2 sets and the KI10 traps instead of jumping. The original contents of the location added to the list are lost.

Note: The KA10 increments the two halves of AC by adding $1\ 000001_8$ to the entire register. In the KI10 the two halves are handled independently.

While the processor is in user mode, if this instruction is executed as an interrupt instruction or by a KA10 MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode, clearing Public. (In the KI10 an interrupt that is not dismissed automatically returns control to kernel mode.)

POPJ **Pop Up and Jump**



Subtract one from each half of AC and place the result back in AC. If the subtraction causes the count in AC left to reach −1, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. Take the next instruction from the location addressed by the right half of the location that was addressed by AC right *prior* to the decrementing, and continue sequential operation from there.

The effective address E is ignored. In the KI10 a POPJ executed as an interrupt instruction cannot set Trap 2.

Note: The KA10 decrements the two halves of AC by subtracting $1\ 000001_8$ from the entire register. In the KI10 the two halves are handled independently.

The address of the top item in the pushdown list is kept in the right half of the pointer in AC, and the program can keep a control count in the left half. In the KA10, incrementing and decrementing both halves of AC together is effected by adding and subtracting $1\ 000001_8$. Hence a count of -2 in AC left is increased to zero if $2^{18} - 1$ is incremented in AC right, and conversely, 1 in AC left is decreased to -1 if zero is decremented in AC right.

Since the pushdown list is independent of the subroutine called, PUSHJ-POPJ can be used like JSA-JRA for multiple entries. Moreover, ordering by level is inherent in the structure of a pushdown list [§2.2], 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 possible.

Unlike JSA-JRA, the pushdown instructions tie up an accumulator, but the usual procedure is to keep both data and jump addresses in a single list so only one AC is required for the most complex pushdown operations. The programmer must keep track of whether a given entry in the list is data or a PC word; in other words, every item inserted by a PUSH should be removed by a POP, and every PUSHJ should be matched by a POPJ. If flag

restoration is desired, the returning

```
        POPJ    P,
```

can be replaced by

```
        POP     P,AC
        JRSTF   (AC)
```

which requires another accumulator. 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.

By trapping or checking overflow and 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. If only jump addresses are kept in the list, the first procedure limits the depth of nesting. A technique to catch extra POPJs is to put a PC word addressing an error routine at the bottom of the list.

2.10 UNIMPLEMENTED OPERATIONS

Codes not assigned as specific instructions act as unimplemented operations, wherein the word given as an instruction is trapped and must be interpreted by a routine included for this purpose by the programmer. Codes in the range 001–077 are unimplemented user operations, or UUOs. Half of these (001–037) are for the local use of the user or Monitor (LUUOs); the other half (040–077) are set aside for user communication with the Monitor (MUUOs) and are interpreted by it (although they may be used by the Monitor as well). Codes 100 and above that are not used for instructions are regarded as the "unassigned codes"; 000 is not regarded as a legal code at all. Instructions that violate the instruction restrictions act in the same manner as MUUOs.

These are convenience mnemonics that mean nothing to the assembler. UUOs are also sometimes called "programmed operators".

Local Unimplemented User Operation

001–037	*A*	*I*	*X*	*Y*
0–8	9–12	13	14–17	18–35

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. Execute the instruction contained in location 41. The original contents of location 40 are lost.

Every LUUO uses some pair of locations numbered 40 and 41, but which such pair depends upon the circumstances. An LUUO in a user program uses relocated locations 40 and 41 and is thus entirely a part of and under control

If a single memory serves as memory number 0 for two KA10 processors, the second

(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 as it is assumed the Monitor would relocate these to different physical blocks.

of the user program. An LUUO in KA10 executive mode uses unrelocated locations. In KI10 executive mode an LUUO uses locations 40 and 41 in the executive process table.

The actions of MUUOs and unassigned codes depend to a considerable degree on the processor. All use at least two consecutive locations, where the first receives the information specified above for an LUUO (in the KI10 a third nonconsecutive location is also used). The unassigned codes are included so that the Monitor steps in when a user gives an incorrect code. The code 000 acts in exactly the same way as an MUUO but is not a standard communication code: it is included so that control returns to the Monitor should a user program wipe itself out.

The unassigned codes are 100–107, 114–117, 123 and 247.

Note that even in a dedicated system, the program must still define a user process table.

KI10. MUUOs and unassigned codes in user or executive mode act in exactly the same way. They store the information specified above for an LUUO in location 424 of the user process table, save the flags and PC (the current PC word) in location 425, set up the flags and PC according to a new PC word taken from a third location, and restart the processor in normal sequence at the location then addressed by PC. In the PC word saved in location 425, bit 0 may represent either Overflow or Disable Bypass depending upon the mode the processor is in when the MUUO is given. If the MUUO is given directly by the program, the address in the right half of the PC word saved is one greater than the location of the MUUO; otherwise it depends upon the circumstances in which the MUUO is executed. The new PC word can be taken from among 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 a trap (page failure or overflow).

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

Note that if overflow traps are enabled, setting a trap flag immediately causes one.

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. A 1 in bit 0 sets both Overflow and Disable Bypass; a 0 clears both. Hence bit 0 should be adjusted to produce the desired state in the flag that is relevant to the mode the processor is entering.

KA10. MUUOs and unassigned codes, 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). The unassigned codes are 100–127, 247 and 257. The codes 130–177, which are the floating point and byte manipulation instructions, are equivalent to the unassigned codes if unimplemented, *ie* if the hardware for them is not included. In this case all codes 100–177 trap to unrelocated 60-61.

The important point is that an MUUO or unassigned code results in executing an instruction in an unrelocated location, *ie* 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 include the unassigned and illegal codes as well.

Note that in executive mode, LUUOs and MUUOs act identically.

2.11 PROGRAMMING EXAMPLES

Before continuing to input-output and related subjects, let us consider some simple programs that demonstrate the use of a variety of the instructions described thus far.

The instruction repertoires of the KA10, the KI10 and the 166 processor used in the PDP–6 are very similar, and 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 three should have some way of distinguishing which machine it is running on. This simple test suffices.

```
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 if AC = 1000000
JRST    KI10       ;KI10 if AC = 0 (no carry between
                   ;halves)
```

Suppose we wish to count the number of 1s in a word. We could of course check every bit in the word. But there is a quicker way if we remember 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 at 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 one every time a 1 is deleted from T. After all 1s have been removed, the TDZE skips.

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 $r_n \ldots r_3 r_2 r_1$. *Eg* 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 pushdown 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.

Space can be saved in the pushdown 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 making the following substitutions.

```
PUSH P,T+1  →  HRLM  T+1,(P)
   POP P,T  →  HLRZ  T,(P)
```

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
```

MACRO interprets a number following ↑D as decimal.

Many data processing situations involve searching for information in tables and lists of all kinds. Suppose we wish to find a particular item in a table beginning at location TAB and containing *N* items. 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
```

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

```
        HRLZI   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
```

Locations used for a list can be scattered throughout memory if data is kept in the left half of each location and the right half addresses the next location in the list. The final location 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. At the end the final item is in T right.

```
        MOVE    T,(T)       ;Move next item to T
FIND:   TRNE    T,777777    ;Skip if AC right = 0
        JRST    .-2
        HLRZS   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
```

Double Precision Floating Point. The following are straightforward routines for handling double precision floating point arithmetic in software format [§2.6 *describes the floating point instructions*].

```
DFAD:   UFA     A+1,M+1     ;Sum of low parts to A+2
```

These routines are given to show the mechanics of double precision floating point operations. They produce correct results in all ordinary circumstances, but do not handle pathological cases.

```
        FADL    A,M         ;Sum of high parts to A, A+1
        UFA     A+1,A+2     ;Add low part of high sum to A+2
        FADL    A,A+2       ;Add low sum to high sum
        POPJ    P,

DFSB:   DFN     A,A+1       ;Negate double length operand
        PUSHJ   P,DFAD      ;Call double floating add
        DFN     A,A+1       ;-(M - AC) = AC - M
        POPJ    P,

DFMP:   MOVEM   A,A+2       ;Copy high AC operand in A+2
        FMPR    A+2,M+1     ;One cross product to A+2
        FMPR    A+1,M       ;Other to A+1
        UFA     A+1,A+2     ;Add cross products into A+2
        FMPL    A,M         ;High product to A, A+1
        UFA     A+1,A+2     ;Add low part to cross sum in A+2
        FADL    A,A+2       ;Add low sum to high part of product
        POPJ    P,
```

A double precision division is of the form

$$\frac{A}{B} = \frac{a + c \times 2^{-27}}{b + d \times 2^{-27}}$$

Using the relationship

$$A/b = q + r \times 2^{-27}/b$$

where q and r are the quotient and remainder produced by FDVL, the following routine computes a double length quotient by the algorithm

$$\frac{A}{B} \cong q + \frac{(r - qd) \times 2^{-27}}{b}$$

which gives a result correct to the next-to-last bit in the low order half.

```
DFDV:   FDVL    A,M         ;Get high part of quotient
        MOVN    A+2,A       ;Copy negative of quotient in A+2
        FMPR    A+2,M+1     ;Multiply by low part of divisor
        UFA     A+1,A+2     ;Add remainder
        FDVR    A+2,M       ;Divide sum by high part of divisor
        FADL    A,A+2       ;Add result to original quotient
        POPJ    P,
```

2.12 INPUT-OUTPUT

The input-output instructions govern all transfers of data to and from the peripheral equipment, and also perform many operations within the proc-

essor. An instruction in the in-out class is designated by 111 in bits 0–2, *ie* its left octal digit is 7. Bits 3–9 address the device that is to respond to the instruction. The format thus allows for 128 codes, two of which, 000 and 004 respectively, address the processor and priority interrupt, and are used for the console as well. The KA10 also uses the first two codes for the time share hardware, but the KI10 has a separate code, 010, for this purpose. A chart in Appendix A lists all devices for which codes have been assigned, and gives their mnemonics and DEC option numbers. Electrical and logical specifications of the IO bus are given in the interface manual.

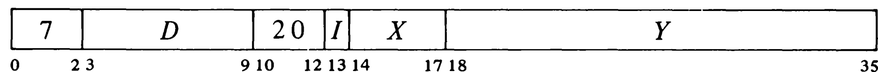
Bits 13–35 are the same as in all other instructions: they are the *I*, *X*, and *Y* parts, which are used to calculate an effective address, set of conditions, or mask to be used in the execution of the instruction. The remaining bits, 10–12, select one of the following eight IO instructions.

NOTE

All instructions described in the remainder of this manual are in-out instructions, which are affected by the time share instruction restrictions. In the KA10 no in-out instruction can be performed by a user mode program unless the User In-out flag is set. In the KI10, in-out instructions using device codes 740 and above are not restricted. But an instruction using a device code under 740 cannot be performed by a user mode program unless User In-out is set and cannot be performed in supervisor mode at all (in-out is normally handled in kernel mode). Any in-out instruction that violates these restrictions does not perform the functions given for it in the instruction description. Instead it acts just like an MUUO [§2.10].

These restrictions will not be mentioned in the instruction descriptions, as they apply to *all* instructions from this point on.

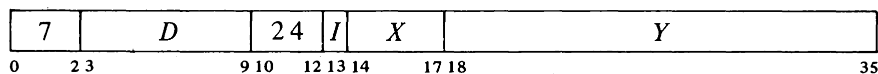
CONO **Conditions Out**



Set up device *D* with the effective initial conditions *E*. The number of condition bits in *E* that are actually used depends on the device.

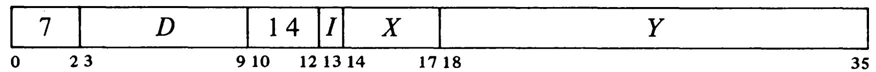
E will always be regarded as being bits 18–35, even though it is actually placed on both halves of the bus and many devices receive the information from the left half.

CONI **Conditions In**



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.

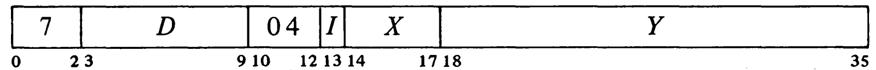
DATAO Data Out



Send the contents of location E to the data buffer in device D, and perform whatever control 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.

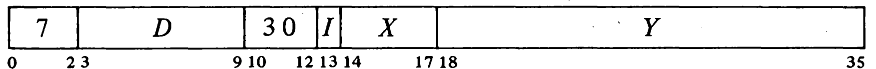
DATAI Data In



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.

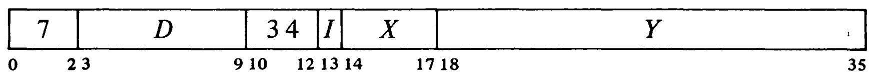
CONSZ Conditions In and Skip if Zero



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 the right 18 are tested.

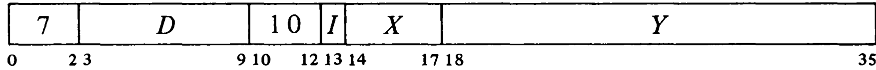
CONSO Conditions In and Skip if One



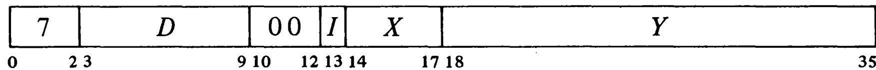
Test the input conditions from device D against the effective mask E. If any condition bit selected by a 1 in E is 1, skip the next instruction in sequence.

If the device supplies more than 18 condition bits, only the right 18 are tested.

BLKO **Block Out**



BLKI **Block In**



Add one to each half of a pointer 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 a BLKI, perform a DATAI.

The remaining actions taken by this instruction depend on whether it is executed as a priority interrupt instruction [§2.13].

♦ *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 channel. Otherwise dismiss the interrupt and return to the interrupted program.

Note: The KA10 increments the two halves of the pointer by adding $1\,000001_8$ to the entire register. In the KI10 the two halves are handled independently.

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, the right half contains an address one 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 of four types, represented by the first four instructions described above. Moreover, a complete treatment of the programming of any device can be given in terms of these four instructions, two of which are for input and two for output. The four exhaust the types of information transfer that occur in the IO system, at least three of which are applicable to any given device. Thus all instruction descriptions in the rest of this manual will be of the CONO, CONI, DATAO and DATAI instructions combined with the various device codes. The discussion of each device will present timing information pertinent to device operation, as internal device timing is dependent only upon the device and not upon processor instruction time (which is given in Appendix C).

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.

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 right eighteen 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, such as status levels from a tape transport, have no direct connection with output conditions.

Data is moved in and out in characters of various sizes or in full 36-bit words. Each transfer between memory and a device data buffer requires a single DATAI or DATAO. Every device has a CONO and CONI, but it may have only one data instruction unless it is capable of both input and output. *Eg*, the paper tape reader has only a DATAI, the tape punch has only a DATAO, but the teletype has both. (A high speed device, such as a disk file, can be connected to a direct-access processor, which in turn is connected directly to memory by a separate memory bus and handles data automatically. This eliminates the need for the program to give a DATAO or DATAI for each transfer.)

A DATAI that addresses an output-only device simply clears location *E*. DATAI PI, (code 70044) produces only this effect as the priority interrupt has no data for input. 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 absolute skip, a DATAI or CONI clears location *E*.

A Typical IO Device. Every device has a 7-bit device selection network, a priority interrupt assignment, and at least two flags, Busy and Done, or some equivalent. The selection network decodes bits 3–9 of the instruction so that only the addressed device responds to signals sent by the processor over the in-out bus. To use the device with the priority interrupt, the program must assign a channel to it. Then whenever an appropriate event occurs in the device, it requests an interrupt on the assigned channel.

The Busy and Done flags together denote the basic state of the device. When both are clear the device is idle. To place the device in operation, a CONO or DATAO sets Busy. If the device will be used for output, the program must give a DATAO that sends the first unit of data – a word or character depending on how the device handles information. When the device has processed a unit of data, it clears Busy and sets Done to indicate that it is ready to receive new data for output, or that it has data ready for input. In the former case the program would respond with a DATAO to send more data; in the latter, with a DATAI to bring in the data that is ready. If an interrupt channel has been assigned to the device, the setting of Done signals the program by requesting an interrupt; otherwise the program must keep testing Done to determine when the device is ready.

Busy and Done both set is a meaningless situation.

All devices function basically as described above even though the number of initial conditions varies considerably. Besides Busy and Done flags, the tape reader and punch have a Binary flag that determines the mode of operation of the device with respect to the data it processes – alphanumeric or binary. The teletype has no binary flag, but it has two Busy flags and two Done flags – one pair for input, another for output. A complicated device, such as magnetic tape, may require two device codes to handle the large number of conditions associated with it. Initial conditions for a tape system include a transport address and an actual command the tape control is to perform; input conditions include error flags and transport status levels.

Occasionally a device with a second code may use a DATAI or DATAO to transmit additional control or maintenance information.

Most IO devices involve motion of some sort, usually mechanical (in a display only the electron beam moves). With respect to mechanical motion

there are two types of devices, those that stay in motion and those that do not. Magnetic tape is an example of the former type. Here the device executes a command (such as read, write, space forward) and the done flag indicates when the entire operation is finished. A separate data flag signals each time the device is ready for the program to give a DATAI or DATAO, but the tape keeps moving until an entire record or file has been processed.

Paper tape, on the other hand, stops after each transfer, but the program need not give a new CONO every time. The reader logic is set up so that a DATAI not only reads the data, but also clears Done and sets Busy. Hence if the instruction is given within a critical time, the tape moves continuously and only two CONOs are required for a whole series of transfers: one to start the tape, and one to stop it after the final DATAI.

Other devices operate in one or the other of these two ways but differ in various respects. The tape punch and teletype output are like the reader. Teletype input is initiated by the operator striking a key rather than by the program. The card reader reads an entire card on a single CONO, with a DATAI required for each column. The DECtape stays in motion, and the program must give a CONO to stop it or it will go all the way to the end zone.

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. 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, selects physical page 0 for the executive process table, and disables overflow traps. 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 memory (including fast memory) except in location 0. The operation affects none of memory except location 0 and the block area. For the KI10 it is recommended that read in be confined to the unpaged area, as bringing data into locations above 337777 would require prior loading of the appropriate pointers into the executive page map in physical page 0.

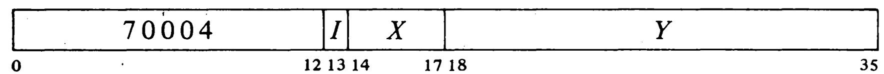
Upon completing the block, the processor halts only if the single instruction switch is on. Otherwise it leaves readin mode and begins normal operation. This is done in the KI10 by jumping to the location addressed by the last word in the block, in the KA10 by executing the last word as an instruction.

Console-Program Communication

Neither the processor nor the priority interrupt system require all four types of IO 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 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 [§§2.13, 2.14].

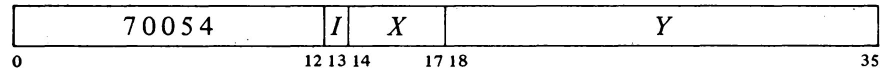
MACRO also recognizes the mnemonic RSW (Read Switches) as equivalent to DATAI APR,.

DATAI APR, **Data In, Console**



Read the contents of the console data switches into location *E*.

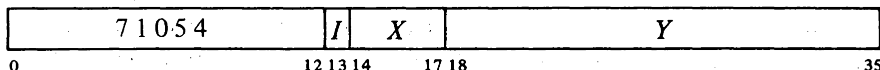
DATAO PI, **Data Out, Console**



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).

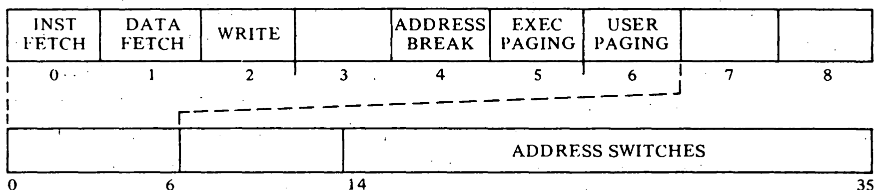
Once the indicators have been loaded by the program, no address condition selected from the console [§§2.18, 2.19] can load them until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key.

DATAO PTR, Operating Data Out, Console



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).

On the KI10 console, all switches are pushbutton-flipflop combinations; the instruction of course controls the flipflops, not the buttons.



For complete information on the use of these switches, see §2.19.

2.13 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; *ie* 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 channels arranged in a priority chain, with assignment of devices to channels entirely at the discretion of the programmer. To assign a device to a channel, the program sends the number of the channel to the device control register as part of the conditions given by a CONO (usually bits 33–35). Channels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the device from the interrupt channels altogether. Any number of devices can be connected to a single channel, and some can be connected to two channels (*eg* a device may signal that data is ready on one channel, that an error has occurred on another).

Interrupt Requests. When a device requires service it sends an interrupt request signal over the in-out bus to its assigned channel in the processor. If the channel is on, the processor accepts the request at the next memory access unless the processor is either starting an interrupt on any channel or holding an interrupt on the same channel. The request signal is a level, so it remains on the bus until turned off by the program (CONO, DATAO or

DATAI). Thus if a request is not accepted because of the conditions given above, it will be accepted when those conditions no longer hold. A single channel 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 channel 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 channel 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. *Eg* 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.

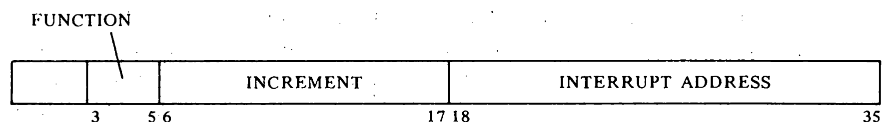
Starting an Interrupt. After a request is accepted the channel must wait for the interrupt to start. No interrupts can be started unless the priority interrupt system is active. Furthermore, the processor cannot start an interrupt if it is already holding an interrupt on a channel with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). If there is a higher priority channel waiting, the processor stops the current program to start an interrupt on the waiting channel that has highest priority. The interrupt starts 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), or following a transfer in a BLT. The KI10 can also interrupt the relatively long process of calculating the quotient in double floating division. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

For the KA10 two fixed memory locations are associated with each channel: unrelocated locations $40 + 2N$ and $41 + 2N$, where N is the channel number. Channel 1 uses locations 42 and 43, channel 2 uses 44 and 45, and so on to channel 7 which uses 56 and 57. The KA10 starts an interrupt for channel N by executing the instruction in the first interrupt location for the channel, *ie* location $40 + 2N$. Even though the processor may be in user mode when an interrupt occurs, the interrupt operations are performed in executive mode.

Interrupt locations for a second processor are $140 + 2N$ and $141 + 2N$.

The KI10 starts an interrupt by sending an interrupt-granted signal for the channel on which it has accepted a request. This 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 channel sends the signal on to the next device. A device that is requesting an interrupt on the specified channel terminates the signal path and sends an interrupt function word back to the processor. The KI10 also has a pair of fixed locations associated with each channel, and these have the same numbers as in the KA10 but are locations in the executive process table. These 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.

Note that there are therefore two orders of priority associated with a KI10 interrupt: first the channel, and then for all devices requesting interrupts simultaneously on the same channel, proximity to the processor on the bus.



Bits 3–5	*Interrupt Function*
0	Processor waiting or no response. If the latter, perform a standard interrupt (see function 1).
1	Standard interrupt – execute the instruction in location 40 + 2*N* 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 6–17 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 bits 18–35 as the effective address.
5	DATAI – do a DATAI for this device using the contents of bits 18–35 as the effective address.
6	Not used – reserved by DEC.
7	Not used – reserved by DEC.

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. This means that for simultaneous requests on a given channel, all KI10 devices have priority over KA10 devices.

At present, functions 6 and 7 produce standard interrupts.

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode.

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 channel, in direct response by the hardware (rather than by the program) to a request on that channel. §2.12 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 special effects produced by different types of interrupt instructions depend upon the processor.

KI10 Interrupt Instructions. Besides instructions, the KI10 can perform other interrupt operations as described above. 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 channel assigned to the processor [§2.14]. These considerations of course do not apply to a service routine called by an interrupt instruction. 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 (*ie* it returns control to the un-

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 §2.12, the condition being that the count is not zero.

changed PC). If the skip condition is not satisfied, the processor executes the instruction contained in the second interrupt location.

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 channel, 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 an executive XCT [§2.15]. 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 nor a BLT will function in a reasonable manner as an interrupt instruction. Therefore do not use them.

KA10 Interrupt Instructions. In the KA10 the interrupt instructions fall into two categories.

♦ *Non-IO Instructions.* After executing a non-IO interrupt instruction, the processor holds an interrupt on the channel 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 [§2.16]).

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. *Eg* 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 IO 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 (*ie* 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 + 2*N* is a BLKI or BLKO and the count does reach zero, the processor executes the instruction in location 41 + 2*N*. This *cannot* be an IO instruction and the actions that result from its execution as an interrupt instruction are those given above for non-IO instructions.

Caution

> The execution, as an interrupt instruction, of a CONO, CONI, CONSO or CONSZ in location 40 + 2*N* or *any* IO instruction in location 41 + 2*N* hangs up the processor.

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 channel. Thus interrupts can be held on a number of channels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that channel or any channel of lower priority (requests, however, can be accepted on lower priority channels).

A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be active when the JEN is given). This instruction restores the channel on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority channels. 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 in the KI10, an MUUO. If flag restoration is not desired, a JRST 10, can be used instead.

Caution

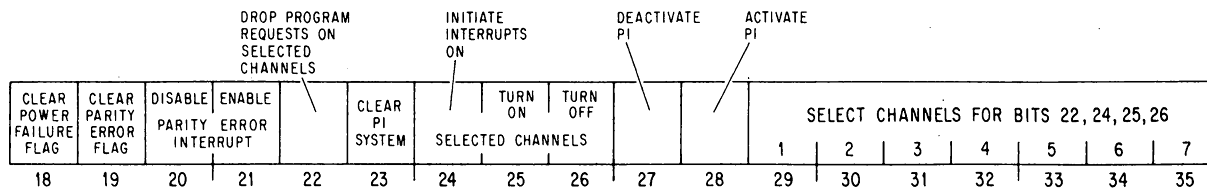
> An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its channel and all channels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Priority Interrupt Conditions. The program can control the priority interrupt system by means of condition IO 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).



Notes.

Bits 18–21 are actually for processor conditions [§2.14].

20 Prevent the setting of the Parity Error flag from requesting an interrupt on the channel assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the channel assigned to the processor.

22 *KI10 only:* On channels selected by 1s 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 channels, 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 channels selected by 1s in bits 29–35, and force the processor to accept them even on channels that are off.
KA10: There is at most one interrupt on a given channel, and a request is lost if it is made by this means to a channel on which an interrupt is already being held.
KI10: The request remains indefinitely, so as soon as an interrupt is completed on a given channel another is started, until the request is turned off by a CONO that selects the same channel and has a 1 in bit 22.

25 Turn on the channels selected by 1s in bits 29–35 so interrupt requests can be accepted on them.

26 Turn off the channels selected by 1s in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

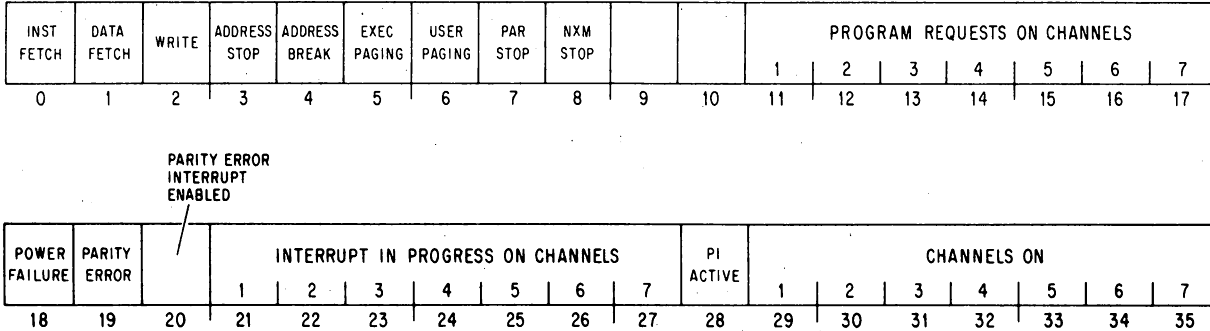
27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.

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 (as well as several bits of KA10 processor conditions and nine KI10 console operating switches) into location *E* as shown.



Notes.

Channels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate channels on which interrupts are currently being held; 1s in bits 11–17 (which are available only in the KI10) indicate channels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the priority interrupt system is active.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 0–8 are available only in the KI10, where they reflect the settings of various console operating switches; for information on these switches refer to §2.19. Bits 18–20 actually read KA10 processor status conditions [§2.14] as follows.

18 Ac power has failed. The program should save PC, the flags and fast memory in core, and halt the processor.

The setting of this flag requests an interrupt on the channel 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 channel assigned to the processor, at which time PC points to the instruction being performed or to the one following it.

Timing. The time a device must wait for an interrupt to start depends on the number of channels in use, and how long the service routines are for devices on higher priority channels. If only one device is using interrupts, it never waits longer than 10 μs with the KI10. With the KA10 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. 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 channels, while a break 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. *Eg* 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 a KI10 MUUO the routine must save locations 424 and 425 of the user process table. In other cases it is not the pair of consecutive locations that are relevant, as the second contains the instruction to handle the UUO. Thus for a KA10 UUO or a KI10 LUUO the routine must save location 40, unrelocated or in the executive process table respectively, 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.

2.14 TRAPPING AND PROCESSOR CONDITIONS

In the performance of a program there are many events that cannot be foreseen and whose occurrence requires special action by the program. There are instructions that test for various conditions, but in say a long string of computations it would be both cumbersome and time consuming to test for overflow at every step. It is far better simply to allow an event such as overflow to break right into the normal program sequence.

For situations of this nature, various internal conditions can interrupt the program. Both processors use condition IO instructions to control the appropriate flags and to inspect other conditions of interest to the program. The KI10 also has a trapping mechanism that allows conditions due directly to the program, and which are often permitted to happen as a matter of course, to interrupt the program sequence without recourse to the priority interrupt system. Violation of instruction restrictions by a user or the supervisor is handled by trapping as an MUUO; violation of memory restrictions is handled as a processor condition in the KA10 (as explained here) but is handled in the KI10 by trapping [§2.15].

Overflow Trapping (KI10 Only)

Overflow produced by an interrupt instruction cannot be detected. In any other circumstances, an instruction in which an arithmetic overflow condition occurs sets Overflow and Trap 1, and an instruction in which a pushdown overflow occurs sets Trap 2. If overflow traps have been enabled by the Monitor, then 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 taken from a particular location in the process table for the program (user or executive). The location as a function of the trap flags set is as follows.

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.

Trap Flags Set	*Trap Type*	*Trap Number*	*Location*
Trap 1 only	Arithmetic overflow	1	421
Trap 2 only	Pushdown overflow	2	422
Trap 1 and 2	Not used by hardware	3	423

A trap can be produced artificially simply by setting up the trap flags with a JRSTF or MUUO. In this way the program can also use trap number 3, which at present cannot result from any hardware-detected condition (it is reserved for future use by DEC).

A trap instruction is executed in the same address space as the instruction that caused it. Overflow in a user instruction traps 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, *eg* by requesting the Monitor to place a PUSHJ to a user routine in the trap location. An MUUO must be used if the Monitor is to handle a user-caused trap.

The trap instruction (the final instruction in an XCT and/or LUUO string) clears the trap flags, so the processor returns to the interrupted 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,

An arithmetic instruction that overflows on every iteration produces an infinite loop if used as a trap instruction for arithmetic overflow. A pushdown instruction in a pushdown overflow trap can overflow only once. (The memory allocated to a push-down stack should have at least one extra location to handle this case – two extras if the program and the trap both use the same pointer.)

should the trap instruction itself set a trap flag (not necessarily the same one), a second trap occurs.

An interrupt can occur between an instruction that overflows and the trap instruction, but the latter 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; but the overflow trap will be taken care of after the offending 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.

KI10 Processor Conditions

In the KI10, page failures and overflow are handled by trapping, but there are a number of other internal conditions that can signal the program by requesting an interrupt on a channel assigned to the processor. The program can actually assign two channels – 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 [§2.13]. Inspection of other conditions and control over all are handled by condition IO 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.

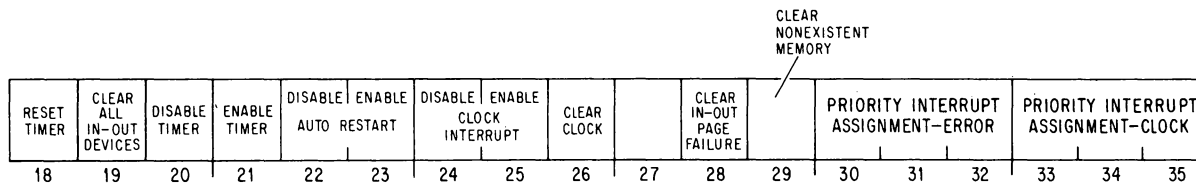
One of the features controlled by the CONO for the processor is the automatic restart after power failure. This restart applies only when the levels on the power mains go below specification while the processor is running, and the power switch is on when power is restored – the machine never begins operation by itself when the operator turns the power switch on or off. Inadequate power, over temperature, etc are indicated by the Power Failure flag. The program must both enable the auto restart feature and respond to the setting of Power Failure in order for the processor to restart itself. If the program fails to clear Power Failure or enable the auto restart within 4 ms after failure is detected, there is no restart. But if the auto restart is enabled and Power Failure is clear, then when power levels are again adequate the processor will restart itself by executing the instruction in location 70 in kernel mode (provided the power switch is on).

The processor device code is 000, mnemonic APR.

CONO APR, **Conditions Out, Arithmetic Processor**

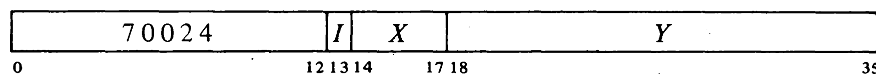
70020	*I*	*X*	*Y*
0 12	13	14 17	18 35

Assign the interrupt channels 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).

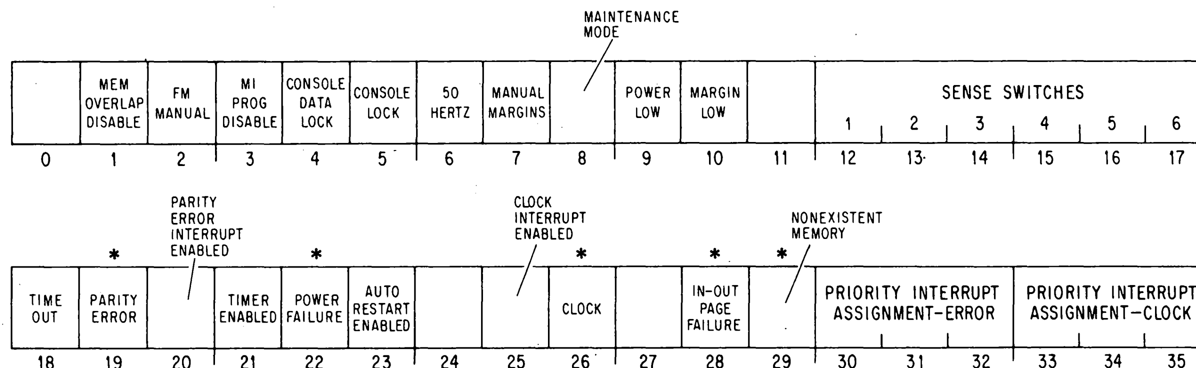


A 1 in bit 19 produces the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system nor the processor conditions).

CONI APR, Conditions In, Arithmetic Processor



Read the status of the processor (as well as various console switches and maintenance functions) into location E as shown.



Notes.

*These bits cause interrupts.

Interrupts are requested on the error channel (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 channel (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 §2.19. Bits 7–10 are maintenance functions for which the reader should refer to Chapter 10 of the maintenance manual.

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.

6 The system is operating on 50 Hz line power. This is important to the program, not only because some IO devices run slower on 50 Hz, but because the program must compensate for the time difference when using the line frequency clock (bit 26).

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.

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.

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 channel, at which time PC points to the instruction being performed or to the one following it.

22 Ac power has failed. The program should save PC, the flags, mode information and fast memory in core, and halt the processor.

The setting of this flag requests an interrupt on the error channel. After 4 ms the processor is cleared. But at that time, if Auto Restart Enabled is set and the program has cleared Power Failure (CONO PI,400000), then when adequate power levels are restored, the processor will go back into normal operation in kernel mode at location 70 (provided the power switch is on).

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 channel.

28 A page failure has occurred in an interrupt instruction. The setting of this flag requests an interrupt on the error channel.

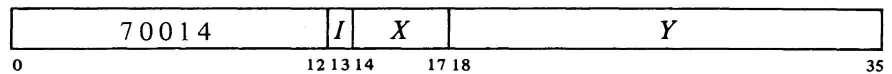
Note: A page failure in an interrupt instruction is regarded as a fatal error, and it causes an interrupt instead of a page failure trap. The kernel mode program is expected to set up the interrupt instructions so that a page failure simply cannot occur.

PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

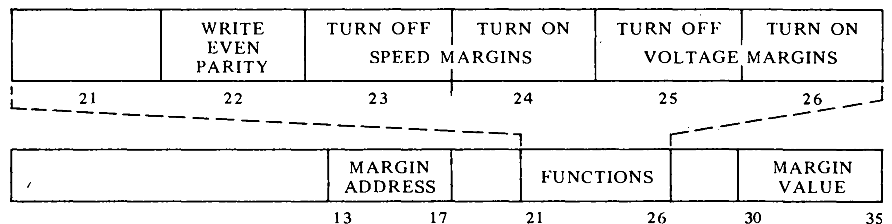
29 The processor attempted to access a memory that did not respond within 100 μs. The setting of this flag requests an interrupt on the error channel, at which time PC points either to the instruction containing the unanswered reference or to the one following it.

This instruction is for maintenance only. For further information refer to Chapter 10 of the *KI10 Maintenance Manual.*

DATAO APR, Maintenance Data Out, Arithmetic Processor



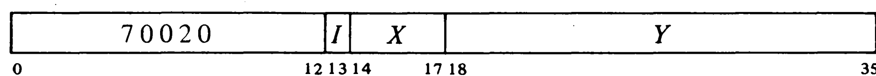
Supply diagnostic information and perform diagnostic functions according to the contents of location *E* as shown.



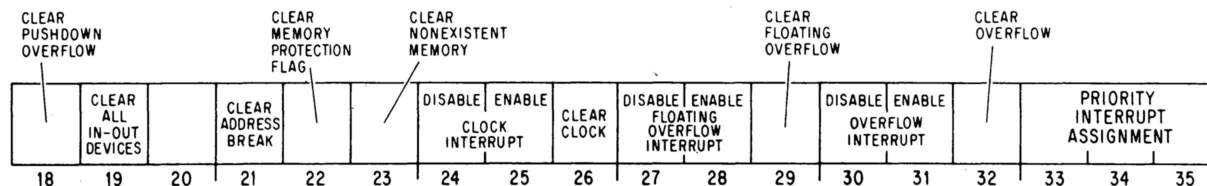
KA10 Processor Conditions

There are a number of internal conditions that can signal the program by requesting an interrupt on a channel assigned to the processor. Flags for power failure and parity error are handled by the condition IO instructions that address the priority interrupt system [§2.13]. 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



Assign the interrupt channel 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).

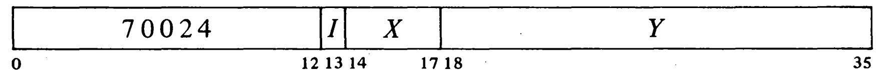


Notes.

Enabling a particular flag to interrupt means that henceforth the setting of the flag will request an interrupt on the channel assigned (by bits 33–35) to the processor. Disabling prevents the flag from triggering a request.

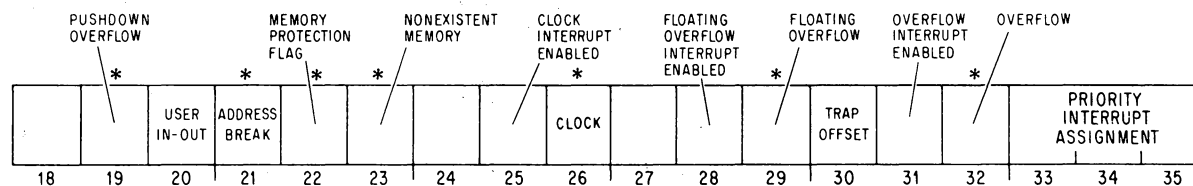
A 1 in bit 19 produces the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system, nor the processor flags cleared by this instruction or CONO PI,).

CONI APR, Conditions In, Arithmetic Processor



Read the status of the processor into the right half of location E as shown (all interrupt requests are made on the channel assigned to the processor).

*These bits request interrupts.



Notes.

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.

20 User In-out – even if the processor is in user mode, there are no instruction restrictions (but memory restrictions still apply) [§2.16].

PC bears no relation to the break if the access was requested for a console key function.

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.

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.

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.

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.

PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

23 Nonexistent Memory – the processor attempted to access a memory that did not respond within 100 μs. The setting of this flag requests an interrupt, at which time PC points either to the instruction containing the unanswered reference or to the one following it.

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 at the beginning of §2.9. 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 at the beginning of §2.9. 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.

2.15 KI10 MODES

General information about the machine modes and paging procedures is given in Chapter 1, in particular at the end of the introductory remarks and at the end of §1.3. 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.

User Programming. As far as user programming is concerned, all of the necessary information has already been presented. For convenience however we list here the rules the user must observe. [*Refer to the Monitor manual for further information including use of the Monitor for input-output.*]

♦ If possible, limit your memory needs to 32K, using addresses 0–37777 and 400000–437777, to gain the savings afforded by having the status of a "small user". There are no restrictions of any kind on addresses 0–17 as these are in fast memory and are available to all users (even though page 0 may otherwise be inaccessible).

♦ If an area of memory is write-protected, *eg* 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 040–077 only in the manner prescribed in the Monitor manual. In general, unless they are prescribed for special circumstances, code 000 and the unassigned codes should not be used.

♦ Unless User In-out is set do not give any IO instruction with device code less than 740, HALT (JRST 4,) or JEN (JRST 12, (specifically JRST 10,)). The program can determine if User In-out is set by examining bit 6 of the PC word stored by JSR, JSP or PUSHJ.

♦ 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, *ie* a location containing a JRST 1,.

The user can give a JRSTF (JRST 2,) but a 0 in bit 5 of the PC 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.

The above rules are the result of KI10 hardware characteristics. But in a real sense many of these rules 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 and/or very large (*eg* larger than available physical core), the actual constraints will be dictated by the particular Monitor and the system manager. It may be desirable (for compatible operation with KA10 systems) to 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 core than necessary or cause excessive page swapping.

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 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.

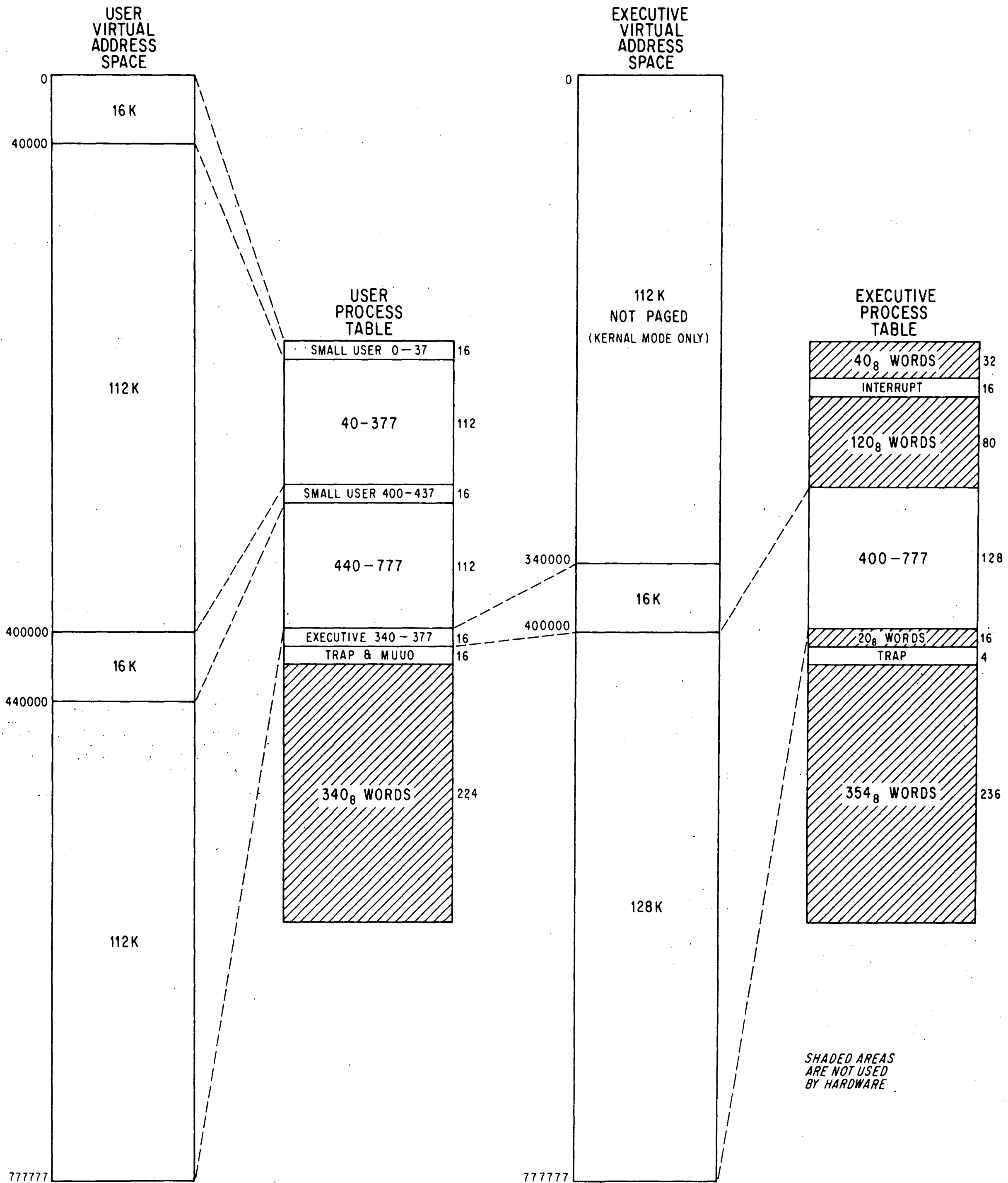
Actually page 0 has only 496 locations using addresses 20–777, as addresses 0–17 reference fast memory, which is unrestricted and available to all programs. (In general a user cannot reference the first sixteen core locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to core and are not made by an instruction executed by an executive XCT [*see below*].

The paging hardware 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. *Eg* 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 400000–437777. This is pages 0–37 and 400–437, 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 400–417. 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.

Thus 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.

The illustrations on the next two pages 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



VIRTUAL ADDRESS SPACE AND PAGE MAP LAYOUT

USER PROCESS TABLE

Address	Contents	
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	PC WORD OF MUUO STORED HERE	
426	EXECUTIVE PAGE FAILURE WORD	
427	USER PAGE FAILURE WORD	
430	KERNEL NO TRAP NEW MUUO PC WORD	
431	KERNEL TRAP NEW MUUO PC WORD	
432	SUPERVISOR NO TRAP NEW MUUO PC WORD	
433	SUPERVISOR TRAP NEW MUUO PC WORD	
434	CONCEALED NO TRAP NEW MUUO PC WORD	
435	CONCEALED TRAP NEW MUUO PC WORD	
436	PUBLIC NO TRAP NEW MUUO PC WORD	
437	PUBLIC TRAP NEW MUUO PC WORD	
440–777	*AVAILABLE TO SOFTWARE*	

EXECUTIVE PROCESS TABLE

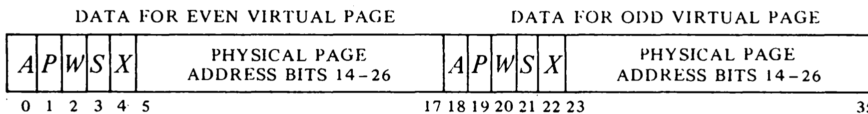
Address	Contents	
0–37	*AVAILABLE TO SOFTWARE*	
40	EXECUTIVE LUUO STORED HERE	
41	LUUO HANDLER INSTRUCTION	
42–57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
60–177	*AVAILABLE TO SOFTWARE*	
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400–417	*AVAILABLE TO SOFTWARE*	
420	EXECUTIVE PAGE FAILURE TRAP INSTRUCTION	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE PUSHDOWN OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424–777	*AVAILABLE TO SOFTWARE*	

PROCESS TABLE CONFIGURATION

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 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. The Monitor also specifies whether each page is public or not and writeable or not. Each word in the page map has this format to supply the necessary information for two virtual pages.

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.



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	Writeable (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 paging hardware 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 IO instructions that address the paging hardware [*see below*].

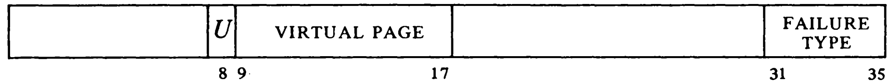
At each reference the hardware compares the page number supplied by

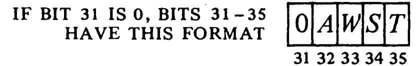
the program with those in the virtual part of the page table. If there is a match for the appropriate address space, the corresponding entry in the physical list is used as the left thirteen bits in the physical address (provided of course that the reference is allowable according to the *P* and *W* bits). If there is no match, the hardware makes a memory reference 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 channel 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. 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.

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.





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.

Since a user page failure trap instruction is executed in user address space, the Monitor should be careful not to have the trap instruction do indirect addressing that might cause another page failure.

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 requested access to the memory location that was specified by the paging and address switches and for which a comparison was enabled, and the memory reference was for the purpose selected by the address condition switches as follows:

Whether or not a comparison can be made is a function of the settings of the paging switches [§2.19] and the state of the User Address Compare Enable flag [*see below*].

The instruction fetch 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) 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.

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.

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.

Tests for hard page failures are actually made in the order given here.

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. 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.

In any 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 writeability. When bringing in a user program, the Monitor would ordinarily indicate as writeable only the buffer area and other pages that will definitely be altered. Then in response to a write failure, the Monitor makes the page writeable 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 kernel program supplies data tables of all kinds to the supervisor program, and the latter cannot affect them). 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. 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.

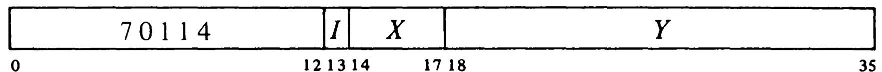
If the Monitor shared block 0 with any users, it would have to store the user accumulators even when taking control only temporarily.

Even while User is set, the interrupt instructions are not part of the user program and are thus subject only to executive restrictions. 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.

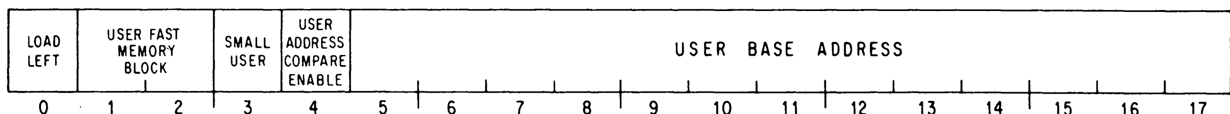
The page failure and overflow trap instructions are executed in the user address space if caused by the user.

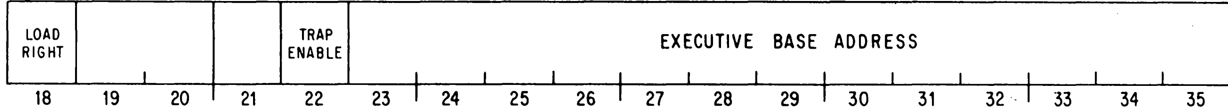
The paging hardware has one non-IO instruction and two condition IO instructions primarily for diagnostic purposes. Otherwise control over the system is exercised by data IO instructions. The device code for the paging hardware is 010, mnemonic PAG.

DATAO PAG, Data Out, Paging



Set up the paging hardware according to the contents of location E as shown.





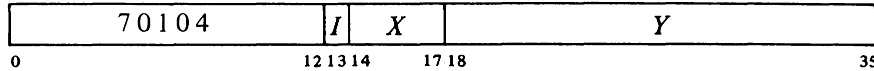
Bits 0 and 18 are change bits. If bit 0 is 0, ignore the rest of the left half word. But if bit 0 is 1, load bits 5–17 into the user base register to select the user process table, select the fast memory block specified by bits 1 and 2 for the user, limit the address space to that of a small user if 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 §2.19.

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 overflow traps if bit 22 is 1.

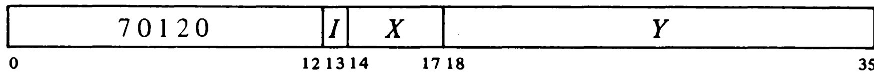
If either bit 0 or 18 is 1, invalidate all data in the associative memory. In other words set the Word Empty bit in each location to indicate that the rest of the word is meaningless and should not be used.

DATAI PAG, Data In, Paging

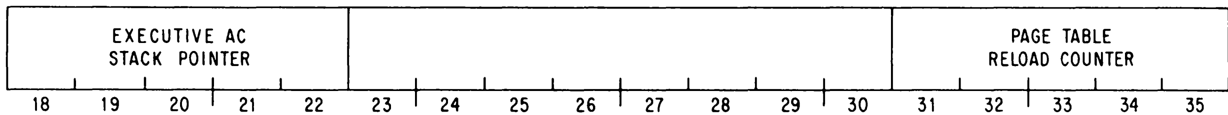


Read the status of the paging hardware into location *E*. The information read is the same as that supplied by a DATAO (bits 0 and 18 are 0).

CONO PAG, Conditions Out, Paging

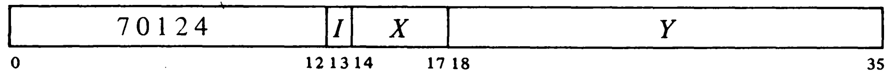


Load the executive stack pointer from bits 18–22 and the page table reload counter from bits 31–35 of the effective conditions *E* as shown.



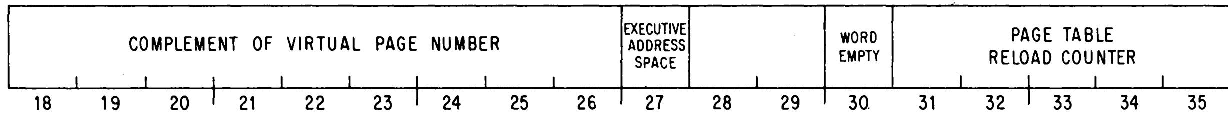
The executive stack pointer specifies a block of sixteen locations in the user process table by supplying the left five bits for a 9-bit address that references a location in the table; this function is used only for accessing stacked fast memory blocks in an instruction executed by an executive XCT [*see below*]. Loading the reload counter causes it to point to the specified location in the page table.

CONI PAG, Conditions In, Paging



This instruction also reads the processor serial number into bits 0–9 of location *E*.

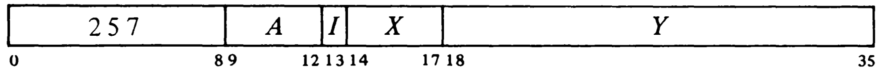
Read the page table reload counter and the contents of the location in the virtual page table specified by it into the right half of location *E* as shown.



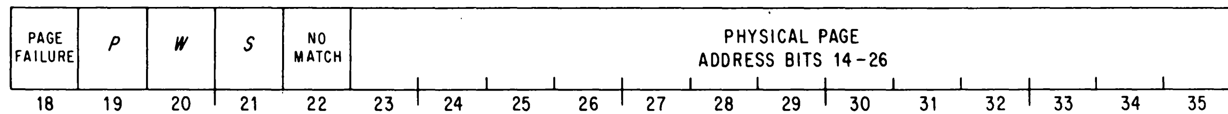
It is possible for the reload counter to change between the CONI and the CONO, so the CONI might read a different location than was selected by the CONO.

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.

MAP Map an Address



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, *ie* bits *P*, *W* and *S* in bits 19–21 and the physical page number in bits 23–35. Clear AC left.



These three instructions 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 an MAP that addresses the specified virtual page reads the contents of the physical-page part of that location.

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.

Executive XCT

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. In order to facilitate communication between Monitor and users, the XCT instruction allows the executive to execute instructions whose memory operand references can cross over the boundary between user and executive address spaces.

It is very important to note that the only difference between an instruction executed by an executive XCT and an instruction performed in normal circumstances is in the way the memory operand references are made. There is no difference in the XCT itself. Everything in the XCT is done in executive address space, and the instruction fetched by the XCT is fetched in executive space. Moreover, in the executed instruction all effective address calculation and accumulator references are in executive space. 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*.

Read the next four paragraphs very carefully (reading them two or three times is highly recommended).

Control over the special effects of the executed instructions is determined by the User In-out flag (whose implied meaning is confined to user mode) and bits 11 and 12 of the *A* portion of the XCT instruction word (in user mode *A* is ignored). If the *A* bits are both 0, the XCT acts as described in §2.9, and the executed instruction differs in no way from the normal case. But if these bits are not both 0, then some memory operand references are made to *user* virtual address space, where the type of reference is determined by the *A* bits and the type of memory is selected by User In-out. With this flag set, the *A* bits affect both core memory and fast memory references, whereas with User In-out clear, the *A* bits affect only fast memory references. For the memory operand references selected by User In-out, the effect of 1s in bits 11 and 12 is as follows: a 1 in bit 12 causes the executed instruction to perform all selected read and read-modify-write memory operand references to be performed in user virtual address space; a 1 in bit 11 causes all selected memory operand write references to be performed in user space; and 1s in both bits cause all types of selected memory operand references in the executed instruction to be performed in user space.

The meaning of user space is obvious in terms of core memory references, but not so for fast memory. When User In-out 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 by bits 11 and 12 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 User In-out is clear, all core memory references are in executive address space. Fast memory references of the types specified by bits 11 and 12 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,.

User Space Fast Memory References

User In-out	*User Fast Memory Block Selected*	
	Zero	*Nonzero*
1	Shadow area	Selected block
0	AC stack	AC stack

There is another flag that plays a role in the execution of instructions by an executive XCT. This is Disable Bypass, bit 0 of the PC word. When Disable Bypass is clear, a bypass in the logic allows an executed instruction to access the concealed user area from supervisor mode. With the flag set, an attempt to do this results in a page failure. Generally the new MUUO PC word would set this flag when the Monitor is being called from public mode, so the concealed area can be accessed only when such access is requested by the concealed program.

Individual Instruction Effects. The effects of execution by an executive XCT on different types of instructions is 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 an executive XCT not affected when executed by an executive XCT, but the first destroys any effect the second would otherwise have on a third instruction (in other words, a pair of executive XCTs 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).

♦ 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, *ie* the load or deposit.

Philosophy. The purpose of the executive XCT is to facilitate the handling of user requirements by the Monitor, but the selection made by User In-out of the references affected by the read and write bits is to allow the Monitor to make recursive calls to itself, *ie* to perform MUUOs in the process of carrying out an MUUO given by the user. Specifically the state of User In-out 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 User In-out 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 User In-out. 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 User In-out is left clear. The particular AC block used at any level is specified by the stack pointer. Hence the AC stack in the user process table is effectively a pushdown list 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.

This makes a different set of sixteen words available at each level using the same addresses.

EXAMPLE. Suppose that the Monitor has been called by an MUUO from the user (hence User In-out 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        ;Initialize pointer: from 0 to 342000
XCT     1,[BLT  T,342000]
```

and restores them with these two.

```
MOVSI   T,342000        ;From 342000 to 0
XCT     2,[BLT  T,17]
```

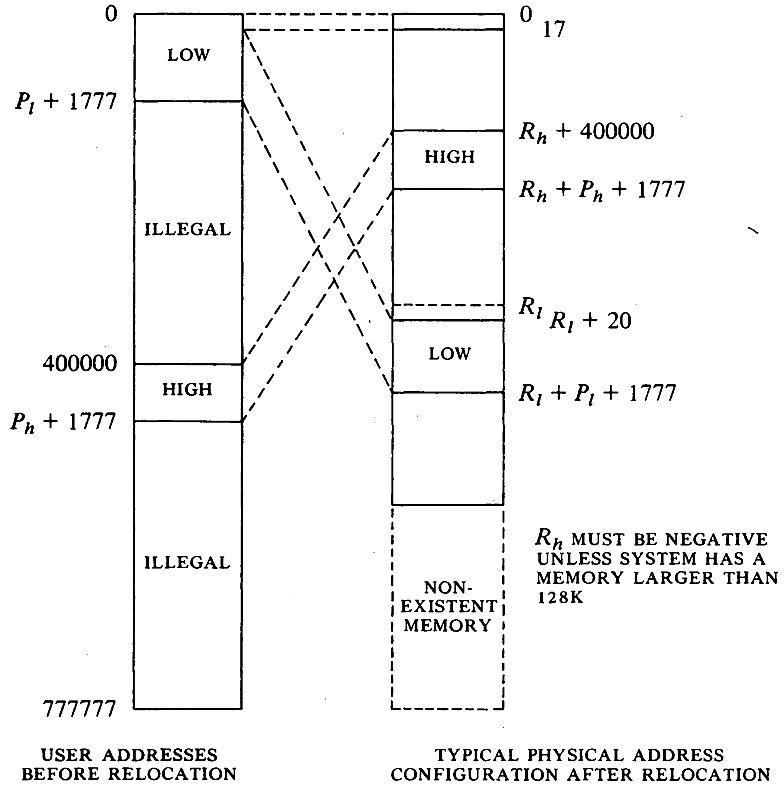
2.16 KA10 MODES

The KA10 has only user and executive modes and uses protection and relocation hardware.

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, *ie* 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

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.



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, *eg* 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. 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 (18–25) 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 (*ie* 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, *ie* more than half the maximum memory capacity ($P_l > 400000$), 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 channel assigned to the processor [§2.14].

User Programming. The user must observe the following rules when programming on a time shared basis. [*Refer to the Monitor manual for further information including use of the Monitor for input-output.*]

◆ Use addresses only within the assigned blocks for all purposes – retrieval of instructions, retrieval of addresses, storage or retrieval of operands. The low part contains locations with addresses from 0 to the maximum; the high part contains from the greater of 400000 or $P_l + 2000$ to the maximum. Either part can address the other.

The user can actually write any size program: the Monitor will assign enough core for his needs. Basically the user must write a sensible program; if he uses absolute addresses scattered all over memory his program cannot be run on a time shared basis with others.

◆ If the high part is write-protected, do not attempt to store anything in it. In particular do not execute a JSR or JSA into the high part.

◆ Use instruction codes 000 and 040–127 only in the manner prescribed in the Monitor manual.

◆ Unless User In-out is set do not give any IO instruction, HALT (JRST 4,) or JEN (JRST 12, (specifically JRST 10,)). The program can determine if User In-out is set by examining bit 6 of the PC word stored by JSR, JSP or PUSHJ.

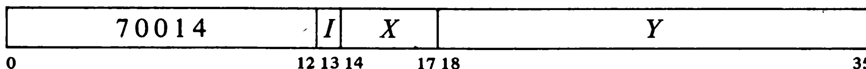
These instructions are illegal unless User In-out is set.

The user can give a JRSTF (JRST 2,) but a 0 in bit 5 of the PC 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 restrictions himself. Note that a 0 in bit 6 will clear User In-out, so a user can discard his own special privileges.

LUUOs (001–037) function normally and are relocated to addresses 40 and 41 in the low block [§2.10].

Monitor Programming. The Monitor must assign the core area for each user program, set up trap and interrupt locations, specify whether the user can give IO 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.

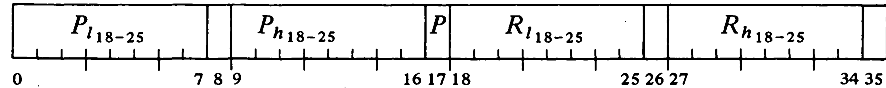
DATAO APR, Data Out, Arithmetic Processor



Load the protection and relocation registers from the contents of location

For a two part nonreentrant program, set $P = 0$. For a one-part nonreentrant program, make $P_h \leqslant P_l$. 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.

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.

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. 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 IO 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.

The trap locations are 140-141 and 160-161 in a second KA10 processor.

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.

2.17 REAL TIME CLOCK DK10

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 [§2.14].

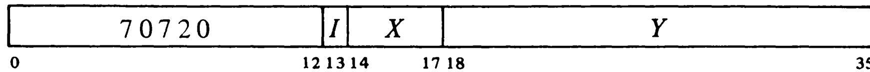
This processor option can be used to signal the end of a specified real time interval or to measure the real time taken by an event. With appropriate software the DK10 can easily be used to keep the time of day. The basic element in the clock is an 18-bit binary counter that is incremented repeatedly 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

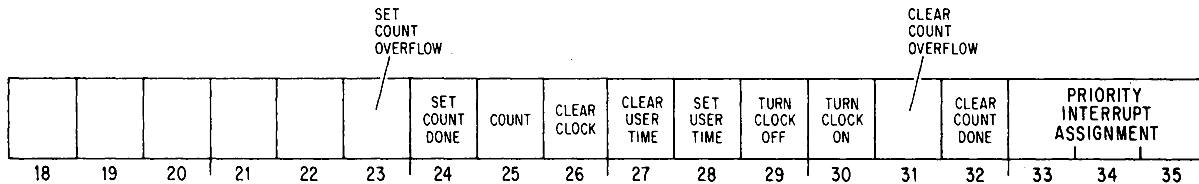
clear, the counter operates continuously; with User Time set, the counter stops while the processor is handling interrupts. Hence in the latter mode the clock discounts interrupt time and can be used to time user programs. In a system that contains two clocks, one can be used by the Monitor to time user 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.

CONO CLK, Conditions Out, Clock



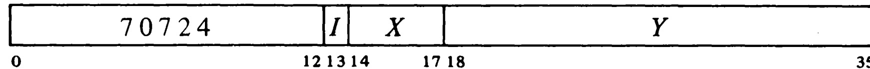
Assign the interrupt channel specified by bits 33–35 of the effective conditions E and perform the functions specified by bits 23–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



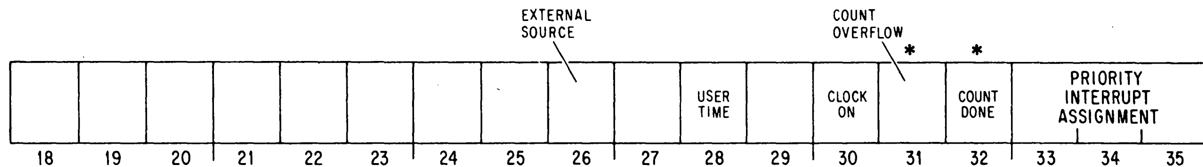
A 1 in bit 26 clears the clock counter and the Count Done, Count Overflow and User Time flags, turns off the clock, and dismisses the PI assignment (assigns zero). The effect of giving conflicting conditions is indeterminate.

A 1 in bit 25 increments the counter provided the clock is off (this is for maintenance only).

CONI CLK, Conditions In, Clock



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.



Notes.

***These bits request interrupts.**

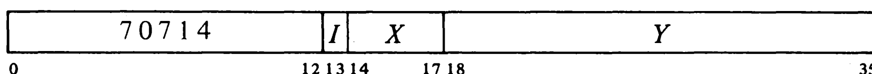
Interrupts are requested on the assigned channel by the setting of Count Overflow and Count Done.

26 The counter is connected to an external source (0 indicates the internal source is connected).

Note that to time a user properly, the Monitor must also compensate for any noninterrupt time taken from the user.

28 The counter cannot be incremented while an interrupt is being held or a request has been accepted and the channel is waiting for an interrupt to start.

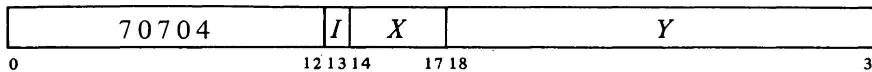
DATAO CLK, Data Out, Clock



The comparison of the counter against the interval register that follows every count is inhibited while this instruction is loading the register.

Load the contents of the right half of location E into the interval register.

DATAI CLK, Data In, Clock



The counter is always stable while being read, and any count held back is picked up immediately afterward.

Read the current contents of the clock counter into the right half of location E.

Following turnon the first count may occur at any time up to the full period of the source.

Remember that although a CONO need not affect the mode or the clock state, every CONO must renew the PI assignment.

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 channel. When the count reaches the specified interval, Count Done sets, requesting an interrupt on the assigned channel. 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.

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

$$T(C + nI)$$

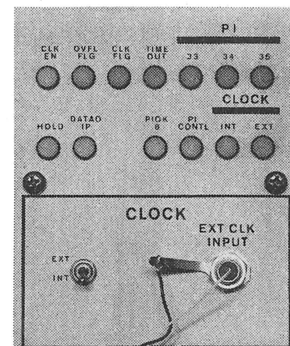
where T is the period of the source, n is the number of clock interrupts since the clock was started, and I is the interval selected by the program. To cause the clock to request an interrupt after $T \times n$ μs, where $n \leqslant 2^{18}$ and T is

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 and 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, 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 2½ seconds, which is octal 750220, is the most straightforward interval with the fewest interrupts. To interrupt every second the interval would be 303240.

Note that an error of .01% amounts to 8.64 seconds in 24 hours.

Operation. The KI10 clock, which is usually installed in a DECtape cabinet, has a small control panel mounted directly on the logic behind the cabinet doors. In the lower part of the panel is a switch for selecting the internal source or an external input from the BNC connector at the right. The external input must be supplied through a 100 ohm coaxial cable and must have a frequency no greater than 400 kHz; its triggering voltage change must be from −3 volts to ground. If the input is a pulse train, the minimum pulse width is 100 ns. If the input is a sequence of level changes, it must have a minimum low level (−3 volts) duration of 400 ns before each positive-going change, a rise time of 60 ns maximum, and a high level duration of 40 ns minimum.

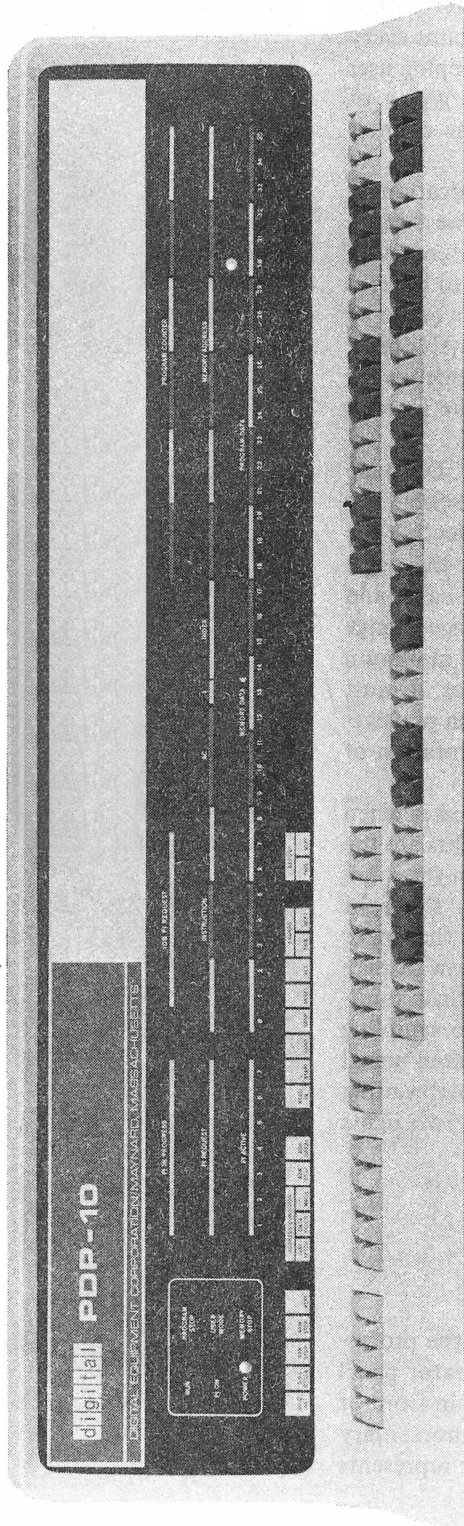


Clock Control Panel

The leftmost light in the upper row at the top of the panel indicates when the clock is on (*ie* when the counter is enabled). The next two lights are the Count Overflow and Count Done flags. TIME OUT indicates when the numbers in the interval register and the clock counter are identical – this light goes out as soon as either changes state. The remaining lights in the upper row are the PI assignment. The two lights at the left in the lower row display signals that synchronize the DATAI and DATAO to the clock so that counting is postponed while the counter is being read and there is no sampling while the interval is being loaded. PIOK8 is a processor-generated signal which indicates that there is no interrupt being held and no channel waiting for an interrupt; the next light is the User Time flag. The final two lights indicate the origin of the clock source.

2.18 KA10 OPERATION

Most of the controls and indicators used for normal operation of the processor and for program debugging are located on the console operator panel shown here. The indicators are on the vertical part of the panel; in front of them are two rows of two-position keys and switches (keys are momentary contact, switches are alternate action). A key or switch is on or represents a 1 when the front part is down.



The thirty-six switches in the front row and the eighteen at the right in the back row are respectively the data and address switches through which the operator can supply words and addresses for the program and for use in conjunction with the operating keys and switches. The correspondence of switches to bit positions is indicated by the numbers at the bottom row of lights. At the left end of the back row are ten operating switches, which supply continuous control levels to the processor. At their right are ten operating keys, which initiate or terminate operations in the processor. The names of the operating keys and switches appear on the vertical part of the panel below the lights.

Also of interest to the operator is the small panel shown opposite, which is located above the main panel at the left of the tape reader. The upper section of this panel contains a total hours meter and the margin-check controls. The lower section contains the power switch, speed controls for slowing down the program, switches to select the device for readin mode (lower part in represents a 1), and four additional operating switches. The normal position for these last four is with the upper part in; in this position FM ENB (fast memory enable) is on, the others are all off.

Indicators

When any indicator is lit the associated flipflop is 1 or the associated function is true. Some indicators display useful information while the processor is running, but many change too frequently and can be discussed only in terms of the information they display when the processor is stopped. The program can stop the processor only at the completion of the HALT instruction; the operator can stop it at the end of every instruction or memory reference, or for maintenance purposes, after every step in any operation that uses the shift counter (shifting, multiplication, division, byte manipulation).

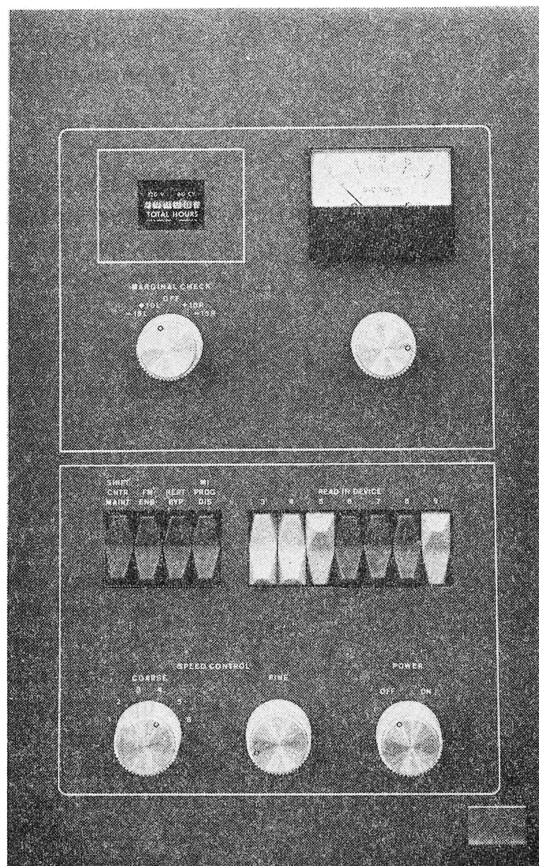
Of the long rows of lights at the right on the operator panel, the top row displays the contents of PC, the middle row displays the instruction being executed or just completed, and the bottom row are the memory indicators. The right half of the middle row displays MA, the left half displays IR [*see page 1-2*]. In an IO instruction the left three instruction lights are on, the remaining instruction lights and the left AC light are the device code, and the remaining AC lights complete the instruction code. The I, index and MA lights change with each indirect reference in an effective address calculation; at the end of an instruction I is always off.

Above the memory indicators appear two pairs of words, PROGRAM DATA and MEMORY DATA. If the triangular light beside the former pair is on, the indicators display a

word supplied by a DATAO PI,; if any other data is displayed the light beside MEMORY DATA is on instead. While the processor is running the physical addresses used for memory reference (the relocated address whenever relocation is in effect) are compared with the contents of the address switches. Whenever the two are equal the contents of the addressed location are displayed in the memory indicators. However, once the program loads the indicators, they can be changed only by the program until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key (see below).

The four sets of seven lights at the left display the state of the priority interrupt channels [*see pages 2-81 to 2-85*]. The PI ACTIVE lights indicate which channels are on. The IOB PI REQUEST lights indicate which channels are receiving request signals over the in-out bus; the PI REQUEST lights indicate channels on which the processor has accepted requests. Except in the case of a program-initiated interrupt, a REQUEST light can go on only if the corresponding ACTIVE light is on. The PI IN PROGRESS lights indicate channels on which interrupts are currently being held; the channel that is actually being serviced is the lowest-numbered one whose light is on. When a PROGRESS light goes on, the corresponding REQUEST goes off and cannot go on again until PROGRESS goes off when the interrupt is dismissed.

At the left end of the panel are a power light and these control indicators.



Above: Margin Check and Maintenance Panel
Opposite: Console Operator Panel

NOTE: If a REQUEST light stays on indefinitely with the associated PROGRESS light off and PC is static, check the PI CYC light on the indicator panel at the top of bay 2. If it is on, a faulty program has hung up the processor. Press STOP.

RUN
The processor is in normal operation with one instruction following another. When the light goes off, the processor stops.

PI ON
The priority interrupt system is active so interrupts can be started (this corresponds to CONI PI, bit 28).

PROGRAM STOP
IR now contains a HALT instruction. If RUN is off, MA displays an address one 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 presses the continue key).

If RUN and PROGRAM STOP are both on, the processor is probably in an indirect address loop. Press STOP.

USER MODE
The processor is in user mode (this corresponds to bit 5 of a PC word).

MEMORY STOP
The processor has stopped at a memory reference. This can be due to single cycle operation, satisfaction of an address condition selected at the console, reference to a nonexistent memory location, or detection of a parity error.

The remaining processor lights are on the indicator panels at the tops of the bays [*illustrated on page F2*]. Bay 2 displays AR, BR and MQ, the output of the AR adder, and the parity buffer which receives every word transmitted over the memory bus. The RL and PR lights at the lower right display the relocation and protection registers for the low part of the area assigned to a user program and the left eight bits of the relocated address for that part. The remaining lights are for maintenance.

The upper four rows on the bay 1 panel include the indicators for reader, punch and teletype, which are described in Chapter 3. The bottom row displays the information on the data lines in the IO bus. The AR lights at the upper right are the flags – FXU is Floating (exponent) Underflow, DCK is No Divide (divide check). OV COND is the condition that the 0 and 1 carries are different, *ie* the condition that indicates overflow. The First Part Done flag is BYF6 in the MISC lights in the top row; User In-out is IOT USER in the EX lights at the center of the panel. The CPA lights in the top row and the five lights under them at the left are the processor conditions – PDL OV is Pushdown (list) Overflow. The AS= lights in the middle row indicate when the (relocated) core memory address or the fast memory address is the same as the address switches. The remaining lights are for maintenance.

The panels on the memories are shown in Appendix F. These are almost exclusively for maintenance, and (as with most of the lights on the processor bays) if the operator must use them he should consult the maintenance manual and the flow charts. The ACTIVE lights indicate which processor currently has access to the memory.

Operating Keys

CAUTION

Never press two keys simultaneously as the processor may attempt to perform both functions at once.

Each key except STOP turns on one of the key indicators at the upper right on the bay 2 panel. These are for flipflops that allow the key functions to be repeated automatically and also allow certain of them to be synchronized to the processor time chain so they can be performed while the processor is running.

READ IN

If RUN is on, pressing this key has no effect.

Clear all IO devices and all processor flags including User; turn on the RIM light in the upper right on bay 1 and the KEY RDI light in the upper right

on bay 2. Execute DATAI *D*,0 where *D* is the device code specified by the readin device switches on the small panel at the left of the reader. Then execute a series of BLKI *D*,0 instructions until the left half of location 0 reaches zero, at which time turn off RIM and KEY RDI. Stop only if the single instruction switch is on; otherwise turn on RUN and execute the last word read as an instruction. [*For information on the data format refer to page 2-79.*]

The rightmost device switch is for bit 9 of the instruction and thus selects the least significant octal digit (which is always 0 or 4) in the device code.

Caution

Do not initiate any other key function while RIM is on. If read in must be stopped (*eg* because of a crumpled tape), press RESET (see below).

START

Load the contents of the address switches into PC, turn on RUN, and begin normal operation by executing the instruction at the location specified by PC.

This key function does not disturb the flags or the IO equipment; hence if USER MODE is lit a user program can be started.

If RUN is on, pressing this key has no effect.

CONT (Continue)

Turn on RUN (if it is off) and begin normal operation in the state indicated by the lights.

STOP

Turn off RUN so the processor stops before beginning the next instruction. Thus the processor usually stops at the end of the current instruction, which is displayed in the lights. However, if a key function that can be performed while RUN is on has been synchronized, the processor performs that function before stopping. In either case PC points to the next instruction.

If the processor does not reach the end of the instruction within 100 μs, inhibit further effective address calculation – it is assumed the processor is caught in an indirect addressing loop. Pressing CONT when the processor is stopped in an address loop causes it to start the same instruction over.

RESET

Clear all IO devices and clear the processor including all flags. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on duplicate the action of the STOP key before clearing.

If STOP will not stop the processor, pressing this key will.

XCT

Execute the contents of the data switches as an instruction without incrementing PC. If RUN is on, insert this instruction between two instructions in the program. Inhibit priority interrupts during its execution to guarantee that it will be finished.

If USER MODE is lit all user restrictions apply to an instruction executed from the console.

Note that an instruction executed from the console can alter the processor state just like any instruction in the program: it can change PC by jumping or skipping, alter the flags, or even cause a nonexistent-memory stop.

NOTE

The remaining key functions all reference memory. They use an absolute address and all of memory is available to them; in other words protection and relocation are not in effect even if USER MODE is lit. However they can set such flags as Address Break and Nonexistent Memory.

EXAMINE THIS
Display the contents of the address switches in the MA lights and the contents of the location specified by the address switches in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on, insert this function between two instructions in the program.

If RUN is on, pressing this key has no effect.

EXAMINE NEXT
Add 1 to the address displayed in the MA lights and display the contents of the location specified by the incremented address in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA).

DEPOSIT
Deposit the contents of the data switches in the location specified by the address switches. Display the address in the MA lights and the word deposited in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on, insert this function between two instructions in the program.

If RUN is on, pressing this key has no effect.

DEPOSIT NEXT
Add 1 to the address displayed in the MA lights and deposit the contents of the data switches in the location specified by the incremented address. Display the word deposited in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA).

Operating Switches

Whenever the processor references memory at the location specified by the address switches (relocated if USER MODE is on), the contents of that location are displayed in the memory indicators (unless the light beside PROGRAM DATA is on). The group of five switches at the left of the keys allows the operator to make the processor halt or request an interrupt when

reference is made to the specified location *in core memory* for a particular purpose (no action is produced by fast memory reference). The purpose is selected by the three address condition switches. INST FETCH selects the condition that access is 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. DATA FETCH selects access for retrieval of an operand (other than in an XCT). WRITE selects access for writing in memory. Whenever reference to the specified location satisfies any selected address condition, the processor performs the action selected by the other two switches. ADR STOP halts the processor with MEMORY STOP on (PC points to the instruction that was being executed, or if the MC WR light on bay 2 is on, PC may point to the one following it); ADR BREAK turns on the CPA ADR BRK light (Address Break flag, CONI APR, bit 21) on bay 1, requesting an interrupt on the processor channel.

AC and index register references can be included by turning off the FM ENB switch (see below).

The description of each switch relates the action it produces while it is on.

SING INST
Whenever the processor is placed in operation, clear RUN so that it stops at the end of the first instruction. Hence the operator can step through a program one instruction at a time, by pressing START for the first one and CONT for subsequent ones. Each time the processor stops, the lights display the same information as when STOP is pressed.

CLK FLAG (Clock flag) on bay 1 is held off to prevent clock interrupts while SING INST is on. Otherwise interrupts would occur at a faster rate than the instructions.

SING INST will not stop the processor if a hangup prevents it from getting to the end of an instruction. Use STOP or RESET.

SING CYCLE
Whenever the processor is placed in operation, stop it with MEMORY STOP on at the end of the first *core memory* reference. Hence the operator can step through a program one memory reference at a time, by pressing START for the first one and CONT for subsequent ones. To determine what information is displayed in the lights, consult the flow charts.

To stop at AC and index register references, turn off the FM ENB switch (see below).

PAR STOP
Stop with MEMORY STOP on at the end of any memory reference in which even parity is detected in a word read. A parity stop is indicated by the following: CPA PAR ERR (Parity Error flag) on bay 1 is on; and among the PAR lights in the bottom row on bay 2, IGN (ignore parity) and ODD are off, STOP is on, and BIT displays the parity bit for the word in the parity buffer at the left.

If IGN is on (it displays a signal from the memory), parity errors are not detected and no stop can occur.

NXM STOP
Stop with MEMORY STOP on if a memory reference is attempted but the memory does not respond within 100 μs. This type of stop is indicated by CPA NXM FLAG (Nonexistent Memory flag) on bay 1 being on.

REPT
If any key (except STOP) is pressed, then every time the key function is finished, wait a period of time determined by the setting of the speed control and repeat the given key function. If CONT is pressed and no switch is on that would stop the program (*eg* SING INST, SING CYCLE), then continue following the repeat delay whenever a HALT instruction is executed. Continue to repeat the key function until RESET is pressed or REPT is turned off.

The key function is repeated once after REPT is turned off, but this is noticeable only with very long repeat delays.

The end of a key function is equivalent to completion of all processor operations associated with the function only for read in, examine, examine next, deposit, and deposit next. In other cases the processor continues in operation. *Eg* the execute function is finished once the instruction to be executed is set up internally, but the processor then executes that instruction. Hence when using speed range 6, the operator must be careful not to allow the key function to restart before the processor is really finished.

The speed control includes a six-position switch that selects the delay range and a potentiometer for fine adjustment within the range. Delay ranges are as follows.

Position	*Range*
1	270 ms to 5.4 seconds
2	38 ms to 780 ms
3	3.9 ms to 78 ms
4	390 μs to 7.8 ms
5	27 μs to 540 μs
6	2.2 μs to 44 μs

MI PROG DIS
Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA) and inhibit the program from displaying any information in the memory indicators. The indicators will thus continually display the contents of locations selected from the console.

REPT BYP
If REPT is on, trigger the repeat delay at the *beginning* of the key function. Hence the function is repeated even if it does not run to completion.

FM ENB
This switch is left on for normal operation with a fast memory. Turning it *off* (lower part in) substitutes the first sixteen core locations for the fast memory. The switch is left off if there is no fast memory, and it can be used to allow stopping or breaking at fast memory references.

SHIFT CNTR MAINT
Stop before each step in any shift operation. Pressing CONT resumes the operation. Once a shift has been stopped, the processor will continue to stop at each step throughout the rest of the given shift operation even if the switch is turned off.

At the right end of panel 1J behind the bay doors are two toggle switches. FP TRP causes the floating point and byte manipulation instructions (codes 130–177) to trap to locations 60-61. MA TRP OFFSET moves the trap and interrupt locations to 140–161 for a second processor connected to the same memory.

Inside each memory bay are switches for selecting the memory number and interleaving memories. Also in the memory are a power switch, a restart pushbutton, and a switch for single step operation (these three are located on the indicator panel for the MB10 memory).

NOTE
Information on the KI10 operation is not available at this time.

Appendices

APPENDIX A

INSTRUCTION AND DEVICE MNEMONICS

The illustration on the next page shows the derivation of the instruction mnemonics. The two tables following it list all instruction mnemonics and their octal codes both numerically and alphabetically. When two mnemonics are given for the same octal code, the first is the preferred form, but the assembler does recognize the second. For completeness, the table includes the MUUOs (indicated by an asterisk) that are recognized by MACRO for communication with the DECsystem-10 Time Sharing Monitor. A double dagger (‡) indicates a KI10 instruction code that is unassigned in the KA10.

In-out device codes are included only in the alphabetic listing and are indicated by a dagger (†). Following the tables is a chart that lists the devices with their mnemonic and octal codes and DEC option numbers for both PDP-10 and PDP-6. A device mnemonic ending in the numeral 2 is the recommended form for the second of a given device, but such codes are not recognized by MACRO – they must be defined by the user.

Beginning on page A11 is a list of all instructions showing their actions in symbolic form.

A2 MNEMONICS

MOV {E / e Negative / e Magnitude / e Swapped} — {to AC / Immediate to AC / to Memory / to Self}

Half word {Right / Left} to {Right / Left} {no effect / Ones / Zeros / Extend sign} — {to AC / Immediate to AC / to Memory / to Self}

BLock Transfer

EXCHange AC and memory

{use present pointer / Increment pointer} and {LoaD Byte into AC / DePosit Byte in memory}

Increment Byte Pointer

{PUSH down / POP up} {~ / and Jump}

SET to {Zeros / Ones / Ac / Memory / Complement of Ac / Complement of Memory} — to {AC / AC Immediate / Memory / Both}

{AND / inclusive OR} {~ / with Complement of Ac / with Complement of Memory / Complements of Both} — to {AC / AC Immediate / Memory / Both}

{Inclusive OR / eXclusive OR / EQuiValence} — to {AC / AC Immediate / Memory / Both}

{SKIP if memory / JUMP if AC} — if {never / Less / Equal / Less or Equal / Always / Greater / Greater or Equal / Not equal}

{Add One to / Subtract One from} {memory and Skip / AC and Jump} if — {never / Less / Equal / Less or Equal / Always / Greater / Greater or Equal / Not equal}

Compare Ac {Immediate / with Memory} and skip if AC — {never / Less / Equal / Less or Equal / Always / Greater / Greater or Equal / Not equal}

Add One to Both halves of AC and Jump if {Positive / Negative}

{Arithmetic SHift / Logical SHift / ROTate} {~ / Combined}

{ADD / SUBtract / MULtiply / Integer MULtiply / DIVide / Integer DIVide} — {~ / Immediate / to Memory / to Both}

{Floating AdD / Floating SuBtract / Floating MultiPly / Floating DiVide} — {~ / Long / to Memory / to Both}; and Round — {~ / Immediate / to Memory / to Both}

Floating SCale

Double Floating Negate

Unnormalized Floating Add

FIX

FIX and Round

FLoaT and Round

Double Floating AdD
Double Floating SuBtract
Double Floating MultiPly
Double Floating DiVide

Double MOV {E / e Negative} {~ / to Memory}

Jump {to SubRoutine / and Save Pc / and Save Ac / and Restore Ac / if Find First One / on Flag and CLear it / on OVerflow (JFCL 10,) / on CaRrY 0 (JFCL 4,) / on CaRrY 1 (JFCL 2,) / on CaRrY (JFCL 6,) / on Floating OVerflow (JFCL 1,) / and ReSTore / and ReSTore Flags (JRST 2,) / and ENable PI channel (JRST 12,)}

HALT (JRST 4,)

PORTAL (JRST 1,)

eXeCuTe

{DATA / BLocK} {In / Out}

CONditions {In / Out}; in and Skip if {all masked bits Zero / some masked bit One}

Test AC {with Direct mask / with Swapped mask / Right with E / Left with E} {No modification / set masked bits to Zeros / set masked bits to Ones / Complement masked bits} and skip {never / if all masked bits Equal 0 / if Not all masked bits equal 0 / Always}

INSTRUCTION MNEMONICS

NUMERIC LISTING

Code	Mnemonic
000	ILLEGAL
001 : : 037	LUUO'S
040	*CALL
041	*INIT
042 043 044 045 046	RESERVED FOR SPECIAL MONITORS
047	*CALLI
050	*OPEN
051	*TTCALL
052 053 054	RESERVED FOR DEC
055	*RENAME
056	*IN
057	*OUT
060	*SETSTS
061	*STATO
062	*STATUS
062	*GETSTS
063	*STATZ
064	*INBUF
065	*OUTBUF
066	*INPUT
067	*OUTPUT
070	*CLOSE
071	*RELEAS
072	*MTAPE
073	*UGETF
074	*USETI
075	*USETO
076	*LOOKUP
077	*ENTER
100	*UJEN
101	
102	
103	
104	
105	
106	
107	
110	‡DFAD
111	‡DFSB
112	‡DFMP
113	‡DFDV
114	
115	
116	
117	
120	‡DMOVE
121	‡DMOVN
122	‡FIX
123	
124	‡DMOVEM
125	‡DMOVNM
126	‡FIXR
127	‡FLTR
130	UFA
131	DFN
132	FSC
133	IBP
134	ILDB
135	LDB
136	IDPB
137	DPB
140	FAD
141	FADL
142	FADM
143	FADB
144	FADR
145	FADRI
146	FADRM
147	FADRB
150	FSB
151	FSBL
152	FSBM
153	FSBB
154	FSBR
155	FSBRI
156	FSBRM
157	FSBRB
160	FMP
161	FMPL
162	FMPM
163	FMPB
164	FMPR
165	FMPRI
166	FMPRM
167	FMPRB
170	FDV
171	FDVL
172	FDVM
173	FDVB
174	FDVR
175	FDVRI
176	FDVRM
177	FDVRB
200	MOVE
201	MOVEI
202	MOVEM
203	MOVES
204	MOVS
205	MOVSI
206	MOVSM
207	MOVSS
210	MOVN
211	MOVNI
212	MOVNM
213	MOVNS
214	MOVM
215	MOVMI
216	MOVMM
217	MOVMS
220	IMUL
221	IMULI
222	IMULM
223	IMULB
224	MUL
225	MULI
226	MULM
227	MULB
230	IDIV
231	IDIVI
232	IDIVM
233	IDIVB
234	DIV
235	DIVI

A4

MNEMONICS

Code	Mnemonic
236	DIVM
237	DIVB
240	ASH
241	ROT
242	LSH
243	JFFO
244	ASHC
245	ROTC
246	LSHC
247	
250	EXCH
251	BLT
252	AOBJP
253	AOBJN
254	JRST
25404	PORTAL
25410	JRSTF
25420	HALT
25450	JEN
255	JFCL
25504	JFOV
25510	JCRY1
25520	JCRY0
25530	JCRY
25540	JOV
256	XCT
257	‡MAP
260	PUSHJ
261	PUSH
262	POP
263	POPJ
264	JSR
265	JSP
266	JSA
267	JRA
270	ADD
271	ADDI
272	ADDM
273	ADDB
274	SUB
275	SUBI
276	SUBM
277	SUBB
300	CAI
301	CAIL
302	CAIE
303	CAILE
304	CAIA
305	CAIGE
306	CAIN
307	CAIG
310	CAM
311	CAML
312	CAME
313	CAMLE
314	CAMA
315	CAMGE
316	CAMN
317	CAMG
320	JUMP
321	JUMPL
322	JUMPE
323	JUMPLE
324	JUMPA
325	JUMPGE
326	JUMPN
327	JUMPG
330	SKIP
331	SKIPL
332	SKIPE
333	SKIPLE
334	SKIPA
335	SKIPGE
336	SKIPN
337	SKIPG
340	AOJ
341	AOJL
342	AOJE
343	AOJLE
344	AOJA
345	AOJGE
346	AOJN
347	AOJG
350	AOS
351	AOSL
352	AOSE
353	AOSLE
354	AOSA
355	AOSGE
356	AOSN
357	AOSG
360	SOJ
361	SOJL
362	SOJE
363	SOJLE
364	SOJA
365	SOJGE
366	SOJN
367	SOJG
370	SOS
371	SOSL
372	SOSE
373	SOSLE
374	SOSA
375	SOSGE
376	SOSN
377	SOSG
400	SETZ
400	CLEAR
401	SETZI
401	CLEARI
402	SETZM
402	CLEARM
403	SETZB
403	CLEARB
404	AND
405	ANDI
406	ANDM
407	ANDB
410	ANDCA
411	ANDCAI
412	ANDCAM
413	ANDCAB
414	SETM
415	SETMI
416	SETMM
417	SETMB
420	ANDCM
421	ANDCMI
422	ANDCMM
423	ANDCMB
424	SETA
425	SETAI
426	SETAM
427	SETAB
430	XOR
431	XORI
432	XORM
433	XORB
434	IOR
434	OR
435	IORI
435	ORI
436	IORM
436	ORM
437	IORB
437	ORB

NUMERIC LISTING A5

440	ANDCB	521	HLLOI	602	TRNE
441	ANDCBI	522	HLLOM	603	TLNE
442	ANDCBM	523	HLLOS	604	TRNA
443	ANDCBB	524	HRLO	605	TLNA
444	EQV	525	HRLOI	606	TRNN
445	EQVI	526	HRLOM	607	TLNN
446	EQVM	527	HRLOS	610	TDN
447	EQVB	530	HLLE	611	TSN
450	SETCA	531	HLLEI	612	TDNE
451	SETCAI	532	HLLEM	613	TSNE
452	SETCAM	533	HLLES	614	TDNA
453	SETCAB	534	HRLE	615	TSNA
454	ORCA	535	HRLEI	616	TDNN
455	ORCAI	536	HRLEM	617	TSNN
456	ORCAM	537	HRLES	620	TRZ
457	ORCAB	540	HRR	621	TLZ
460	SETCM	541	HRRI	622	TRZE
461	SETCMI	542	HRRM	623	TLZE
462	SETCMM	543	HRRS	624	TRZA
463	SETCMB	544	HLR	625	TLZA
464	ORCM	545	HLRI	626	TRZN
465	ORCMI	546	HLRM	627	TLZN
466	ORCMM	547	HLRS	630	TDZ
467	ORCMB	550	HRRZ	631	TSZ
470	ORCB	551	HRRZI	632	TDZE
471	ORCBI	552	HRRZM	633	TSZE
472	ORCBM	553	HRRZS	634	TDZA
473	ORCBB	554	HLRZ	635	TSZA
474	SETO	555	HLRZI	636	TDZN
475	SETOI	556	HLRZM	637	TSZN
476	SETOM	557	HLRZS	640	TRC
477	SETOB	560	HRRO	641	TLC
500	HLL	561	HRROI	642	TRCE
501	HLLI	562	HRROM	643	TLCE
502	HLLM	563	HRROS	644	TRCA
503	HLLS	564	HLRO	645	TLCA
504	HRL	565	HLROI	646	TRCN
505	HRLI	566	HLROM	647	TLCN
506	HRLM	567	HLROS	650	TDC
507	HRLS	570	HRRE	651	TSC
510	HLLZ	571	HRREI	652	TDCE
511	HLLZI	572	HRREM	653	TSCE
512	HLLZM	573	HRRES	654	TDCA
513	HLLZS	574	HLRE	655	TSCA
514	HRLZ	575	HLREI	656	TDCN
515	HRLZI	576	HLREM	657	TSCN
516	HRLZM	577	HLRES	660	TRO
517	HRLZS	600	TRN	661	TLO
520	HLLO	601	TLN	662	TROE

A6 MNEMONICS

663	TLOE	673	TSOE	70010	BLKO
664	TROA	674	TDOA	70014	DATAO
665	TLOA	675	TSOA	70020	CONO
666	TRON	676	TDON	70024	CONI
667	TLON	677	TSON	70030	CONSZ
670	TDO	70000	BLKI	70034	CONSO
671	TSO	70004	DATAI		
672	TDOE	70004	RSW		

INSTRUCTION MNEMONICS

ALPHABETIC LISTING

†ADC	024	AOSA	354	†CDP	110
ADD	270	AOSE	352	†CDR	114
ADDB	273	AOSG	357	CLEAR	400
ADDI	271	AOSGE	355	CLEARB	403
ADDM	272	AOSL	351	CLEARI	401
AND	404	AOSLE	353	CLEARM	402
ANDB	407	AOSN	356	†CLK	070
ANDCA	410	†APR	000	*CLOSE	070
ANDCAB	413	ASH	240	CONI	70024
ANDCAI	411	ASHC	244	CONO	70020
ANDCAM	412	BLKI	70000	CONSO	70034
ANDCB	440	BLKO	70010	CONSZ	70030
ANDCBB	443	BLT	251	†CPA	000
ANDCBI	441	CAI	300	†CR	150
ANDCBM	442	CAIA	304	DATAI	70004
ANDCM	420	CAIE	302	DATAO	70014
ANDCMB	423	CAIG	307	†DC	200
ANDCMI	421	CAIGE	305	†DCSA	300
ANDCMM	422	CAIL	301	†DCSB	304
ANDI	405	CAILE	303	‡DFAD	110
ANDM	406	CAIN	306	‡DFDV	113
AOBJN	253	*CALL	040	‡DFMP	112
AOBJP	252	*CALLI	047	DFN	131
AOJ	340	CAM	310	‡DFSB	111
AOJA	344	CAMA	314	†DIS	130
AOJE	342	CAME	312	DIV	234
AOJG	347	CAMG	317	DIVB	237
AOJGE	345	CAMGE	315	DIVI	235
AOJL	341	CAML	311	DIVM	236
AOJLE	343	CAMLE	313	†DLB	060
AOJN	346	CAMN	316	†DLC	064
AOS	350	†CCI	014	†DLS	240

‡DMOVE	120	FSBRB	157	HRLS	507
‡DMOVEM	124	FSBRI	155	HRLZ	514
‡DMOVN	121	FSBRM	156	HRLZI	515
‡DMOVNM	125	FSC	132	HRLZM	516
DPB	137	*GETSTS	062	HRLZS	517
†DPC	250	HALT	25420	HRR	540
†DSI	464	HLL	500	HRRE	570
†DSK	170	HLLE	530	HRREI	571
†DSS	460	HLLEI	531	HRREM	572
†DTC	320	HLLEM	532	HRRES	573
†DTS	324	HLLES	533	HRRI	541
*ENTER	077	HLLI	501	HRRM	542
EQV	444	HLLM	502	HRRO	560
EQVB	447	HLLO	520	HRROI	561
EQVI	445	HLLOI	521	HRROM	562
EQVM	446	HLLOM	522	HRROS	563
EXCH	250	HLLOS	523	HRRS	543
FAD	140	HLLS	503	HRRZ	550
FADB	143	HLLZ	510	HRRZI	551
FADL	141	HLLZI	511	HRRZM	552
FADM	142	HLLZM	512	HRRZS	553
FADR	144	HLLZS	513	IBP	133
FADRB	147	HLR	544	IDIV	230
FADRI	145	HLRE	574	IDIVB	233
FADRM	146	HLREI	575	IDIVI	231
FDV	170	HLREM	576	IDIVM	232
FDVB	173	HLRES	577	IDPB	136
FDVL	171	HLRI	545	ILDB	134
FDVM	172	HLRM	546	IMUL	220
FDVR	174	HLRO	564	IMULB	223
FDVRB	177	HLROI	565	IMULI	221
FDVRI	175	HLROM	566	IMULM	222
FDVRM	176	HLROS	567	*IN	056
‡FIX	122	HLRS	547	*INBUF	064
‡FIXR	126	HLRZ	554	*INIT	041
‡FLTR	127	HLRZI	555	*INPUT	066
FMP	160	HLRZM	556	IOR	434
FMPB	163	HLRZS	557	IORB	437
FMPL	161	HRL	504	IORI	435
FMPM	162	HRLE	534	IORM	436
FMPR	164	HRLEI	535	JCRY	25530
FMPRB	167	HRLEM	536	JCRY0	25520
FMPRI	165	HRLES	537	JCRY1	25510
FMPRM	166	HRLI	505	JEN	25460
FSB	150	HRLM	506	JFCL	255
FSBB	153	HRLO	524	JFFO	243
FSBL	151	HRLOI	525	JFOV	25504
FSBM	152	HRLOM	526	JOV	25540
FSBR	154	HRLOS	527	JRA	267

A8

MNEMONICS

Mnemonic	Code
JRST	254
JRSTF	25410
JSA	266
JSP	265
JSR	264
JUMP	320
JUMPA	324
JUMPE	322
JUMPG	327
JUMPGE	325
JUMPL	321
JUMPLE	323
JUMPN	326
LDB	135
*LOOKUP	076
†LPT	124
LSH	242
LSHC	246
‡MAP	257
†MDF	260
MOVE	200
MOVEI	201
MOVEM	202
MOVES	203
MOVM	214
MOVMI	215
MOVMM	216
MOVMS	217
MOVN	210
MOVNI	211
MOVNM	212
MOVNS	213
MOVS	204
MOVSI	205
MOVSM	206
MOVSS	207
*MTAPE	072
†MTC	220
†MTM	230
†MTS	224
MUL	224
MULB	227
MULI	225
MULM	226
*OPEN	050
OR	434
ORB	437
ORCA	454
ORCAB	457
ORCAI	455
ORCAM	456
ORCB	470
ORCBB	473
ORCBI	471
ORCBM	472
ORCM	464
ORCMB	467
ORCMI	465
ORCMM	466
ORI	435
ORM	436
*OUT	057
*OUTBUF	065
*OUTPUT	067
†PAG	010
†PI	004
†PLT	140
POP	262
POPJ	263
PORTAL	25404
†PTP	100
†PTR	104
PUSH	261
PUSHJ	260
*RELEAS	071
*RENAME	055
ROT	241
ROTC	245
RSW	70004
SETA	424
SETAB	427
SETAI	425
SETAM	426
SETCA	450
SETCAB	453
SETCAI	451
SETCAM	452
SETCM	460
SETCMB	463
SETCMI	461
SETCMM	462
SETM	414
SETMB	417
SETMI	415
SETMM	416
SETO	474
SETOB	477
SETOI	475
SETOM	476
*SETSTS	060
SETZ	400
SETZB	403
SETZI	401
SETZM	402
SKIP	330
SKIPA	334
SKIPE	332
SKIPG	337
SKIPGE	335
SKIPL	331
SKIPLE	333
SKIPN	336
SOJ	360
SOJA	364
SOJE	362
SOJG	367
SOJGE	365
SOJL	361
SOJLE	363
SOJN	366
SOS	370
SOSA	374
SOSE	372
SOSG	377
SOSGE	375
SOSL	371
SOSLE	373
SOSN	376
*STATO	061
*STATUS	062
*STATZ	063
SUB	274
SUBB	277
SUBI	275
SUBM	276
TDC	650
TDCA	654
TDCE	652
TDCN	656
TDN	610
TDNA	614
TDNE	612
TDNN	616
TDO	670
TDOA	674
TDOE	672
TDON	676

TDZ	630
TDZA	634
TDZE	632
TDZN	636
TLC	641
TLCA	645
TLCE	643
TLCN	647
TLN	601
TLNA	605
TLNE	603
TLNN	607
TLO	661
TLOA	665
TLOE	663
TLON	667
TLZ	621
TLZA	625
TLZE	623
TLZN	627
†TMC	340
†TMS	344
TRC	640
TRCA	644
TRCE	642
TRCN	646
TRN	600
TRNA	604
TRNE	602
TRNN	606
TRO	660
TROA	664
TROE	662
TRON	666
TRZ	620
TRZA	624
TRZE	622
TRZN	626
TSC	651
TSCA	655
TSCE	653
TSCN	657
TSN	611
TSNA	615
TSNE	613
TSNN	617
TSO	671
TSOA	675
TSOE	673
TSON	677
TSZ	631
TSZA	635
TSZE	633
TSZN	637
*TTCALL	051
UFA	130
*UGETF	073
*UJEN	100
*USETI	074
*USETO	075
†UTC	210
†UTS	214
XCT	256
XOR	430
XORB	433
XORI	431
XORM	432

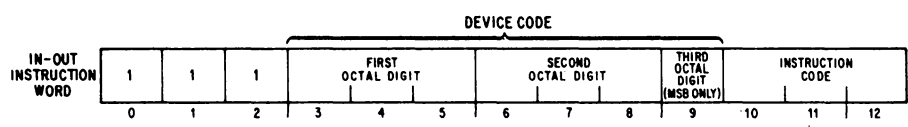
FIRST OCTAL DIGIT \ SECOND AND THIRD OCTAL DIGITS	00	04	10	14	20	24	30	34	40	44	50	54	60	64	70	74
0	6,10 APR CPA CENTRAL PROCESSOR	6,10 PI PRIORITY INTERRUPT	10 PAG* KI10 PAGING	10 DA10 CCI PDP-8,9 INTERFACE	10 DA10 CCI2 PDP-8,9 INTERFACE	10 AD10 ADC ANALOG-DIGITAL CONVERTER	10 AD10 ADC2 ANALOG-DIGITAL CONVERTER						10 DLB PDP-11 DATA LINK	DL10 DLC	10 DK10 CLK REAL TIME CLOCK	10 DK10 CLK2 REAL TIME CLOCK
1	6 761 10 PTP PAPER TAPE PUNCH	6 760 10 PTR PAPER TAPE READER	10 CP10 CDP CARD PUNCH	6 461 CDR CARD READER	6 626 10 TTY CONSOLE TELETYPE	6 646 10 LP10 LPT LINE PRINTER	6,10 340 10 VP10 DIS DISPLAY	6,10 340 10 VP10 DIS2 DISPLAY	10 XY10 PLT PLOTTER	10 XY10 PLT2 PLOTTER	10 CR10 CR CARD READER	10 CR10 CR2 CARD READER	10 DLB2† PDP-11 DATA LINK	DL10 DLC2	10 RC10 DSK SMALL DISK	10 RC10 DSK2 SMALL DISK
2	6 136 DC DATA CONTROL	6 136 DC2 DATA CONTROL	6 UTC DECTAPE	551 UTS	6 MTC MAGNETIC TAPE	MTS	516 MTM	6 646 10 LP10 LPT2 LINE PRINTER	10 DC10 DLS DATA LINE SCANNER	10 DC10 DLS2 DATA LINE SCANNER	10 RP10 DPC DISK PACK SYSTEM	10 RP10 DPC2 DISK PACK SYSTEM	10 RP10 DPC3 DISK PACK SYSTEM	10 RP10 DPC4 DISK PACK SYSTEM	6 270 DF DISK FILE	
3	6 DCSA DATA COMMUNICATION	630 DCSB			10 DTC DECTAPE	TD10 DTS	10 DTC2 DECTAPE	TD10 DTS2	10 TMC MAGNETIC TAPE	TM10 TMS	10 TMC2 MAGNETIC TAPE	TM10 TMS2				
4													10 DSS SINGLE SYNCHRONOUS LINE UNIT	DS10 DSI	10 DSS2 SINGLE SYNCHRONOUS LINE UNIT	DS10 DSI2
5																
6																
7																

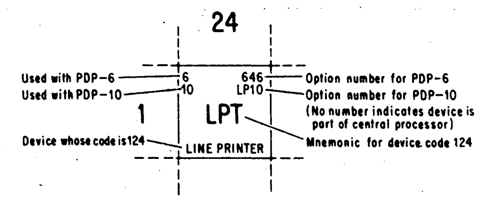
CODES IN THIS SECTION RESERVED FOR USER SPECIAL DEVICES

KI10 UNRESTRICTED CODES RESERVED FOR USERS

KI10 UNRESTRICTED CODES RESERVED FOR DEC

*DRUM PROCESSOR IN PDP-6
†PDP-7,8 INTERFACE IN PDP-6





DEVICE MNEMONICS

ALGEBRAIC REPRESENTATION

The remaining pages of this Appendix list, in symbolic form, the actual operations performed by the instructions. The grouping, as given below, differs slightly from that used in Chapter 2.

Boolean	A13	In-out	A17
Byte manipulation	A14	Program control	A17
Fixed point arithmetic	A14	Pushdown list	A17
Floating point arithmetic	A14	Shift and rotate	A17
Full word data transmission	A15	Test, arithmetic	A18
Half word data transmission	A16	Test, logical	A19

The terminology and notation used also vary somewhat from that in the body of the manual, as follows.

AC — The accumulator address in bits 9–12 of the instruction word (represented by A in the instruction descriptions).

AC+1 — The address one greater than AC, except that AC+1 is 0 if AC is 17.

E — The result of the effective address calculation. E is eighteen bits when used as an address, half word operand, mask or output conditions, but is a signed 9-bit quantity when used as a scale factor or a shift number.

E+1 — The address one greater than E, except that E+1 is 0 if E is 777777.

PC — The 18-bit program counter.

(X) — The word contained in register X.

$(X)_L$ — The left half of (X).

$(X)_R$ — The right half of (X).

$(X)_S$ — The word contained in X with its left and right halves swapped.

A_n — The value of bit n of the quantity A.

A,B — A 36-bit word with the 18-bit quantity A in its left half and the 18-bit quantity B in its right half (either A or B may be 0).

(X,Y) — The contents of registers X and Y concatenated into a double word operand.

$((X))$ — The word contained in the register addressed by (X), *ie* addressed by the word in register X.

$A \rightarrow B$ — The quantity A replaces the quantity B (A and B may be half words, full words or double words). *Eg*

$$(AC) + (E) \rightarrow (AC)$$

means the word in accumulator AC plus the word in memory location E replaces the word in AC.

(AC) (E) — The word in AC and the word in E.

$\wedge \vee \forall \sim$ — The Boolean operators AND, inclusive OR, exclusive OR, and complement (logical negation).

$+ - \times \div ||$ The arithmetic operators for addition, negation or subtraction, multiplication, division, and absolute value (magnitude).

Square brackets are used occasionally for grouping. With respect to the values of their terms, the equations for a given instruction are in chronological order; *eg* in the pair of equations

$$(AC) + 1 \rightarrow (AC)$$
$$If\ (AC) = 0: \ E \rightarrow (PC)$$

the quantity tested in the second equation is the word in AC after it has been incremented by one.

Boolean

SETZ	400	0 → (AC)	SETO	474	777777777777 → (AC)
SETZI	401	0 → (AC)	SETOI	475	777777777777 → (AC)
SETZM	402	0 → (E)	SETOM	476	777777777777 → (E)
SETZB	403	0 → (AC) (E)	SETOB	477	777777777777 → (AC) (E)
SETA	424	(AC) → (AC) [*no-op*]	SETCA	450	~ (AC) → (AC)
SETAI	425	(AC) → (AC) [*no-op*]	SETCAI	451	~ (AC) → (AC)
SETAM	426	(AC) → (E)	SETCAM	452	~ (AC) → (E)
SETAB	427	(AC) → (E)	SETCAB	453	~ (AC) → (AC) (E)
SETM	414	(E) → (AC)	SETCM	460	~ (E) → (AC)
SETMI	415	0,E → (AC)	SETCMI	461	~ [0,E] → (AC)
SETMM	416	(E) → (E) [*no-op*]	SETCMM	462	~ (E) → (E)
SETMB	417	(E) → (AC) (E)	SETCMB	463	~ (E) → (AC) (E)
AND	404	(AC) ∧ (E) → (AC)	ANDCA	410	~ (AC) ∧ (E) → (AC)
ANDI	405	(AC) ∧ 0,E → (AC)	ANDCAI	411	~ (AC) ∧ 0,E → (AC)
ANDM	406	(AC) ∧ (E) → (E)	ANDCAM	412	~ (AC) ∧ (E) → (E)
ANDB	407	(AC) ∧ (E) → (AC) (E)	ANDCAB	413	~ (AC) ∧ (E) → (AC) (E)
ANDCM	420	(AC) ∧ ~ (E) → (AC)	ANDCB	440	~ (AC) ∧ ~ (E) → (AC)
ANDCMI	421	(AC) ∧ ~ [0,E] → (AC)	ANDCBI	441	~ (AC) ∧ ~ [0,E] → (AC)
ANDCMM	422	(AC) ∧ ~ (E) → (E)	ANDCBM	442	~ (AC) ∧ ~ (E) → (E)
ANDCMB	423	(AC) ∧ ~ (E) → (AC) (E)	ANDCBB	443	~ (AC) ∧ ~ (E) → (AC) (E)
IOR	434	(AC) ∨ (E) → (AC)	ORCA	454	~ (AC) ∨ (E) → (AC)
IORI	435	(AC) ∨ 0,E → (AC)	ORCAI	455	~ (AC) ∨ 0,E → (AC)
IORM	436	(AC) ∨ (E) → (E)	ORCAM	456	~ (AC) ∨ (E) → (E)
IORB	437	(AC) ∨ (E) → (AC) (E)	ORCAB	457	~ (AC) ∨ (E) → (AC) (E)
ORCM	464	(AC) ∨ ~ (E) → (AC)	ORCB	470	~ (AC) ∨ ~ (E) → (AC)
ORCMI	465	(AC) ∨ ~ [0,E] → (AC)	ORCBI	471	~ (AC) ∨ ~ [0,E] → (AC)
ORCMM	466	(AC) ∨ ~ (E) → (E)	ORCBM	472	~ (AC) ∨ ~ (E) → (E)
ORCMB	467	(AC) ∨ ~ (E) → (AC) (E)	ORCBB	473	~ (AC) ∨ ~ (E) → (AC) (E)
XOR	430	(AC) ⊻ (E) → (AC)	EQV	444	~ [(AC) ⊻ (E)] → (AC)
XORI	431	(AC) ⊻ 0,E → (AC)	EQVI	445	~ [(AC) ⊻ 0,E] → (AC)
XORM	432	(AC) ⊻ (E) → (E)	EQVM	446	~ [(AC) ⊻ (E)] → (E)
XORB	433	(AC) ⊻ (E) → (AC) (E)	EQVB	447	~ [(AC) ⊻ (E)] → (AC) (E)

Byte Manipulation

IBP	133	*Operations on* (E) [*see page 2-16*] *If* P − S ⩾ 0: P − S → P *If* P − S < 0: Y + 1 → Y 36 − S → P
LDB	135	BYTE IN ((E)) → (AC) [*see page 2-16*]
DPB	137	BYTE IN (AC) → BYTE IN ((E)) [*see page 2-16*]
ILDB	134	IBP *and* LDB
IDPB	136	IBP *and* DPB

Fixed Point Arithmetic

ADD	270	(AC) + (E) → (AC)	SUB	274	(AC) − (E) → (AC)
ADDI	271	(AC) + 0,E → (AC)	SUBI	275	(AC) − 0,E → (AC)
ADDM	272	(AC) + (E) → (E)	SUBM	276	(AC) − (E) → (E)
ADDB	273	(AC) + (E) → (AC) (E)	SUBB	277	(AC) − (E) → (AC) (E)
IMUL	220	(AC) × (E) → (AC)*	MUL	224	(AC) × (E) → (AC,AC+1)
IMULI	221	(AC) × 0,E → (AC)*	MULI	225	(AC) × 0,E → (AC,AC+1)
IMULM	222	(AC) × (E) → (E)*	MULM	226	(AC) × (E) → (E)†
IMULB	223	(AC) × (E) → (AC) (E)*	MULB	227	(AC) × (E) → (AC,AC+1) (E)
IDIV	230	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	DIV	234	(AC,AC+1) ÷ (E) → (AC) REMAINDER → (AC+1)
IDIVI	231	(AC) ÷ 0,E → (AC) REMAINDER → (AC+1)	DIVI	235	(AC,AC+1) ÷ 0,E → (AC) REMAINDER → (AC+1)
IDIVM	232	(AC) ÷ (E) → (E)	DIVM	236	(AC,AC+1) ÷ (E) → (E)
IDIVB	233	(AC) ÷ (E) → (AC) (E) REMAINDER → (AC+1)	DIVB	237	(AC,AC+1) ÷ (E) → (AC) (E) REMAINDER → (AC+1)

*The high order word of the product is discarded.
†The low order word of the product is discarded.

Floating Point Arithmetic

FAD	140	(AC) + (E) → (AC)	FADR	144	(AC) + (E) → (AC)
FADL	141	(AC) + (E) → (AC,AC+1)	FADRI	145	(AC) + E,0 → (AC)
FADM	142	(AC) + (E) → (E)	FADRM	146	(AC) + (E) → (E)
FADB	143	(AC) + (E) → (AC) (E)	FADRB	147	(AC) + (E) → (AC) (E)
FSB	150	(AC) − (E) → (AC)	FSBR	154	(AC) − (E) → (AC)
FSBL	151	(AC) − (E) → (AC,AC+1)	FSBRI	155	(AC) − E,0 → (AC)
FSBM	152	(AC) − (E) → (E)	FSBRM	156	(AC) − (E) → (E)
FSBB	153	(AC) − (E) → (AC) (E)	FSBRB	157	(AC) − (E) → (AC) (E)

FMP	160	(AC) × (E) → (AC)	FMPR	164	(AC) × (E) → (AC)
FMPL	161	(AC) × (E) → (AC,AC+1)	FMPRI	165	(AC) × E,0 → (AC)
FMPM	162	(AC) × (E) → (E)	FMPRM	166	(AC) × (E) → (E)
FMPB	163	(AC) × (E) → (AC) (E)	FMPRB	167	(AC) × (E) → (AC) (E)
FDV	170	(AC) ÷ (E) → (AC)	FDVR	174	(AC) ÷ (E) → (AC)
FDVL	171	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	FDVRI	175	(AC) ÷ E,0 → (AC)
FDVM	172	(AC) ÷ (E) → (E)	FDVRM	176	(AC) ÷ (E) → (E)
FDVB	173	(AC) ÷ (E) → (AC) (E)	FDVRB	177	(AC) ÷ (E) → (AC) (E)

UFA	130	(AC) + (E) → (AC+1) *without normalization*
DFN	131	− (AC,E) → (AC,E)
FSC	132	(AC) × 2^E → (AC)
FLTR	127	(E) *floated, rounded* → (AC)

FIX	122	(E) *fixed* → (AC)	FIXR	126	(E) *fixed, rounded* → (AC)

DFAD	110	(AC,AC+1) + (E,E+1) → (AC,AC+1)
DFSB	111	(AC,AC+1) − (E,E+1) → (AC,AC+1)
DFMP	112	(AC,AC+1) × (E,E+1) → (AC,AC+1)
DFDV	113	(AC,AC+1) ÷ (E,E+1) → (AC,AC+1)

DMOVE	120	(E,E+1) → (AC,AC+1)	DMOVEM	124	(AC,AC+1) → (E,E+1)
DMOVN	121	−(E,E+1) → (AC,AC+1)	DMOVNM	125	−(AC,AC+1) → (E,E+1)

Full Word Data Transmission

EXCH	250	(AC) ↔ (E)
BLT	251	*Move* E − $(AC)_R$ + 1 *words starting with* $((AC)_L)$ → $((AC)_R)$ [*see page 2-10*]

MOVE	200	(E) → (AC)	MOVS	204	$(E)_S$ → (AC)
MOVEI	201	0,E → (AC)	MOVSI	205	E,0 → (AC)
MOVEM	202	(AC) → (E)	MOVSM	206	$(AC)_S$ → (E)
MOVES	203	*If* AC ≠ 0: (E) → (AC)	MOVSS	207	$(E)_S$ → (E) *If* AC ≠ 0: (E) → (AC)
MOVN	210	− (E) → (AC)	MOVM	214	|(E)| → (AC)
MOVNI	211	− [0,E] → (AC)	MOVMI	215	0,E → (AC)
MOVNM	212	− (AC) → (E)	MOVMM	216	|(AC)| → (E)
MOVNS	213	− (E) → (E) *If* AC ≠ 0: (E) → (AC)	MOVMS	217	|(E)| → (E) *If* AC ≠ 0: (E) → (AC)

Half Word Data Transmission

HLL	500	$(E)_L \rightarrow (AC)_L$	HLLZ	510	$(E)_L,0 \rightarrow (AC)$
HLLI	501	$0 \rightarrow (AC)_L$	HLLZI	511	$0 \rightarrow (AC)$
HLLM	502	$(AC)_L \rightarrow (E)_L$	HLLZM	512	$(AC)_L,0 \rightarrow (E)$
HLLS	503	*If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLLZS	513	$0 \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HLLO	520	$(E)_L,777777 \rightarrow (AC)$	HLLE	530	$(E)_L,[(E)_0 \times 777777] \rightarrow (AC)$
HLLOI	521	$0,777777 \rightarrow (AC)$	HLLEI	531	$0 \rightarrow (AC)$
HLLOM	522	$(AC)_L,777777 \rightarrow (E)$	HLLEM	532	$(AC)_L,[(AC)_0 \times 777777] \rightarrow (E)$
HLLOS	523	$777777 \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLLES	533	$(E)_0 \times 777777 \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HLR	544	$(E)_L \rightarrow (AC)_R$	HLRZ	554	$0,(E)_L \rightarrow (AC)$
HLRI	545	$0 \rightarrow (AC)_R$	HLRZI	555	$0 \rightarrow (AC)$
HLRM	546	$(AC)_L \rightarrow (E)_R$	HLRZM	556	$0,(AC)_L \rightarrow (E)$
HLRS	547	$(E)_L \rightarrow (E)_R$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLRZS	557	$0,(E)_L \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HLRO	564	$777777,(E)_L \rightarrow (AC)$	HLRE	574	$[(E)_0 \times 777777],(E)_L \rightarrow (AC)$
HLROI	565	$777777,0 \rightarrow (AC)$	HLREI	575	$0 \rightarrow (AC)$
HLROM	566	$777777,(AC)_L \rightarrow (E)$	HLREM	576	$[(AC)_0 \times 777777],(AC)_L \rightarrow (E)$
HLROS	567	$777777,(E)_L \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HLRES	577	$[(E)_0 \times 777777],(E)_L \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HRR	540	$(E)_R \rightarrow (AC)_R$	HRRZ	550	$0,(E)_R \rightarrow (AC)$
HRRI	541	$E \rightarrow (AC)_R$	HRRZI	551	$0,E \rightarrow (AC)$
HRRM	542	$(AC)_R \rightarrow (E)_R$	HRRZM	552	$0,(AC)_R \rightarrow (E)$
HRRS	543	*If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRRZS	553	$0 \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HRRO	560	$777777,(E)_R \rightarrow (AC)$	HRRE	570	$[(E)_{18} \times 777777],(E)_R \rightarrow (AC)$
HRROI	561	$777777,E \rightarrow (AC)$	HRREI	571	$[E_{18} \times 777777],E \rightarrow (AC)$
HRROM	562	$777777,(AC)_R \rightarrow (E)$	HRREM	572	$[(AC)_{18} \times 777777],(AC)_R \rightarrow (E)$
HRROS	563	$777777 \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRRES	573	$(E)_{18} \times 777777 \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
HRL	504	$(E)_R \rightarrow (AC)_L$	HRLZ	514	$(E)_R,0 \rightarrow (AC)$
HRLI	505	$E \rightarrow (AC)_L$	HRLZI	515	$E,0 \rightarrow (AC)$
HRLM	506	$(AC)_R \rightarrow (E)_L$	HRLZM	516	$(AC)_R,0 \rightarrow (E)$
HRLS	507	$(E)_R \rightarrow (E)_L$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRLZS	517	$(E)_R,0 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$

HRLO	524	$(E)_R,777777 \rightarrow (AC)$	HRLE	534	$(E)_R,[(E)_{18} \times 777777] \rightarrow (AC)$
HRLOI	525	$E,777777 \rightarrow (AC)$	HRLEI	535	$E,[E_{18} \times 777777] \rightarrow (AC)$
HRLOM	526	$(AC)_R,777777 \rightarrow (E)$	HRLEM	536	$(AC)_R,[(AC)_{18} \times 777777] \rightarrow (E)$
HRLOS	527	$(E)_R,777777 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$	HRLES	537	$(E)_R,[(E)_{18} \times 777777] \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$

In-out

CONO	70020	$E \rightarrow$ COMMAND	CONSZ	70030	*If* $\text{STATUS}_R \wedge E = 0$: *skip*
CONI	70024	STATUS $\rightarrow (E)$	CONSO	70034	*If* $\text{STATUS}_R \wedge E \neq 0$: *skip*
DATAO	70014	$(E) \rightarrow$ DATA	DATAI	70004	DATA $\rightarrow (E)$

BLKO	70010	$(E) + 1000001 \rightarrow (E)$*	$((E)_R) \rightarrow$ DATA [*see page 2-77*]
BLKI	70000	$(E) + 1000001 \rightarrow (E)$*	DATA $\rightarrow ((E)_R)$ [*see page 2-77*]

Program Control

JSR	264	FLAGS,$(PC) \rightarrow (E)$	$E + 1 \rightarrow (PC)$	
JSP	265	FLAGS,$(PC) \rightarrow (AC)$	$E \rightarrow (PC)$	
JRST	254	$E \rightarrow (PC)$	[*If* $AC \neq 0$, *see page 2-63*]	
JSA	266	$(AC) \rightarrow (E)$	$E,(PC) \rightarrow (AC)$	$E + 1 \rightarrow (PC)$
JRA	267	$E \rightarrow (PC)$	$((AC)_L) \rightarrow (AC)$	
JFCL	255	*If* AC $\wedge$ FLAGS $\neq 0$:	$E \rightarrow (PC)$	$\sim$ AC $\wedge$ FLAGS $\rightarrow$ FLAGS
XCT	256	*Execute* (E)		
JFFO	243	*If* $(AC) = 0$: $0 \rightarrow (AC + 1)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$ [*see page 2-61*]		
MAP	257	PHYSICAL MAP DATA $\rightarrow (AC)$		

Pushdown List

PUSH	261	$(AC) + 1000001 \rightarrow (AC)$*	$(E) \rightarrow ((AC)_R)$	
POP	262	$((AC)_R) \rightarrow (E)$	$(AC) - 1000001 \rightarrow (AC)$*	
PUSHJ	260	$(AC) + 1000001 \rightarrow (AC)$*	FLAGS,$(PC) \rightarrow ((AC)_R)$	$E \rightarrow (PC)$
POPJ	263	$((AC)_R)_R \rightarrow (PC)$	$(AC) - 1000001 \rightarrow (AC)$*	

Shift and Rotate

ASH	240	$(AC) \times 2^E \rightarrow (AC)$	ASHC	245	$(AC,AC+1) \times 2^E \rightarrow (AC,AC+1)$
ROT	241	*Rotate* (AC) E *places*	ROTC	246	*Rotate* (AC,AC+1) E *places*
LSH	242	*Shift* (AC) E *places*	LSHC	247	*Shift* (AC,AC+1) E *places*

*In the KI10, 1 is added to or subtracted from each half separately.

Arithmetic Testing

AOBJP	252	(AC) + 1000001 → (AC)*	*If* (AC) ≥ 0: E → (PC)
AOBJN	253	(AC) + 1000001 → (AC)*	*If* (AC) < 0: E → (PC)

CAI	300	*No-op*	CAM	310	*No-op*
CAIL	301	*If* (AC) < E: *skip*	CAML	311	*If* (AC) < (E): *skip*
CAIE	302	*If* (AC) = E: *skip*	CAME	312	*If* (AC) = (E): *skip*
CAILE	303	*If* (AC) ≤ E: *skip*	CAMLE	313	*If* (AC) ≤ (E): *skip*
CAIA	304	*Skip*	CAMA	314	*Skip*
CAIGE	305	*If* (AC) ≥ E: *skip*	CAMGE	315	*If* (AC) ≥ (E): *skip*
CAIN	306	*If* (AC) ≠ E: *skip*	CAMN	316	*If* (AC) ≠ (E): *skip*
CAIG	307	*If* (AC) > E: *skip*	CAMG	317	*If* (AC) > (E): *skip*
JUMP	320	*No-op*	SKIP	330	*If* AC ≠ 0: (E) → (AC)
JUMPL	321	*If* (AC) < 0: E → (PC)	SKIPL	331	*If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*
JUMPE	322	*If* (AC) = 0: E → (PC)	SKIPE	332	*If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*
JUMPLE	323	*If* (AC) ≤ 0: E → (PC)	SKIPLE	333	*If* AC ≠ 0: (E) → (AC) *If* (E) ≤ 0: *skip*
JUMPA	324	E → (PC)	SKIPA	334	*If* AC ≠ 0: (E) → (AC) *Skip*
JUMPGE	325	*If* (AC) ≥ 0: E → (PC)	SKIPGE	335	*If* AC ≠ 0: (E) → (AC) *If* (E) ≥ 0: *skip*
JUMPN	326	*If* (AC) ≠ 0: E → (PC)	SKIPN	336	*If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
JUMPG	327	*If* (AC) > 0: E → (PC)	SKIPG	337	*If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*
AOJ	340	(AC) + 1 → (AC)	SOJ	360	(AC) − 1 → (AC)
AOJL	341	(AC) + 1 → (AC) *If* (AC) < 0: E → (PC)	SOJL	361	(AC) − 1 → (AC) *If* (AC) < 0: E → (PC)
AOJE	342	(AC) + 1 → (AC) *If* (AC) = 0: E → (PC)	SOJE	362	(AC) − 1 → (AC) *If* (AC) = 0: E → (PC)
AOJLE	343	(AC) + 1 → (AC) *If* (AC) ≤ 0: E → (PC)	SOJLE	363	(AC) − 1 → (AC) *If* (AC) ≤ 0: E → (PC)
AOJA	344	(AC) + 1 → (AC) E → (PC)	SOJA	364	(AC) − 1 → (AC) E → (PC)
AOJGE	345	(AC) + 1 → (AC) *If* (AC) ≥ 0: E → (PC)	SOJGE	365	(AC) − 1 → (AC) *If* (AC) ≥ 0: E → (PC)

*In the KI10, 1 is added to or subtracted from each half separately.

AOJN	346	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$	SOJN	366	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) \neq 0$: $E \rightarrow (PC)$
AOJG	347	$(AC) + 1 \rightarrow (AC)$ *If* $(AC) > 0$: $E \rightarrow (PC)$	SOJG	367	$(AC) - 1 \rightarrow (AC)$ *If* $(AC) > 0$: $E \rightarrow (PC)$
AOS	350	$(E) + 1 \rightarrow (E)$ *If* $(AC) \neq 0$: $(E) \rightarrow (AC)$	SOS	370	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$
AOSL	351	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) < 0$: *skip*	SOSL	371	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) < 0$: *skip*
AOSE	352	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) = 0$: *skip*	SOSE	372	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) = 0$: *skip*
AOSLE	353	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \leqslant 0$: *skip*	SOSLE	373	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \leqslant 0$: *skip*
AOSA	354	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *Skip*	SOSA	374	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *Skip*
AOSGE	355	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \geqslant 0$: *skip*	SOSGE	375	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \geqslant 0$: *skip*
AOSN	356	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \neq 0$: *skip*	SOSN	376	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) \neq 0$: *skip*
AOSG	357	$(E) + 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) > 0$: *skip*	SOSG	377	$(E) - 1 \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ *If* $(E) > 0$: *skip*

Logical Testing and Modification

TLN	601	*No-op*	TRN	600	*No-op*
TLNE	603	*If* $(AC)_L \wedge E = 0$: *skip*	TRNE	602	*If* $(AC)_R \wedge E = 0$: *skip*
TLNA	605	*Skip*	TRNA	604	*Skip*
TLNN	607	*If* $(AC)_L \wedge E \neq 0$: *skip*	TRNN	606	*If* $(AC)_R \wedge E \neq 0$: *skip*
TLZ	621	$(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZ	620	$(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZE	623	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZE	622	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZA	625	$(AC)_L \wedge \sim E \rightarrow (AC)_L$ *skip*	TRZA	624	$(AC)_R \wedge \sim E \rightarrow (AC)_R$ *skip*
TLZN	627	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZN	626	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$

A20 MNEMONICS

TLC	641	$(AC)_L \forall E \rightarrow (AC)_L$	TRC	640	$(AC)_R \forall E \rightarrow (AC)_R$
TLCE	643	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \forall E \rightarrow (AC)_L$	TRCE	642	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \forall E \rightarrow (AC)_R$
TLCA	645	$(AC)_L \forall E \rightarrow (AC)_L$ *skip*	TRCA	644	$(AC)_R \forall E \rightarrow (AC)_R$ *skip*
TLCN	647	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \forall E \rightarrow (AC)_L$	TRCN	646	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \forall E \rightarrow (AC)_R$
TLO	661	$(AC)_L \vee E \rightarrow (AC)_L$	TRO	660	$(AC)_R \vee E \rightarrow (AC)_R$
TLOE	663	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TROE	662	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TLOA	665	$(AC)_L \vee E \rightarrow (AC)_L$ *skip*	TROA	664	$(AC)_R \vee E \rightarrow (AC)_R$ *skip*
TLON	667	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TRON	666	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TDN	610	*No-op*	TSN	611	*No-op*
TDNE	612	*If* $(AC) \wedge (E) = 0$: *skip*	TSNE	613	*If* $(AC) \wedge (E)_S = 0$: *skip*
TDNA	614	*Skip*	TSNA	615	*Skip*
TDNN	616	*If* $(AC) \wedge (E) \neq 0$: *skip*	TSNN	617	*If* $(AC) \wedge (E)_S \neq 0$: *skip*
TDZ	630	$(AC) \wedge \sim (E) \rightarrow (AC)$	TSZ	631	$(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZE	632	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZE	633	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZA	634	$(AC) \wedge \sim (E) \rightarrow (AC)$ *skip*	TSZA	635	$(AC) \wedge \sim (E)_S \rightarrow (AC)$ *skip*
TDZN	636	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZN	637	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDC	650	$(AC) \forall (E) \rightarrow (AC)$	TSC	651	$(AC) \forall (E)_S \rightarrow (AC)$
TDCE	652	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \forall (E) \rightarrow (AC)$	TSCE	653	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \forall (E)_S \rightarrow (AC)$
TDCA	654	$(AC) \forall (E) \rightarrow (AC)$ *skip*	TSCA	655	$(AC) \forall (E)_S \rightarrow (AC)$ *skip*
TDCN	656	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \forall (E) \rightarrow (AC)$	TSCN	657	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \forall (E)_S \rightarrow (AC)$
TDO	670	$(AC) \vee (E) \rightarrow (AC)$	TSO	671	$(AC) \vee (E)_S \rightarrow (AC)$
TDOE	672	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSOE	673	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$
TDOA	674	$(AC) \vee (E) \rightarrow (AC)$ *skip*	TSOA	675	$(AC) \vee (E)_S \rightarrow (AC)$ *skip*
TDON	676	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSON	677	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$

APPENDIX B

INPUT-OUTPUT CODES

The table beginning on the next page lists the complete teletype code. The lower case character set (codes 140–176) is not available on the Model 35, but giving one of these codes causes the teletype to print the corresponding upper case character. Other differences between the 35 and 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the console teletype, and the definitions bear no necessary relation to the use of the codes in conjunction with the DECsystem–10 software.

The line printer has the same codes and characters as the teletype. The 64-character printer has the figure and upper case sets, codes 040–137 (again, giving a lower case code prints the upper case character). The "96"-character printer has these plus the lower case set, codes 040–176. The latter printer actually has only ninety-five characters unless a special character is "hidden" under the delete code, 177. A hidden character is printed by sending its code prefixed by the delete code. Hence a character hidden under DEL is printed by sending the printer two 177s in a row.

Besides printing characters, the line printer responds to ten control characters, HT, CR, LF, VT, FF, DLE and DC1–4. The 128-character printer uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control characters that affect the printer and also under null and delete. In all cases, prefixing DEL causes the hidden character to be printed. The extra thirty-three characters that complete the set are ordered special for each installation.

The first page of the table of card codes [*pages B6–8*] lists the column punch required to represent any character in the two DEC codes. The octal codes listed are those used by the DECsystem–10 software. In other words, when reading cards, the Monitor translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. The remaining pages of the table show the relationship between the DEC card codes and several IBM card punches. Each of the column punches is produced by a single key on any punch for which a character is listed, the character being that which is printed at the top of the card.

TELETYPE CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 35.
1	001	SOH	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also EOA, end of address. Control B.
0	003	ETX	End of text; also EOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is . . . ") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you . . . ?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 35.
0	011	HT	Horizontal tab. Control I on Model 35.
0	012	LF	Line feed or line space (NEW LINE); advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 35.
1	013	VT	Vertical tab (VTAB). Control K on Model 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DC0).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (~~TAPE~~, AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is generated by control shift K on Model 35, but the Monitor translates it to 175.
1	034	FS	File separator. Control shift L on Model 35.
0	035	GS	Group separator. Control shift M on Model 35.

TELETYPE CODE B3

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	036	RS	Record separator. Control shift N on Model 35.
1	037	US	Unit separator. Control shift O on Model 35.
1	040	SP	Space.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052	*	Repeats on Model 37.
0	053	+	
1	054	,	
0	055	-	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	Ø	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	

B4 INPUT-OUTPUT CODES

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[	Shift K on Model 35.
0	134	\	Shift L on Model 35.
1	135	]	Shift M on Model 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	

TELETYPE CODE B5

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.
1	171	y	
1	172	z	
0	173	{	
1	174			
0	175	}	This code generated by ALT MODE on Model 35.
0	176	~	This code generated by ESC key (if present) on Model 35, but the Monitor translates it to 175.
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 35 local line feed.
LOC CR	Model 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 21-character message.

CARD CODES

Character	PDP-10 ASCII	DEC 029	DEC 026	Character	PDP-10 ASCII	DEC 029	DEC 026
Space	040	*None*	*None*	@	100	8 4	8 4
!	041	11 8 2	12 8 7	A	101	12 1	12 1
"	042	8 7	0 8 5	B	102	12 2	12 2
#	043	8 3	0 8 6	C	103	12 3	12 3
$	044	11 8 3	11 8 3	D	104	12 4	12 4
%	045	0 8 4	0 8 7	E	105	12 5	12 5
&	046	12	11 8 7	F	106	12 6	12 6
'	047	8 5	8 6	G	107	12 7	12 7
(	050	12 8 5	0 8 4	H	110	12 8	12 8
)	051	11 8 5	12 8 4	I	111	12 9	12 9
*	052	11 8 4	11 8 4	J	112	11 1	11 1
+	053	12 8 6	12	K	113	11 2	11 2
,	054	0 8 3	0 8 3	L	114	11 3	11 3
−	055	11	11	M	115	11 4	11 4
.	056	12 8 3	12 8 3	N	116	11 5	11 5
/	057	0 1	0 1	O	117	11 6	11 6
0	060	0	0	P	120	11 7	11 7
1	061	1	1	Q	121	11 8	11 8
2	062	2	2	R	122	11 9	11 9
3	063	3	3	S	123	0 2	0 2
4	064	4	4	T	124	0 3	0 3
5	065	5	5	U	125	0 4	0 4
6	066	6	6	V	126	0 5	0 5
7	067	7	7	W	127	0 6	0 6
8	070	8	8	X	130	0 7	0 7
9	071	9	9	Y	131	0 8	0 8
:	072	8 2	11 8 2 *or* 11 0	Z	132	0 9	0 9
;	073	11 8 6	0 8 2	[	133	12 8 2	11 8 5
<	074	12 8 4	12 8 6	\	134	11 8 7	8 7
=	075	8 6	8 3	]	135	0 8 2	12 8 5
>	076	0 8 6	11 8 6	↑	136	12 8 7	8 5
?	077	0 8 7	12 8 2 *or* 12 0	←	137	0 8 5	8 2

Binary	7 9
Mode Switch	12 0 2 4 6 8
End of File	12 11 0 1, 6 7 8 9, 12 11 0 1 6 7 8 9

The octal codes given above are those generated by the Monitor from the column punches. The card reader interface actually supplies a direct binary equivalent of the column punch, as listed in the following two pages.

The first end-of-file punch is not recognized by Card Reader Stacker (CDRSTK); the other two are recognized only by Card Reader Stacker.

CARD CODES B7

Column Punch	Character	Octal
None	*Space*	0000
0	0	1000
1	1	0400
2	2	0200
3	3	0100
4	4	0040
5	5	0020
6	6	0010
7	7	0004
8	8	0002
9	9	0001
12 1	A	4400
12 2	B	4200
12 3	C	4100
12 4	D	4040
12 5	E	4020
12 6	F	4010
12 7	G	4004
12 8	H	4002
12 9	I	4001
11 1	J	2400
11 2	K	2200
11 3	L	2100
11 4	M	2040
11 5	N	2020
11 6	O	2010
11 7	P	2004
11 8	Q	2002
11 9	R	2001
0 1	/	1400
0 2	S	1200
0 3	T	1100
0 4	U	1040
0 5	V	1020
0 6	W	1010
0 7	X	1004
0 8	Y	1002
0 9	Z	1001

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12	&	+	&	+	&	4000
11	–	–	–	–	–	2000
12 0				?		5000
11 0				:		3000
8 2			:	←	:	0202
8 3	#	=	#	=	#	0102
8 4	@	-	@	@	@	0042
8 5			'	↑	'	0022
8 6			=	'	=	0012
8 7			"	\	"	0006
12 8 2			¢	?	[	4202
12 8 3	.	.	.	.	.	4102
12 8 4	¤	)	<	)	<	4042
12 8 5			(	]	(	4022
12 8 6			+	<	+	4012

B8 INPUT-OUTPUT CODES

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12 8 7			|	!	↑	4006
11 8 2			!	:	!	2202
11 8 3	$	$	$	$	$	2102
11 8 4	*	*	*	*	*	2042
11 8 5			)	[	)	2022
11 8 6			;	>	;	2012
11 8 7			¬	&	\	2006
0 8 2			*See note*	;	]	1202
0 8 3	,	,	,	,	,	1102
0 8 4	%	(	%	(	%	1042
0 8 5			←	"	←	1022
0 8 6			>	#	>	1012
0 8 7			?	%	?	1006
12 11 0 1				*End of File**	*End of File**	7400
12 0 2 4 6 8				*Mode Switch*	*Mode Switch*	5252
7 9				*Binary*	*Binary*	*xx*05
6 7 8 9				*End of File*†	*End of File*†	
12 11 0 1 6 7 8 9				*End of File*†	*End of File*†	

NOTE: There is a single key for the 0 8 2 punch on the 029 but printing is suppressed.

The Monitor translates the octal code for the 12 0 punch in DEC 026 to 4202 (which corresponds to a 12 8 2 punch), and the code for 11 0 to 2202 (11 8 2).

*Not recognized as end of file by Card Reader Stacker (CDRSTK).

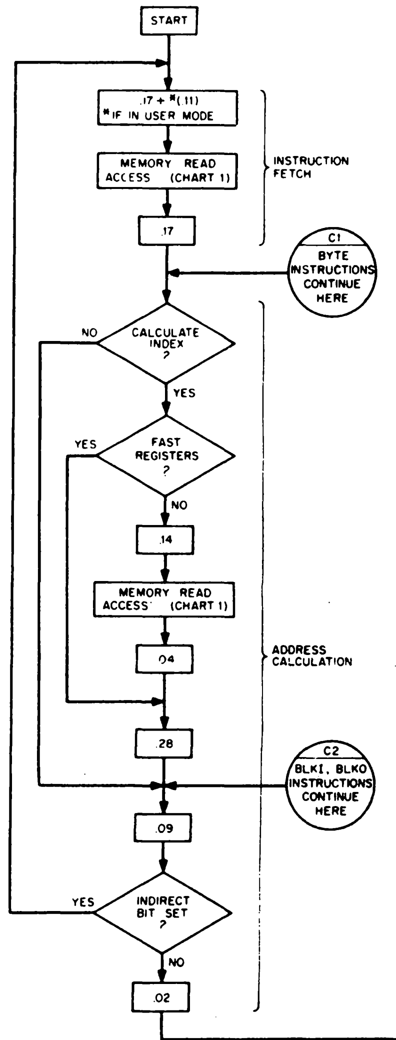
†Recognized only by Card Reader Stacker (CDRSTK).

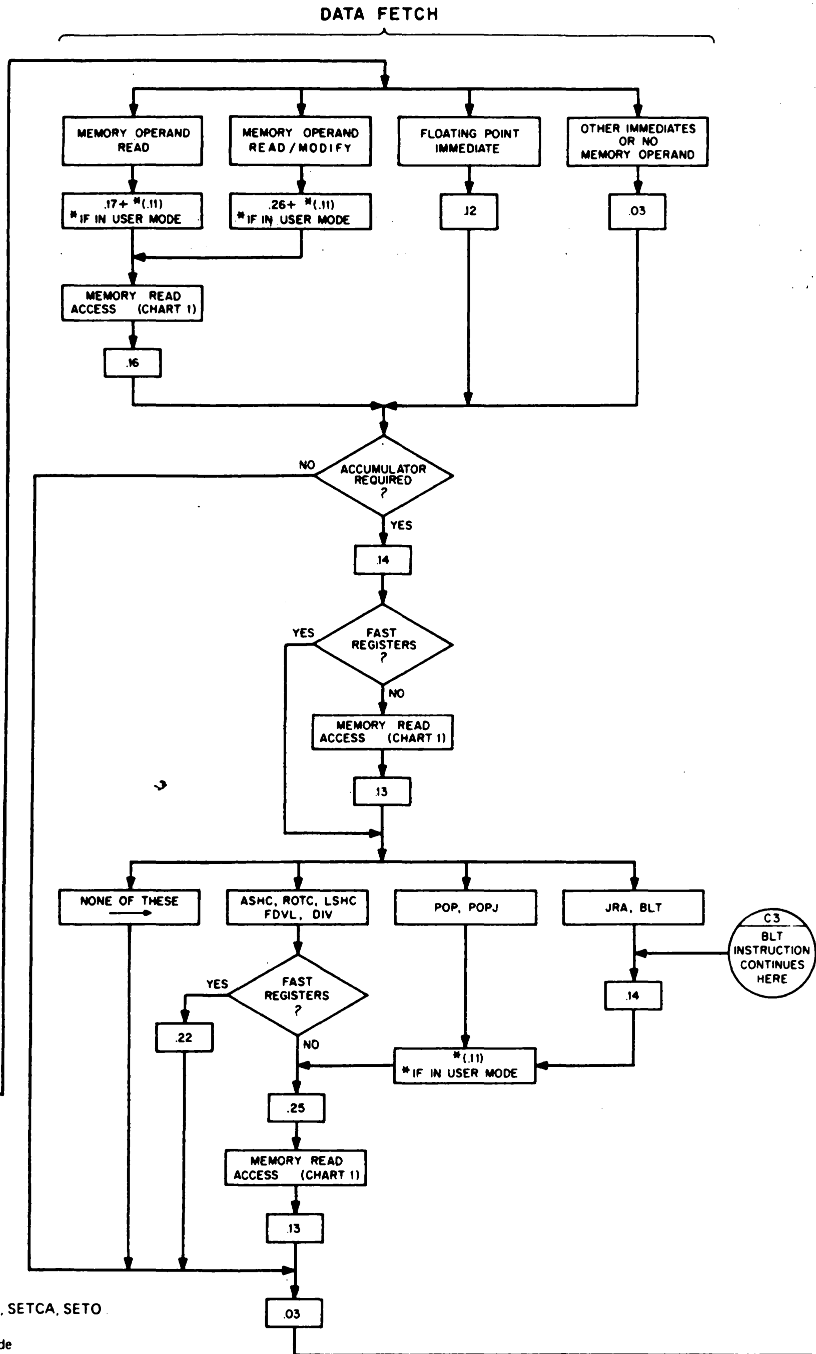
APPENDIX C

TIMING

The chart on the next two pages shows the detailed timing for the KA10. A similar chart for the KI10 and timing tables for both processors will be added later.

C2 TIMING





KA10 INSTRUCTION TIMING FLOW CHART

INSTRUCTIONS THAT USE READ/MODIFY

All Boolean in Memory and Both modes except SETZ, SETA, SETCA, SETO
ADDM, ADDB, SUBM, SUBB
HRRM, HRLM, HLRM, HLLM and all half words in Self mode
MOVES, MOVNS, MOVMS, MOVSS
ILDB, IDPB (first time only)
IBP, BLKI, BLKO, DFN, EXCH
AOS, SOS in all modes

KA10 TIMING

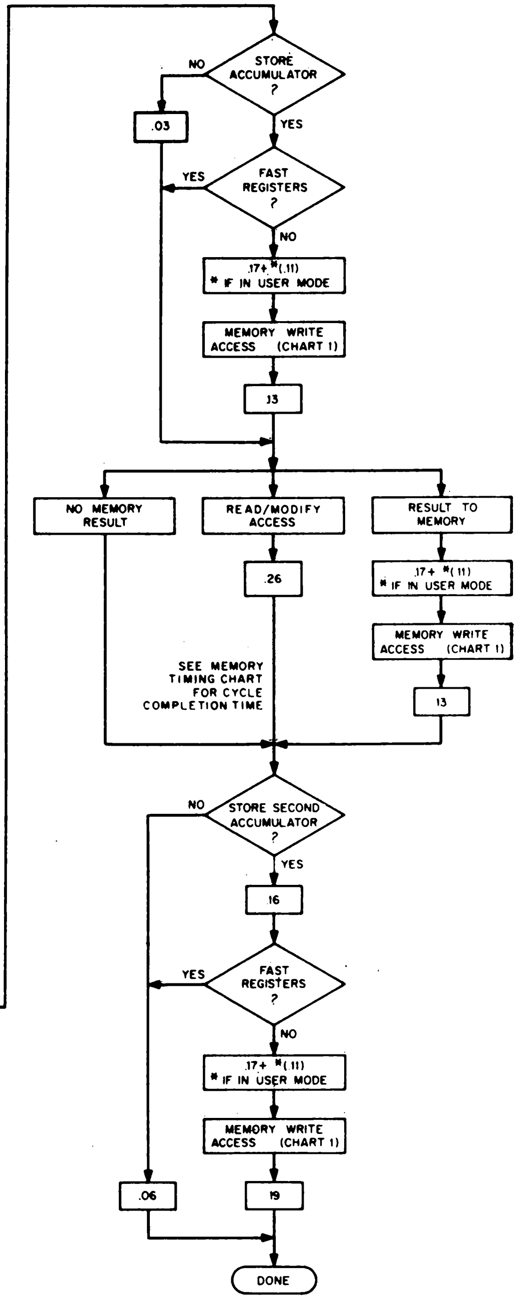
C3

INSTRUCTION EXECUTION

Instruction		Time	Additional
Boolean (except ANDCA, ANDCB, ORCA, ORCB), Half Words (except HLR, HLRI, HRL, HRLI), MOVE, MOVS, EXCH, JFCL, JRST, JSP, XCT, UUO		27	
ANDCA, ANDCB, ORCA, ORCB, HLR, HLRI, HRL, HRLI, JSR, JSA, JRA, Test class		62	
MOVN, MOVM, ADD, SUB, AOBJP, AOBJN, CAM, CAI, SKIP, JUMP, AOJ, AOS, SOJ, SOS		.45	
PUSH, PUSHJ, POP, POPJ, DFN		.80	
JFFO		80	+ .19 times number of leading 0s mod 18
BLT		69	(+ .11 if User) + memory write access + .52 If not done + 09 and go to C3
IBP		38	+ 26 if overflow word boundary
LDB, DPB	First time	.61	+ 15 per size count; Go to C1
ILDB, IDPB	First time	.74	{ + .15 per size count; + .26 if overflow } Go to C1
ILDB, LDB	Second time	.45	+ .15 per position count
IDPB, DPB	Second time	95	+ 15 per position count
Shift group		{ .39 Left; 23 Right }	+ .15 per shift
MUL		6 02	+ .13 per transition
Average except MULI		8.36	(18 transitions for 2.34)
IMUL		6 34	+ .13 per transition
Average except IMULI		7.51	(9 transitions for 1 17)
FMP		6.39	+ 13 per transition
Average except FMPRI		8.21	(14 transitions for 1 82)
Note Immediate mode multiplication has only half the average number of transitions			
DIV, IDIV		13.78	
FSC		1.52	+ 25 per shift to normalize
FAD, UFA		2 38	{ + 15 per shift to unnormalize; + .25 per shift to normalize }
Average		4 33	
FSB		Same as FAD + .18	
Rounding (except divide) only when actually done		+.96	
Long mode (except divide)		+.69	
FDVR, FDV (except FDVL)		12.00	
FDVL with fast ACs		13.28	
FDVL without fast ACs		12.32	(+ .11 if User) + memory read access + .89
CONO, CONI, CONSO, CONSZ, DATAO, DATAI		.12	Then wait until 4.50 has passed since last here
CONO, CONI, DATAO, DATAI		+2.69	
CONSO, CONSZ		+2.90	
BLKO, BLKI		.60	Then turn into DATAO, DATAI and go to C2

.03

DATA STORE



MEMORY TIMING

MEMORY	MA10	MB10	MB10	FAST	MD10	ME10
PROCESSORS	SINGLE OR MULTI	SINGLE	MULTI	SINGLE (BUILT IN)	SINGLE OR MULTI	SINGLE OR MULTI
CYCLE	1.00	1.65	1.75	—	1.8	1.00
READ ACCESS	.61	.60	.70	.21	.83	.61
WRITE ACCESS	.20	.20	.30	.21	.33	.20
MODIFY COMPLETION	.57	.97	.97	—	1.23	.65

NOTES:

MEMORY ACCESS TIMES INCLUDE DELAY INTRODUCED BY 10 FEET OF CABLE

ALL TIMES ARE NOMINAL MAXIMUMS

APPENDIX D

KA10 ALGORITHMS

All arithmetic operations on full and half words are performed in the 36-bit parallel adder. There are two sets of summand inputs to the adder, each set of 36 supplying one input to each adder stage. One set supplies the contents of AR, its complement, or zero; the other set supplies the contents of BR, its complement, or zero. Each stage also has a carry input, which is generated by the next less significant stage. Every stage has two outputs; the carry already mentioned, and a sum. The 36 sum outputs together form the sum of the two input words. The least significant stage has a carry input from the logic for performing twos complement arithmetic and incrementing by one. The negative of a number is formed at the sum outputs simply by supplying the complement of the number at one set of inputs and asserting the carry into stage 35. Adder stage 17 has extra input gating so that 1 can be added to or subtracted from both halves of AR simultaneously.

This appendix treats only the algorithms used in the KA10; for information on the KI10 algorithms refer to the maintenance manual.

The adder produces a sum in the same way that one adds binary numbers using pencil and paper. Each adder stage has three inputs, two summand bits and a carry, and two outputs, sum and carry. The sum output of a given stage is 1 if any one or all three of the inputs are 1. The carry out is 1 if two or three of the inputs are 1. Calculations are performed as though the words represented 36-bit unsigned numbers, *ie* the signs are treated just like 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 summands have different signs. The result has a minus sign when there is a carry into the sign bit and the summands have the same sign, or the summands have different signs and there is no carry.

Thus the program can interpret the numbers processed in fixed point arithmetic as signed numbers with 35 magnitude bits or as unsigned 36-bit numbers. A computation on signed numbers produces a result which is correct as an unsigned 36-bit number even if overflow occurs, but the hardware interprets the result as a signed number to detect 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.

All operations discussed below have two operands, one of which is supplied to the adder from BR, which acts simply as a buffer and has no special input gating. MQ has shift gating so it can function as a low order extension of AR for handling double length operands. All actual computations take place in the single 36-bit adder, but the sum output can be placed in either AR or MQ, and all transfers to MQ from AR or BR are made through the adder. In multiplication MQ holds the multiplier and thus

controls the summation of partial products; as the multiplier is shifted out, the low order word of the product is shifted in. In division MQ supplies the low order part of the dividend to AR as the quotient is being constructed in MQ.

In any extended arithmetic operation, the requisite number of steps is counted in the 9-bit shift counter SC, which has a carry network for this purpose. SC also has a 9-bit adder for use in computations on floating point exponents and size and position calculations in byte manipulation.

FIXED POINT ALGORITHMS

Fixed point numbers are explained in detail in § 1.1. For convenience let us take the computer representation of the positive number x as $+[x]$ where the brackets enclose the number in bits 1–35. Similarly the representation of $-x$ is $-[2^{35}-x]$ or $-[1-x]$ depending on whether we are regarding numbers as integers or as proper fractions. The most negative number, -2^{35}, has the form $-[0]$, which is equivalent to the unsigned integer 2^{35}.

Addition. There are four cases of addition of two positive 35-bit numbers x and y.

I.	$x+y$	
II.	$(-x)+(-y)$	
III.	$x+(-y)$,	$x \geqslant y$
IV.	$x+(-y)$,	$x < y$

The operands are held in AR and BR, but it makes no difference which one is in which register. The result appears in AR. For convenience in the exposition we shall regard the numbers as proper fractions; to view them as integers, simply substitute "2^{35}" for each occurrence of "1". Since the twos complement format allows a representation for -1, either x or y may be 1 in II, and y may be 1 in IV.

I. If $x+y<1$ the adder output placed in AR is $+[x+y]$. If $x+y \geqslant 1$ the carry out of stage 1 changes the sign. Consequently if the addition of two positive numbers gives a negative result, it is apparent that the sum exceeds the capacity of the register. The processor detects the overflow by checking the sign carries: there is a carry into the sign stage but none out of it. AR then contains

$$-[x+y-1]$$

II. Ignoring the carry into the sign bit in the addition of two negatives would give

$$\begin{array}{l} -[1-x] \\ \underline{-[1-y]} \\ +[1+1-x-y] \end{array}$$

If $x+y \leqslant 1$ the carry changes the sign and the result is

$$-[1-x-y]$$

which is the representation of $-(x+y)$. If $x+y>1$ there is no carry into the sign, and its absence in the presence of a carry out indicates overflow. AR contains

$$+[1-(x+y-1)]$$

III. Ignoring the carry into the sign in an addition where the signs are different would give

$$\begin{array}{l} +[x] \\ -[1-y] \\ \hline -[1+x-y] \end{array}$$

Since $x \geqslant y$, it follows that $1+x-y \geqslant 1$. Hence the carry changes the sign and the result is

$$+[x-y]$$

When the operand signs are different, the magnitude of the result cannot exceed the larger operand magnitude and there can be no overflow. Since in this case the positive number is at least as large in magnitude as the negative, there is always a carry into the sign, and this added to the operand minus sign produces a carry out.

IV. The addition of numbers of differing signs where the negative has the larger magnitude gives

$$\begin{array}{l} +[x] \\ -[1-y] \\ \hline -[1+x-y] \end{array}$$

Since $x<y$, then $1+x-y<1$. Hence there are no carries associated with the sign and no overflow. The above result is the twos complement representation of $x-y$, *ie* $-(y-x)$.

Subtraction. The minuend from AC is in AR, and the subtrahend, which is either $0, E$ or the word from location E, is in BR. Subtraction is done directly by adding the twos complement of BR to AR. The logic supplies the complement of BR to the adder and a carry into the adder LSB.

Let x be the absolute value of the number in AR, and y the absolute value of the number in BR. There are four cases.

I.	$x-(-y)$			
II.	$(-x)-y$			
III.	$x-y,$	$x \geqslant y;$	$(-x)-(-y),$	$x \leqslant y$
IV.	$x-y,$	$x<y;$	$(-x)-(-y),$	$x>y$

These correspond respectively to the four cases of addition discussed previously.

Multiplication. The multiplier, $0, E$ or the contents of location E, is in MQ, and the multiplicand from AC is in BR. AR is clear. The 36-step procedure is as follows.

If MQ35 (the multiplier LSB) is 1, subtract BR algebraically from AR, but put the result in AR shifted one place to the right, with the LSB of the result going into MQ0, and shift MQ right so a bit of the multiplier is dropped from MQ35. Put the sign of the result in AR0 and AR1 (as though the shift followed the subtraction and did not affect the sign but did move it to AR1). If MQ35 is 0, simply shift AR and MQ right one, with AR35 going into MQ0.

In each subsequent step perform only the shift if the bits moved in and out of MQ35 on the previous step were the same. If they were different, add or subtract along with the shift: if the shift moved a 0 in and a 1 out, add BR to AR; if a 1 in and a 0 out, subtract BR from AR.

Thus the low order bits of the running sum of partial products are shifted into MQ as the multiplier is shifted out. At each step the effect of the multiplicand in BR on the partial sum in AR is one binary order of magnitude greater than in the preceding step because the partial sum was shifted right. Therefore BR can be combined directly with AR. If MQ35 is initially 0, there is no subtraction until a 1 is shifted into it. Simple shifting then continues until the next transition (from 1 to 0), following which BR is added.

The process continues in this way, subtracting at every 0-1 transition, adding at every 1-0 transition. After 35 steps MQ0–34 contains the low half of the product magnitude, and MQ35 contains the sign of the multiplier. At the final step, add or subtract as required but put the result directly into AR; shift only MQ to move the low magnitude into the correct position and make MQ0 equal to the sign of the whole product.

If the original operands were both negative and the result is also negative, set Overflow; this can occur only when -2^{35} is squared. In IMUL, if the high word is not null (*ie* if AR is neither clear nor all 1s), set Overflow; move MQ to AR for storage of the low word.

To see that this procedure results in a correct product, consider the positive binary integer

$$\underset{8\,7\,6\,5\,4\,3\,2\,1\,0}{1\,0\,0\,1\,1\,1\,0\,1\,1}$$

where the decimal digits below the binary digits are the powers of 2 corresponding to the bit positions. This number is obviously equal to

$$\begin{array}{r} 100000000 \\ +\quad 111000 \\ +\quad 11 \end{array}$$

Now an n-bit string of 1s whose rightmost bit corresponds to 2^k is equal to $2^{k+n}-2^k$, or equivalently $2^k(2^n-2^0)$, *ie* 2^n-2^0 is a string of n 1s and the 2^k shifts the string left k places. Hence

$$\begin{array}{rcccl} 100000000 & = & 2^{8+1}-2^8 & = & 2^9-2^8 \\ 111000 & = & 2^{3+3}-2^3 & = & 2^6-2^3 \\ 11 & = & 2^{2+0}-2^0 & = & 2^2-2^0 \\ \hline 100111011 & & = & & 2^9-2^8+2^6-2^3+2^2-2^0 \end{array}$$

In this last representation, each power of 2 that is subtracted corresponds to

a transition from 0 to 1 (scanning right to left), whereas each that is added corresponds to a 1-0 transition. The largest term corresponds to the transition to the sign bit, which is 0 for a positive number. The multiplication algorithm interprets the multiplier in this manner, alternately subtracting and adding the multiplicand to the partial sum in the order-of-magnitude positions corresponding to the transitions. If a multiplier of the same magnitude were negative, it would have the form

$$\begin{array}{cccccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ - & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{array}$$

in which the extra bit at the left represents the sign. The number is now equivalent to

$$-2^9 + 2^8 - 2^6 + 2^3 - 2^2 + 2^1 - 2^0$$

wherein opposite signs correspond to opposite transitions. The algorithm may thus use exactly the same sequence for a negative multiplier: this time the subtraction of greatest magnitude is detected by the transition to the sign bit, which is now 1.

Division. The divisor, $0,E$ or the contents of location E, is in BR. In DIV the high and low halves of the dividend from two accumulators are in AR and MQ respectively. In IDIV the one-word dividend from AC is in AR. The two types of division differ mainly in setting up the dividend; in both cases the algorithm processes a positive dividend to get a positive quotient.

In DIV if the dividend is negative (AR0 = 1), make it positive and set the negative dividend flag. To negate the dividend, move the low word to AR and the complement of the high word to MQ. Then move the negative of the low word back to MQ and the complement of the high word back to AR. Now the double length negative of the original dividend is in AR and MQ unless MQ is clear; in this event add 1 to AR to give the twos complement negative of the high word. Once the dividend is in positive form shift MQ left one place to close the hole between the two halves; in other words drop the low sign and get the 70-bit magnitude into AR1–35, MQ0–34.

If the IDIV dividend in AR is negative, negate it and set the negative dividend flag. Move the one word dividend in positive form to MQ and clear AR. Shift MQ left, as the algorithm operates on a double length dividend in both types of division although the high part is null in this case.

After the dividend is set up, compare the divisor with it to determine whether the division can be performed. Subtract the absolute value of the divisor from the high half of the dividend (if the divisor is positive, subtract it; if negative, add it). Since the dividend is positive, the result is also positive if the magnitude of the divisor is less than or equal to the number in AR. For a fixed fraction, the divisor is subtracted from the actual dividend and no overflow is allowed. For a fixed integer, AR is clear and the result is positive only for a zero divisor; the worst possible case is the division of $2^{35} - 1$ by 1, whose integral result can be accommodated. (Placing the one word dividend in MQ effectively multiplies it by 2^{-35}, making it the fractional part of a two word dividend with the binary point in the middle. The quotient is then a proper fraction, which is multiplied by 2^{35} simply by interpreting it as an integer.) Thus if the result of this initial subtraction is

positive, set Overflow and No Divide, and terminate the procedure so the processor goes on to the next instruction. Dividing by zero is of course meaningless. The reason for prohibiting a fractional division where the result would be greater than 1 is that it is impossible to determine the position of the binary point in the quotient. So it is up to the programmer to shift the dividend to the correct position beforehand. If the result of the initial subtraction is negative, the division can be performed and the processor goes into the division loop.

In division on paper, one subtracts out the divisor the number of times it goes into the dividend, then shifts the dividend one place to the left (or the divisor to the right) and again subtracts out. In binary computations the divisor goes into the dividend either once or not at all. Each subtraction of the divisor thus generates a single bit of the quotient. If the subtraction leaves a positive difference, *ie* if the dividend is larger than the divisor, a 1 is entered into the quotient. If the difference is negative, a 0 is entered. To compensate for subtracting too much, the hardware could add the divisor back into the dividend before going to the next subtraction step. But the PDP-10 algorithm instead shifts first and adds the divisor back in at the new position. It then continues to shift and add putting 0s into the quotient until the result again becomes positive. This procedure generates the same quotient without ever going back a step.

The hardware procedure is as follows. As each addition or subtraction is formed in the adder, put the result in AR shifted one place to the left with AR35 receiving a new bit of the dividend from MQ0, and shift MQ left bringing in a bit of the quotient at MQ35. The bit brought in is the complement of the sign from the adder: if the divisor does not go into the dividend, the resulting minus sign (1) produces a 0 quotient bit; if the divisor does go in, the plus sign gives a 1. Each step loads one bit of the quotient into MQ35, and the low half of the dividend is shifted out of MQ as the quotient is shifted in.

The first step is the test subtraction. In each subsequent step, subtract the absolute value of the divisor if the quotient bit generated in the previous step is 1, but add it back in if the quotient bit is 0. Since the divisor may have either sign, subtract it algebraically if its sign differs from the quotient bit, add it if its sign is the same.

The hardware executes 36 steps to generate 35 magnitude bits. The initial test step must give a 0, which serves as the sign since we are producing a positive quotient. In the final step put the result of the addition or subtraction directly in AR without shifting so the remainder is in the correct position, but shift MQ left putting the sign from the first step in MQ0 and bringing in the last quotient bit. (The bit dropped out of MQ0 is superfluous; it was brought into MQ35 when the hole was closed between the dividend halves.)

To complete the division we must make sure the remainder is correct and determine the correct signs of the results. Since the operations were performed on positive operands, the remainder should also be positive. A negative remainder indicates that too much has been subtracted. To correct this add the absolute value of the divisor back in. If the negative dividend flag is set, negate AR so the remainder has the sign of the original dividend.

Now move the corrected remainder to MQ and move the quotient to AR. If the negative dividend flag and the divisor sign are of opposite states, negate AR to produce the correct quotient sign. The correct quotient and remainder are now in AR and MQ ready for storage.

As an example of the way this algorithm operates, consider a division of 3-bit fixed fractions with a dividend of +.100100 and a divisor of +.101. By paper computation we obtain the quotient this way.

```
          .111
    101|100.100
        10 1
        10 00
         1 01
          110
          101
            1
```

Taking the processor registers to be four bits in length, AR contains 0.100, MQ has 0.100, and BR has 0.101. Before starting we close the hole changing MQ to 1.000. The sequence has four steps.

```
          0.100   1.000
        - 0.101
          1.111
 1 ←      1.111   0.000
        + 0.101
          0.100
 2 ←      1.000   0.001
        - 0.101
          0.011
 3 ←      0.110   0.011
        - 0.101
          0.001
 4        0.001  ←0.111
```

The quotient is in MQ at the right, the remainder in AR at the left.

FLOATING POINT ALGORITHMS

§1.1 explains floating point numbers and §2.6 discusses the general characteristics of floating point arithmetic. Exponent computations are done in the SC adder using the exponents and signs from the floating point operands. Remember, the sign is that of the whole number, not of the exponent. Although bits 1–8 of a floating point number represent an exponent in the range −128 to +127, the discussion is entirely in terms of the excess 128 exponents in positive form, *ie* the set of numbers 0–255. Computations generally use twos complement operations even though the exponent in a

negative number is a ones complement. The SC sign bit is used to detect exponent overflow and underflow.

After exponent calculations are complete, operations on the fractions are done by the fixed point logic in AR, BR and MQ. Bits 1–8 of AR and BR are filled with null bits, 0s in a positive number, 1s in a negative. Double length operands are in AR and MQ with MQ8–35 forming a magnitude extension of AR. In almost all circumstances the logic treats AR0–35 and MQ8–35 as a single 64-bit register; in all two-word shifting AR35 is connected to MQ8 and MQ0–7 is ignored. Except in division the fixed point calculation generates a double length fraction, which is shifted arithmetically (in right shifting the sign goes into AR1; in left shifting the sign is unaffected and 0s enter MQ35). Almost all floating point instructions normalize the result, thus making use of the low order part even though the instruction may store only the high order word.

Addition, Subtraction. E,0 or the word from location E is in BR, and AC is in AR. For subtraction move the negative of the subtrahend from BR to AR and move the minuend from AR to BR. This reduces subtraction to addition, so the rest of the algorithm is the same for both.

The initial objective is to determine the difference between the exponents and to determine which exponent is the larger. If the signs of the operands differ, add the exponents into SC. If the signs are the same, subtract the BR exponent from the AR exponent by adding the twos complement. Let x and y be the AR and BR exponents in positive form. The table below shows the calculations as a function of the operand signs, and the sign of the result in SC as a function both of the operand signs and the relative values of x and y.

AR+, BR+		AR+, BR−		AR−, BR+		AR−, BR−	
$+[x]$		$+[x]$		$-[255-x]$		$-[255-x]$	
$-[256-y]$		$-[255-y]$		$+[y]$		$+[1+y]$	
$-[256+x-y]$		$-[255+x-y]$		$-[255-x+y]$		$-[256-x+y]$	
SC+	SC−	SC+	SC−	SC+	SC−	SC+	SC−
$x \geqslant y$	$x < y$	$x > y$	$x \leqslant y$	$x < y$	$x \geqslant y$	$x \leqslant y$	$x > y$

As can be seen from the above, if AR already contains the number with the smaller exponent, the SC and AR signs differ. Hence if the SC and AR signs are the same, switch BR and AR so the number with the smaller exponent can be shifted. If the exponents are equal, the signs may or may not be the same but it matters not whether the transfer takes place.

To control the shifting we must now get the negative of the difference between the exponents. Let d be $|x-y|$. There are four cases as a function of the SC sign and whether the AR and BR signs are equal. The second column lists the present contents of SC, the third tells what must be done to arrive at $-[256-d]$ in SC.

SC+, AR0 = BR0	$+[d]$	Negate SC
SC+, AR0 ≠ BR0	$+[d-1]$	Complement SC

SC−, AR0 = BR0	$-[256 - d]$	Do nothing
SC−, AR0 ≠ BR0	$-[255 - d]$	Add 1 to SC

If $d < 64$ (indicated by a negative SC with a 0 in either SC1 or SC2) nullify AR1−8 and shift AR and MQ right d places so its bits correctly match the BR bits in order of magnitude. If $d > 64$ clear AR for its contents are of no significance.

Now move the larger exponent from BR to SC in positive form, nullify BR1−8, and add BR and AR into AR as fixed fractions. Finally enter the normalizing sequence.

This sequence first tests for a zero result. If AR and MQ8−35 are clear, bypass the rest of the procedure. If the fractional result has overflowed into AR8 (indicated by AR0 ≠ AR8 or AR8 = 1 and AR9−35 = 0), shift right and increase the exponent by one. The number is now normalized.

Complement the exponent in SC. If the instruction is not UFA and the number is not normalized go into the normalizing loop. In each step shift the double length fraction left and add 1 to the negative exponent (decreasing its magnitude by 1). Terminate the loop when the fraction is normalized, indicated by the sign and the MSB of the fraction being different (AR0 ≠ AR9) or the magnitude being ½ (AR9 = 1 and AR10−35 = 0).

If the instruction specifies rounding, adjust the high fraction so it is rounded and is in twos complement form if negative. The rounding is away from zero. For a positive result the high fraction must be increased if the low fraction is greater than half the value of the high fraction LSB. In a negative result the high fraction is a ones complement, which is one greater in magnitude than the twos complement. Hence it is already rounded and should be decreased in magnitude if the low fraction is < ½LSB. In either case add 2^{-27} into AR if MQ8 is 1 unless MQ9−35 is clear in a negative number. A 1 in MQ8 indicates a low fraction ≥ ½LSB in a positive number, ≤ ½LSB in a negative number. The condition that MQ9−35 not be zero in a negative number is the case where the low fraction is exactly ½LSB. If the high fraction is actually changed, renormalize it. A single normalizing shift is all that is required and it occurs in only two cases: a right shift when $1 - 2^{-27}$ is rounded, a left shift when $-½$ is changed to a correct twos complement.

Once the number has been normalized (and rounded if necessary) the exponent is in negative form. Thus if the SC sign bit is 0, set Overflow and Floating Overflow. If SC1 is also 0, the sign bit must have been changed by decreasing the exponent, so also set Floating Underflow (the maximum possible exponent overflow is 128 giving an SC contents of 777_8, and this can occur only in division). Insert the exponent in correct form into AR1−8.

The result is now ready to store from AR unless the instruction is in long mode. To ready the double length result subtract 27 from the positive exponent in SC. Save the high word in MQ, and move the low word to AR, but only if the decreased exponent is still positive. If the sign is 1, the true exponent of the low word is less than −128, so clear AR. (Note that this condition is also true if the low exponent is > 127, which can occur only if the high exponent is > 154.) If the low word is nonzero, shift AR right one place to put the fraction in bits 9−35 (remember that all shift operations

use MQ8–35), clear AR0 so the low word has a positive sign even if the double length fraction is negative, and insert the low exponent in positive form in bits 1–8. Finally switch AR and MQ so the high and low words are in correct position for storage.

Scaling. The 9-bit signed scale factor from bits 18 and 28–35 of E is in SC, and AC is in AR and BR. If the floating point number being scaled is positive, simply add the sign and exponent from BR0–8 to SC; if the number is negative, add the complement of BR0–8 to SC. Let x be the exponent in positive form and let y be the absolute value of the scale factor. There are only two cases,

$$\begin{array}{l} +[x] \\ +[y] \\ \hline +[x+y] \end{array} \qquad \begin{array}{l} +[x] \\ -[256-y] \\ \hline +[x-y] \end{array}$$

and in either the result is in positive form in SC.

Now enter the normalizing sequence described under floating addition. Only left shifting can occur bringing 0s in from MQ. The result can be zero, and exponent overflow or underflow can occur; but there is no rounding, and at the end the one-word result is in AR ready for storage.

Multiplication. E,0 or the word from location E is in BR, and AC is in AR. Place the AR exponent in positive form in SC, and add the positive form of the BR exponent to it. Since both are in excess-128 code, subtract 128. Save the result in the floating exponent register FE so SC can be used to control the multiplication of the fractions.

Nullify the exponent parts of AR and BR. Move the multiplier from BR to MQ and the multiplicand from AR to BR. Clear AR. Now multiply the fractions by the same procedure given for fixed point multiplication with the following differences:

- ♦ There are only 28 steps instead of 36.
- ♦ The shift register extension of AR for the construction of the product is MQ8–35. As the multiplier is shifted out, bits of the product come in at MQ8.
- ♦ In the final step place the adder output directly into AR but do not shift MQ – the low fraction is in MQ8–34, the correct position for normalization.

Clear MQ35, move the exponent back to SC, and enter the normalizing sequence described under floating addition. If the operands are normalized, at most one left shift is needed to normalize the result.

Division. The divisor, E,0 or the contents of location E, is in BR. The dividend from AC is in AR. In long mode the low half of the dividend from the second accumulator is in MQ; otherwise MQ is clear.

If the dividend is negative, make it positive and set the negative dividend flag. Except in long mode, negate the dividend simply by negating AR. For long mode follow the procedure given for DIV in the second paragraph of the fixed division algorithm. With a floating point operand the left MQ shift puts the low fraction in MQ8–34.

Place the AR exponent in positive form in SC. Subtract the magnitude of the BR exponent from it by adding the negative form of the exponent (ones complement) plus 1. Since the excess-128 factors cancel in the subtraction, add 128. Save the result in the floating exponent register FE so SC can be

used to control the division of the fractions.

Nullify the exponent parts of AR and BR. Subtract the absolute value of the divisor from the high half of the dividend. If the result is positive, indicating the divisor is less than or equal to the dividend, shift AR and MQ right and increase the exponent in SC by 1. Save the adjusted exponent in FE. The shift divides by 2, so if the operands are normalized, the dividend must now be less than the divisor.

Now divide the fractions by the same procedure given for fixed point fractional division with the following differences:

♦ Since the dividend has already been adjusted, the test in the first step stops the division only if the divisor is zero, or is unnormalized and less than the dividend. A normalized divisor cannot cause the quotient to overflow. If the result of the initial subtraction is positive, terminate the procedure and set Floating Overflow as well as Overflow and No Divide.

♦ Instead of 36 steps there are only 29 if the instruction specifies rounding, otherwise 28.

♦ The shift register extension of AR is MQ8–35. As quotient bits are brought in at MQ35, dividend bits are supplied to AR35 from MQ8. The shifting clears MQ0–7.

♦ The MQ shift in the final step places a 27-bit quotient fraction in MQ9–35 or a 28-bit fraction in MQ8–35.

♦ As in the fixed point algorithm generate the correct signed remainder, put it in MQ, and move the quotient to AR but leave it positive.

If the instruction specifies rounding, shift AR right placing the 27-bit fraction in the correct position, and if the bit shifted out of AR35 is 1, add it back into AR35 to round the positive quotient. If the quotient is zero bypass the rest of the procedure. The reaminder will also be zero except in an FDVL where the double length dividend is unnormalized and its high fraction is zero.

Complement the exponent in SC. If the instruction uses normalized operands the initial dividend adjustment guarantees that the quotient will be normalized. If it is not, shift AR left (bringing 0s into AR35) until a 1 appears in AR9, each time increasing the negative exponent by 1 (decreasing its magnitude).

Since the exponent is in negative form, if SC0 is 0, set Overflow and Floating Overflow. If SC1 is also 0, the sign bit must have been changed by decreasing the exponent, so also set Floating Underflow. Insert the exponent in correct form into AR1–8. If the negative dividend flag and the divisor sign (BR0) are of opposite states, negate AR to produce the correct quotient sign.

The quotient is now ready for storage from AR and the remaining operations are performed only for long mode. Save the quotient in BR and bring the high half of the original dividend from AC to AR. Put the dividend exponent in SC. Decrease its magnitude by 26 if the dividend was shifted right at the beginning to allow the division to be performed; otherwise decrease it by 27. Move the remainder to AR and insert the exponent in it provided the remainder is not zero and the exponent is within the proper range, −128 to 127 (the test is that the sign resulting from the exponent calculation is the same as the sign of the remainder). If the exponent is

outside that range clear AR; the assumption is that the remainder is of no significance (*ie* the exponent is too small). Move the remainder with its correct exponent from AR to MQ and put the quotient back in AR. The two words are now ready for storage.

Double Precision Division. The software routine that performs double precision floating point division and the algorithm it utilizes are given at the end of §2.11. FDVL performs the division

$$A/b = q + r2^{-27}/b$$

where q and r are the quotient and remainder. In a double precision division the divisor is of the form

$$B = b + d2^{-27}$$

Using the expansion

$$\frac{1}{x+y} = \frac{1}{x}\left[1 - \frac{y}{x} + \frac{y^2}{x^2} - \frac{y^3}{x^3} + \dots\right] \qquad (y^2 < x^2)$$

and letting $x = b$ and $y = d2^{-27}$ gives

$$\frac{A}{B} = \left(q + \frac{r2^{-27}}{b}\right)\left[1 - \frac{d2^{-27}}{b} + \frac{d^2 2^{-54}}{b^2} - \frac{d^3 2^{-81}}{b^3} + \dots\right]$$

Multiplying out and gathering like terms gives

$$\frac{A}{B} = q + \frac{1}{b}(r - qd)2^{-27} - \frac{d}{b^2}(r - qd)2^{-54} + \frac{d^2}{b^3}(r - qd)2^{-81} - \dots$$

where the first two terms on the right are those in the equation at the bottom of page 2-67.

The ratio of adjacent terms is

$$\frac{T_{n+1}}{T_n} = \frac{-d2^{-27}}{b}$$

In an alternating convergent series, the error due to truncation is smaller than the first term dropped. Therefore

$$|Error| < \frac{d2^{-27}}{b} T_n$$

Since the maximum value of d is less than 1 and the minimum value of b (normalized) is ½,

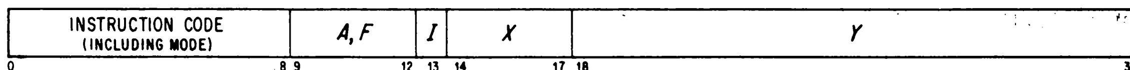
$$|Error| < T_n 2^{-26}$$

APPENDIX G

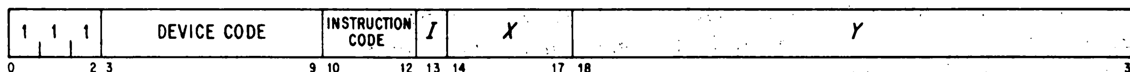
BIT ASSIGNMENTS

The drawing on pages G2 and G3 shows the formats of the various types of words used by the processor. Bit assignments in the condition and data words for the IO instructions will be added later.

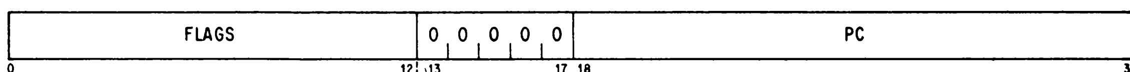
BASIC INSTRUCTIONS

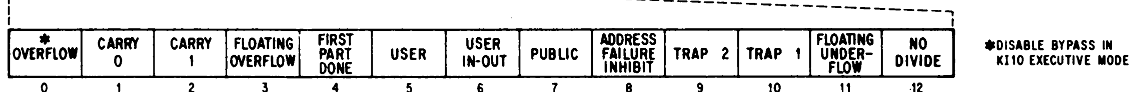


IN-OUT INSTRUCTIONS

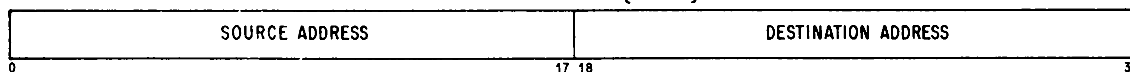


PC WORD

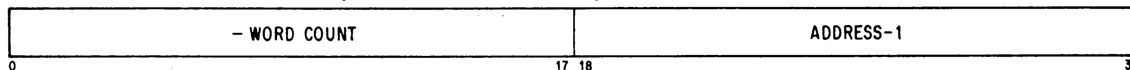




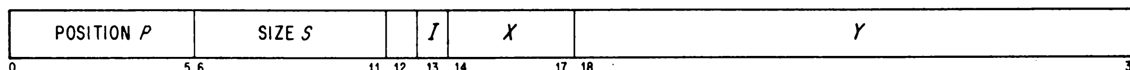
BLT POINTER {XWD}



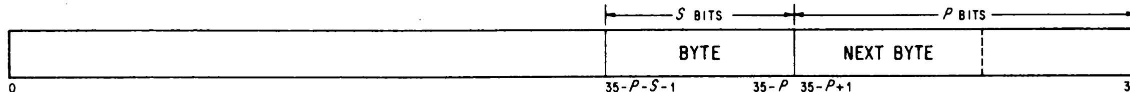
BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



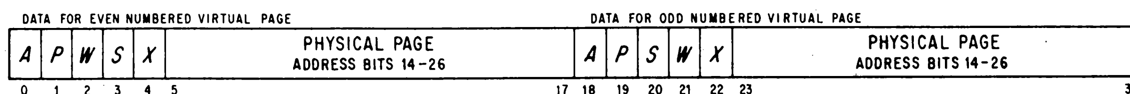
BYTE POINTER



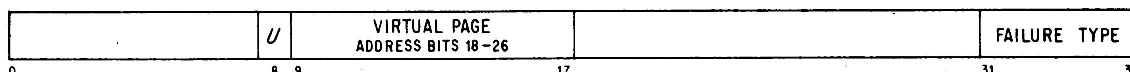
BYTE STORAGE



PAGE MAP WORD

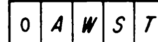


PAGE FAIL WORD

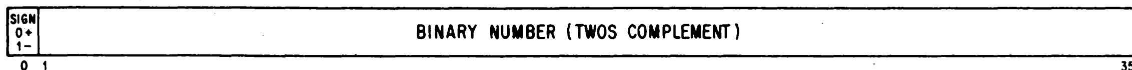


20 SMALL USER VIOLATION
21 PROPRIETARY VIOLATION
22 PAGE REFILL FAILURE
23 ADDRESS FAILURE

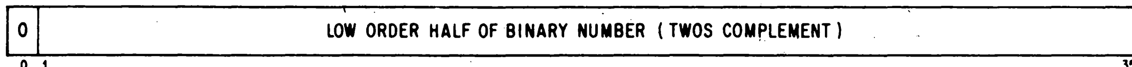
IF BIT 31 IS 0, BITS 31-35 HAVE THIS FORMAT



FIXED POINT OPERANDS



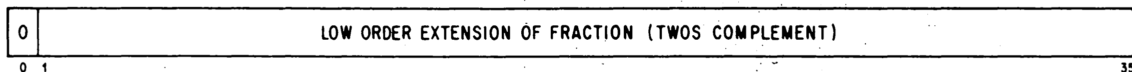
LOW ORDER WORD IN DOUBLE LENGTH FIXED POINT OPERANDS



FLOATING POINT OPERANDS

LOW ORDER WORD IN SOFTWARE DOUBLE LENGTH FLOATING POINT OPERANDS

LOW ORDER WORD IN HARDWARE DOUBLE LENGTH FLOATING POINT OPERANDS



POWERS OF TWO

2^N	N	2^{-N}
1	0	1.0
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

IN-OUT DEVICE BIT ASSIGNMENTS

DEVICE	CODE	FUNCTION	0/18	1/19	2/20	3/21	4/22	5/23	6/24	7/25	8/26	9/27	10/28	11/29	12/30	13/31	14/32	15/33	16/34	17/35
APR	000	CONO	CLR PDL OV	I/O RESET		CLR ADR BREAK	CLR MEM PROT	CLR NXM	CLR CLOCK ENABLE	SET CLOCK ENABLE	CLR CLOCK FLAG	CLR FOV ENABLE	SET FOV ENABLE	CLR FOV FLAG	CLR AROV ENABLE	SET AROV ENABLE	CLR AROV FLAG	PIA		
		CONI		PDL OV FLAG	IOT USER FLAG	ADR BREAK FLAG	MEM PROT FLAG	NXM FLAG		CLOCK ENABLE	CLOCK FLAG		FOV ENABLE	FOV FLAG	TRAP OFFSET	AROV ENABLE	AROV FLAG	PIA		
		DATAO	PROTECTION REGISTER (LH 0-7) RELOCATION REGISTER (RH 18-25)																	
		DATAI	← 36 DATA SWITCHES →																	
PI	004	CONO	CLEAR POWER FAIL FLAG	CLEAR PARITY FLAG	CLEAR PARITY ENABLE	SET PARITY ENABLE		CLEAR PI SYSTEM	REQUEST INTERRUPT ON CHANNEL	TURN ON CHANNELS	TURN OFF CHANNELS	TURN OFF PI ACTIVE	TURN ON PI ACTIVE	SELECT CHANNELS 1-7 FOR BITS 24, 25, 26						
		CONI	POWER FAILURE FLAG	PARITY ERROR FLAG	PARITY ENABLE	INTERRUPT IN PROGRESS ON CHANNELS 1-7							PI ACTIVE	CHANNEL ACTIVE 1-7						
		DATAO	← 36 BITS TO THE MEMORY INDICATORS →																	
ADC A-D CONVERTER (AD10)	024																			
PTP	100	CONO													BINARY	BUSY	DONE FLAG	PIA		
		CONI												OUT OF TAPE	BINARY	BUSY	DONE FLAG	PIA		
		DATAO											HOLE 8 UNLESS BINARY	HOLE 7 UNLESS BINARY	HOLE 6	HOLE 5	HOLE 4	HOLE 3	HOLE 2	HOLE 1
PTR	104	CONO													BINARY	BUSY	DONE FLAG	PIA		
		CONI										TAPE FLAG			BINARY	BUSY	DONE FLAG	PIA		
		DATAI	← 36 BIT WORD IF BINARY, 8 BITS (28-35) IF NOT BINARY →																	
TTY	120	CONO							TEST FLAG	TTI BUSY CLEAR	TTI FLAG CLEAR	TTO BUSY CLEAR	TTO FLAG CLEAR	TTI BUSY SET	TTI FLAG SET	TTO BUSY SET	TTO FLAG SET	PIA		
		CONI							TEST FLAG					TTI BUSY	TTI FLAG	TTO BUSY	TTO FLAG	PIA		
		DATAO											8 BIT CHARACTER TO TELETYPE							
		DATAI											8 BIT CHARACTER FROM KEYBOARD							
LPT (LP10)	124	CONO								CLEAR PRINTER			BUSY	DONE FLAG	ERROR PIA			DONE PIA		
		CONI							128 CHAR	96 CHAR		ERROR	BUSY	DONE FLAG	ERROR PIA			DONE PIA		
		DATAO	FIRST CHARACTER							SECOND CHARACTER							THIRD			
			CHARACTER			FOURTH CHARACTER							FIFTH CHARACTER							
PLT (XY10)	140	CONO														BUSY	DONE FLAG	PIA		
		CONI													POWER ON	BUSY	DONE FLAG	PIA		
		DATAO													PEN RAISE	PEN LOWER	-X DRUM UP	+X DRUM DOWN	+Y CARRIAGE LEFT	-Y CARRIAGE RIGHT
CR (CR10)	150	CONO						CLEAR READER	OFFSET CARD		READ CARD	TROUBLE ENABLE	CLEAR DATA MISSED	READY ENABLE	CLEAR END OF FILE	CLEAR END OF CARD	CLEAR DATA FLAG	PIA		
		CONI	READER TABLE ENAB	READER READY ENAB	PICK ERROR	PHOTO CELL ERROR	CARD MOTION ERROR	READER STOP	CARD IN READER	HOPPER STACKER	READING CARD	READER TROUBLE	DATA MISSED	READY TO READ	END OF FILE	END OF CARD	DATA FLAG	PIA		
		DATAI							ROW 12	ROW 11	ROW 0	ROW 1	ROW 2	ROW 3	ROW 4	ROW 5	ROW 6	ROW 7	ROW 8	ROW 9

THIS IS A TEMPORARY PAGE

DEVICE	CODE	FUNCTION	0/18	1/19	2/20	3/21	4/22	5/23	6/24	7/25	8/26	9/27	10/28	11/29	12/30	13/31	14/32	15/33	16/34	17/35
DSK (RC10)	170	CONO	SELECT SECTOR CTR		CLEAR DISK DESIG ERROR	CLR TRACK SELECT ERROR	CLEAR DISK NOT READY	CLEAR PWR SUP FAILURE	CLR DISK PAR ERR	CLR CHAN DATA PAR ERR	CLR CHAN CONTROL PAR ERR	CLEAR NXM	CLEAR ILLEGAL WRITE	CLEAR OVER-RUN	WRITE CHAN CNTL WD	STOP CLEAR BUSY	CLEAR DONE	PIA		
		CONI LH						MAINT SECTOR SELECT	PROTECT LOWER	← BOUNDARY SWITCHES (BCD) → DISK		TRACK 100'S		10'S				1'S		
		CONI RH	DATA XFER IN PROG	SEARCH ERROR	DISK DESIG ERROR	TRACK SELECT ERROR	DISK NOT READY	PWR SUPPLY FAIL	DISK PARITY ERROR	CHANNEL DATA PAR ERR	CHANNEL CONTROL PAR ERR	NXM	ILLEGAL WRITE	OVER-RUN	CHAN CNTL WD WRITTEN	BUSY	DONE	PIA		
		DATAO LH	DISK SELECT		← TRACK (BCD) → 100'S		10'S				1'S			← SECTOR (BCD) →	10'S			1'S		
		DATAO RH	INITIAL PARITY						DISABLE PAR ERR STOP	DISABLE MEMORY PAR ERR STOP	WRITE	INITIAL CHANNEL CONTROL WORD ADDRESS								WRITE EVEN PARITY
		DATAI RH	PARITY REGISTER								SECTOR CTR SELECTED		← SECTOR COUNTER (BCD) →		10'S			1'S		
DTC (TD10)	320	CONO	STOP	GO FORWARD	GO REVERSE	DELAY INHIBIT	SELECT	CLEAR SELECT & UNIT N/E	TRANSPORT NUMBER			FUNCTION NUMBER: 0 = NOTHING, 1 = READ ALL, 2 = READ BLOCK #, 3 = READ DATA, 4 = WRITE MK & TM, 5 = WRITE ALL, 6 = WRITE BLOCK #, 7 = WRITE DATA			DATA PIA			FLAGS PIA		
		CONI	STOP	GO FORWARD	GO REVERSE		SELECT	DESELECT	TRANSPORT NUMBER						DATA PIA			FLAGS PIA		
		DATAO	← 36 BIT WORD →																	
		DATAI	← 36 BIT WORD →																	
DTS	324	CONO	PARITY ERR ENABLE	DATA MISSED ENABLE	JOB DONE ENABLE	ILLEGAL OPERATION ENABLE	END ZONE ENABLE	BLOCK MISSED ENABLE											STOP ALL TRANSPORT	FUNCTION STOP
		CONI LH	PARITY ERR ENABLE	DATA MISSED ENABLE	JOB DONE ENABLE	ILLEGAL OPERATION ENABLE	END ZONE ENABLE	BLOCK MISSED ENABLE	DELAY IN PROGRESS	ACTIVE	UP TO SPEED	BLOCK NUMBER	REVERSE CHECK	DATA	FINAL DATA	CHECKSUM	IDLE	BLOCK NUMBER READ		FUNCTION STOP
		CONI RH	PARITY ERROR	DATA MISSED	JOB DONE	ILLEGAL OPERATION	END ZONE	BLOCK MISSED	WRITE LOCK ON	WRTM SWITCH ON	INCOMPLETE BLOCK		MARK TRACK ERROR	SELECT ERROR					FLAG PI REQUEST	DATA REQUEST
		DATAO RH				UP TO SPEED TEST														MARK TRACK DATA
		DATAI LH																PARITY		
		DATAI RH	REGISTER			READ/WRITE REGISTER								NOT MK BN SPACE	NOT MK BN END	NOT MK DATA SYNC	NOT MK REV DATA END	NOT MK DATA	NOT MK FWD DATA END	NOT MK BN SYNC
DLS (DC10)	240	CONO													CLEAR DLS	SET DATA TERM RDY	RESET SCANNER	PIA		
		CONI													DTR DISABLED	TRANS FLAG	RCVR FLAG	PIA		
DC10B/ DC10E		DATAO LH												USE LINE NO.	LINE NUMBER					
		DATAO RH										TRANS DISAB	BIT 8 CAUSE PI	BIT 7 OFF HOOK	BIT 6 CRQ	BIT 5 DPR	BIT 4 NB8	BIT 3 NB4	BIT 2 NB2	BIT 1 NB1
		DATAI LH													LINE NUMBER					
		DATAI RH										RCVR DATA	BIT 8	BIT 7 OFF HOOK	BIT 6 DLO	BIT 5 PND	BIT 4 ACR	BIT 3 CB	BIT 2 CE	BIT 1
TMC (TM10)	340	CONO	UNIT			PARITY 0 = EVEN 1 = ODD	CORE DUMP	FUNCTION: 0 = NO-OP, 1 = REWIND, 2 = READ, 3 = RD COMPARE, 4 = WRITE, 5 = WRITE EOF, 6 = SPACE FWD, 7 = SPACE REV, 10 = NO-OP, PI INT ON TUR, 12 = READ ACRS RECORD, 13 = RD/CMP ACRS RECD, 14 = WR W/LONG EOR, 16 = SPACE FILE, 17 = SPACE FILE REV				NEXT UNIT ENAB	DENSITY 0 = 200, 1 = 556, 2 = 800, 3 = (800)		FLAG PIA			DATA PIA ‡		
		CONI RH	UNIT			PARITY	CORE DUMP					NEXT UNIT ENAB	DENSITY		FLAG PIA			DATA PIA		
		CONI LH																NEXT UNIT		
		DATAO 7-CHN OR CORE DMP	← 36 BIT WORD → (7 CHN: SIX 6 BIT CHARACTERS AS IN DATAI BELOW)																	
		DATAO 9-CHN	← 1 ST CHARACTER →								← 2ND CHARACTER →								←	
			← 3 RD CHARACTER →						← 4 TH CHARACTER →								IGNORED			
		DATAI 7-CHN	← 1 ST CHARACTER →						← 2 ND CHARACTER →						← 3 RD CHARACTER →					
			← 4 TH CHARACTER →						← 5 TH CHARACTER →						← 6 TH CHARACTER →					
		DATAI 9-CHN	← 1 ST CHARACTER →								← 2ND CHARACTER →								←	
			← 3 RD CHARACTER →						← 4 TH CHARACTER →								PAR ERR 1ST CHAR	PAR ERR 2ND CHAR	PAR ERR 3RD CHAR	PAR ERR 4TH CHAR
		DATAI 9-CHN CORE DMP	← 1 ST CHARACTER →								← 2ND CHARACTER →									
			← 3 RD CHARACTER →						← 4 TH CHARACTER →								5 TH CHARACTER →			
TMS	344	CONO														CLEAR * DATA PAR ERR	CLEAR * WT CW DONE	WRITE * CTRL WORD	XFER ‡ CRC	STOP
		CONI LH												CW * PAR ERR	NON-EX * MEM	DATA * PAR ERR	WT CW * DONE	CHARACTER COUNTER		
		CONI RH	XPORT HUNG	REWINDING	LOAD POINT	ILLEG OP	PAR ERROR	END OF FILE	END OF TAPE	READ COMPARE ERROR	RECORD LENGTH INCORRECT	DATA LATE	BAD TAPE	JOB DONE	TAPE UNIT RDY (TUR)	CHANNEL * ERROR	WRITE LOCK	7-CHAN TAPE	NEXT UNIT	DATA ‡ REQ
		DATAO RH										INITIAL CHANNEL CONTROL WORD ADDRESS								WRITE EVEN PARITY
CCI (DA10)	014	CONO						CLR SELF TEST	SET SELF TEST	CLR FM10 FULL	SET FM10 FULL	CLR FM10 EMPTY	SET FM10 EMPTY	CLR TO10 FULL	SET TO10 FULL	CLR TO10 EMPTY	SET TO10 EMPTY	PIA		
		CONI							SELF TEST (1)		FM10 FULL (1)		FM10 EMPTY (1)		TO10 FULL (1)		TO10 EMPTY (1)	PIA		
		DATAI DATAO FOR PDP8	1ST PDP 8 BYTE (A)						2ND PDP 8 BYTE (B)						3RD PDP 8 BYTE (C)					
			← 36 BITS →																	
		DATAI DATAO FOR PDP9	← 36 BITS →																	
			1ST PDP 9 BYTE (A)									2ND PDP 9 BYTE (B)								

‡ BITS FOR TM10A ONLY

* BITS FOR TM10B ONLY

DEC-10-AMZC-D

MACRO-10 ASSEMBLER
PROGRAMMER'S REFERENCE MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Edition April 1967
2nd Printing October 1967
3rd Edition (Rev) August 1968
4th Edition (Rev) June 1969
5th Edition (Rev) October 1969
6th Edition (Rev) August 1970
7th Edition (Rev) April 1972

Copyright © 1967, 1968, 1969, 1970, 1971, 1972 by
Digital Equipment Corporation

The material in this manual is for information
purposes and is subject to change without notice.

The following are trademarks of Digital Equipment
Corporation, Maynard, Massachusetts:

DEC	PDP
FLIP CHIP	FOCAL
DIGITAL	COMPUTER LAB

CONTENTS

CHAPTER 1	INTRODUCTION	205
1.1	MACRO-10 LANGUAGE - STATEMENTS	206
1.2	INSTRUCTION WORD FORMATS	206
1.2.1	Primary Instruction Format	207
1.2.2	Input/Output Instruction Format	208
1.3	COMMUNICATION WITH MONITORS	209
1.4	OPERATING PROCEDURES	209
1.5	MACRO STATEMENTS	209
1.5.1	Symbols	209
1.5.2	Labels	210
1.5.3	Symbolic Addresses	210
1.5.4	Operators	211
1.5.5	Symbolic Operators	211
1.5.6	Operands	212
1.5.7	Symbolic Operands	212
1.5.8	Comments	213
1.6	STATEMENT PROCESSING	213
1.6.1	Order of Statement Evaluation	214
1.6.2	Order of Expression Evaluation	214
1.7	USER-DEFINED SYMBOLS	215
1.7.1	Direct Assignment Statements	215
1.7.2	Local and Global Symbols	216
1.7.3	Deleted Symbols	217
1.8	NUMBERS	218
1.8.1	Arithmetic and Logical Operations	219
1.8.2	Evaluating Expressions	219
1.8.3	Numeric Terms	220
1.8.4	Binary Shifting	221
1.8.5	Left Arrow Shifting	222
1.8.6	Floating Point Decimal Numbers	222
1.8.7	Fixed Point Decimal Numbers	222
1.9	ADDRESS ASSIGNMENTS	223
1.9.1	Setting and Referencing the Location Counter	224
1.9.2	Indirect Addressing	224
1.9.3	Indexing	224
1.10	LITERALS	225

CHAPTER 2 MACRO-10 ASSEMBLER STATEMENTS - PSEUDO-OPS 227

2.1 ADDRESS MODE: RELOCATABLE OR ABSOLUTE 227

2.1.1 Relocation Before Execution - PHASE and DEPHASE Statements 229

2.2 NAMING PROGRAMS 230

2.2.1 Program Subtitles 231

2.3 PROGRAM ORIGIN 231

2.3.1 HISEG Statements - The HISEG Pseudo-Op Statement 232

2.3.2 TWOSEG Statements 232

2.4 ENTERING DATA 233

2.4.1 RADIX Statements 233

2.4.2 Entering Data Under the Prevailing Radix 234

2.4.3 DEC and OCT Statements 234

2.4.4 Changing the Local Radix for a Single Numeric Term 235

2.4.5 RADIX 50 Statement 236

2.4.6 EXP Statement 236

2.4.7 Z Statement 236

2.5 INPUT DATA WORD FORMATTING 236

2.5.1 BYTE Statement 236

2.5.2 POINT Statement - Handling Bytes 237

2.5.3 IOWD Statement: Formatting I/O Transfer Words 239

2.5.4 XWD Statement: Entering Two Half-Words of Data 239

2.5.5 Text Input 240

2.5.5.1 ASCII, ASCIZ, and SIXBIT Statement 240

2.5.6 Reserving Storage 241

2.5.6.1 Reserving a Single Location 242

2.5.7 VAR Statements 243

2.5.8 BLOCK Statements 243

2.5.9 END Statements 243

2.5.10 LIT Statements 244

2.5.11 Multi-Program Assembly 244

2.5.12 PASS2 Statements 245

2.5.13 PURGE Statements 245

2.5.14 XPUNGE Statements 245

2.5.15 Linking Subroutines 246

2.5.15.1 EXTERN Statements 246

2.5.15.2 INTERN Statements 247

2.5.15.3 ENTRY Statements 247

2.6	SUPPRESSION OF SYMBOLS	248
2.6.1	SUPPRESS SYMBOL Statement	248
2.6.2	ASUPPRESS Statement	248
2.6.3	Listing Control Statements	249
2.7	CONDITIONAL ASSEMBLY	252
2.8	ASSEMBLER CONTROL STATEMENTS	253
2.8.1	REPEAT Statements	253
2.8.2	OPDEF Statements	254
2.8.3	SYN Statements	255
2.8.4	Extended Instruction Statements	256
2.9	MULTI-FILE ASSEMBLY	257
2.9.1	UNIVERSAL Name	257
2.9.2	SEARCH Name	258
CHAPTER 3	MACROS	259
3.1	DEFINITION OF MACROS	259
3.2	MACRO CALLS	260
3.3	MACRO FORMAT	261
3.4	CREATED SYMBOLS	262
3.5	CONCATENATION	263
3.6	DEFAULT ARGUMENTS	264
3.7	INDEFINITE REPEAT	265
3.8	NESTING AND REDEFINITION	266
3.8.1	ASCII Interpretation	268
CHAPTER 4	ERROR DETECTION	269
4.1	SINGLE-LETTER ERROR CODES	269
4.2	ERROR MESSAGES	275
4.2.1	LOOKUP Errors	277
4.2.2	MACRO I/O Error Messages	278
CHAPTER 5	RELOCATION	279
CHAPTER 6	ASSEMBLY OUTPUT	283
6.1	ASSEMBLY LISTING	283
6.2	BINARY PROGRAM OUTPUT	284
6.2.1	Relocatable Binary Programs - LINK Format	284

6.2.1.1	LINK Formats for the Block Types	285
6.2.2	Absolute Binary Programs	288
6.2.2.1	RIM10B Format	288
6.2.2.2	RIM10 Format	289
6.2.2.3	RIM Format	290
6.2.2.4	END Statements	290
CHAPTER 7	PROGRAMMING EXAMPLES	293
APPENDIX A	OP CODES, PSEUDO-OPS, AND MONITOR I/O COMMANDS	307
A.1	ASSEMBLER PSEUDO-OPS AND MONITOR CO COMMANDS	307
A.2	MACHINE MNEMONICS AND OCTAL CODES	309
APPENDIX B	SUMMARY OF PSEUDO-OPS	311
B.1	PSEUDO-OPS	311
B.1.1	Conditional Assembly Statements	313
APPENDIX C	SUMMARY OF CHARACTER INTERPRETATIONS	315
APPENDIX D	STORAGE ALLOCATION	319
APPENDIX E	TEXT CODES	323
APPENDIX F	RADIX 50 REPRESENTATION	325
APPENDIX G	SUMMARY OF RULES FOR DEFINING AND CALLING MACROS	327
G.1	ASSEMBLER INTERPRETATION	327
G.2	CHARACTER HANDLING	327
G.2.1	Blanks	327
G.2.2	Brackets	327
G.2.3	Parentheses	328
G.2.4	Commas	328
G.2.5	Semicolons	328
G.2.6	Carriage Return	328
G.2.7	Back-Slash	328
G.3	CONCATENATION	328

APPENDIX H	.	OPERATING INSTRUCTIONS	331
	H.1	REQUIREMENTS	331
	H.2	INITIALIZATION	331
	H.3	COMMANDS	332
	H.3.1	General Command Format	332
	H.3.2	Disk File Command Format	332
	H.4	SWITCHES	334

Chapter 1
Introduction

MACRO-10 is the symbolic assembly program for the PDP-10, and operates in a minimum of 7K pure plus 1K impure core memory in all PDP-10 systems. MACRO-10 is a two-pass assembler. It is completely device independent, allowing the user to select standard peripheral devices for input and output files. For example, a terminal can be used for input of the symbolic source program, DECtape for output of the assembled binary object program, and a line printer can be used to output the program listing.

This assembler performs many useful functions, making machine language programming easier, faster, and more efficient. Basically, the assembler processes the PDP-10 programmer's source program statements by translating mnemonic operation codes to the binary codes needed in machine instructions, relating symbols to numeric values, assigning relocatable or absolute core addresses for program instructions and data, and preparing an output listing of the program which includes notification of any errors detected during the assembly process.

MACRO-10 also contains powerful macro capabilities which allow the programmer to create new language elements, thus expanding and

adapting the assembler to perform specialized functions for each programming job.

1.1 MACRO-10 LANGUAGE - STATEMENTS

MACRO-10 programs are usually prepared on a terminal, with the aid of a text editing program, as a sequence of statements. Each statement is normally written on a single line and terminated by a carriage return-line feed sequence. MACRO-10 statements are virtually format free; that is, elements of a statement are not placed in numbered columns with rigidly controlled spacing between elements, as in punched-card oriented assemblers.

There are four types of elements in a MACRO-10 statement which are separated by specific characters. These elements are identified by the order of appearance in the statement, and by the separating, or delimiting, character which follows or precedes the elements.

Statements are written in the general form:

```
     label: operator    operand,operand;comments(carriage return-line feed)
```

The assembler converts statements written in the foregoing form and translates them into machine instruction words. The formats used by the machine instructions are described in the following paragraphs.

1.2 INSTRUCTION WORD FORMATS

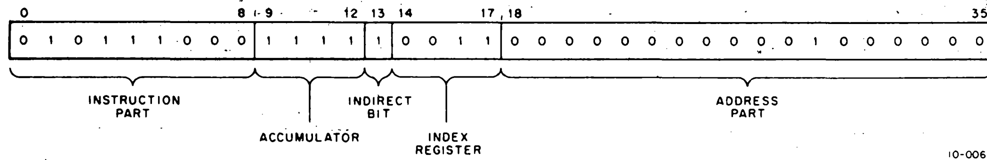
There are two types of machine instruction word formats: primary and input/output.

The PDP-10 machine instructions are fully described in the PDP-10 System Reference Manual and listed alphabetically in Appendix A of this manual. Monitor I/O commands, or programmed operators have the same formats. (See monitor manuals.)

The primary instruction statements may have two operands: (1) an accumulator address and (2) a memory address. A memory address may be modified by indexing and indirect addressing.

1.2.1 Primary Instruction Format

After processing primary instruction statements, the assembler produces machine instructions in the general 36-bit word format shown below:



In general, the mnemonic operation code, or operator, in the symbolic statement is translated to its binary equivalent and placed in bits 0-8 of the machine instruction. The address operand is evaluated and placed in the address part (bits 18-35 of the machine instruction. The assembler assigns sequential binary addresses to each statement as it is processed by means of the location counter. Labels are given the current value of the location counter and are stored in the assembler's symbol table, where the corresponding binary addresses can be found if another instruction uses the same symbol as an address reference.

Any one of 16 possible accumulators may be specified in an instruction by identifying them symbolically or numerically as operands in the statement followed by a comma. The indirect address bit is set to 1 when the character @ prefixes a memory reference. Indexing is specified by writing the index register used in parentheses immediately following the memory reference. (All PDP-10 accumulators, except accumulator 0, may be used as index registers.) Actually, expressions enclosed in parentheses (in the index register position) are evaluated as 36-bit quantities; their halves are exchanged, and then each half is added into the corresponding half of the binary word being assembled. For example, the statements

```
MOVSI AC,(1.Ø)          ;MOVE 1.Ø TO AC
MOVSI AC,(SIXBIT /DSK/)
```

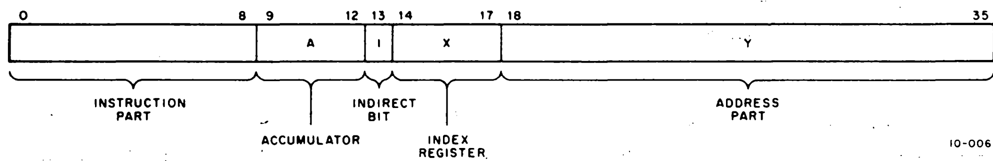
are equivalent to

```
MOVSI AC,2Ø14ØØ         ;MOVE 1.Ø TO AC
MOVSI AC,446353
```

To illustrate this general view of assembler processing, here is a typical symbolic instruction. Assume that AC17, TEMP and XR are defined symbols, with values of 17, 100, and 3, respectively.

```
LABEL:  ADD AC17,@TEMP(XR)    ;STATEMENT EXAMPLE
```

This is processed by the assembler and stored as a binary machine instruction like this:



The mnemonic instruction code, ADD, has been translated to its octal equivalent, 270, and stored in bits 0-8. The first operand specifies accumulator 17_8. The effective memory address will be found at execution time by adding the contents of index register 3 to the value of TEMP, then taking this value as the address of the word whose address points to the word to be added to AC17.

A comment following a semicolon does not affect the program in any way, but it is printed in the output listing.

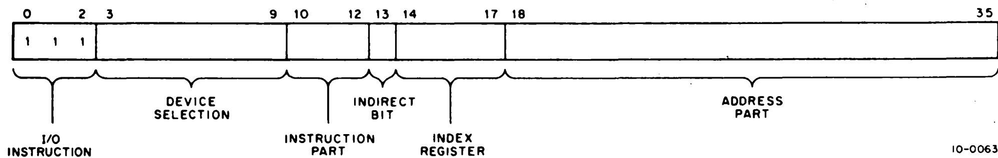
1.2.2 Input/Output Instruction Format

There are eight PDP-10 I/O statements; in each statement the first operand is either a peripheral device number or a device mnemonic (see PDP-10 System Reference Manual for complete list). The second operand is a memory address. For example,

```
READ:  DATAI PTR,@NUM(4)
```

requests that data be read in from a paper-tape reader, to be stored at the indirect, indexed, address given.

The format for I/O instruction words is shown below:



1.3 COMMUNICATION WITH MONITORS

Programs assembled with MACRO-10 which operate under executive control of a monitor must use monitor facilities for device independent I/O services. This is done by means of programmed operators (operation codes 040 through 077) such as CALL. INIT, LOOKUP, IN, OUT, and CLOSE.

Additional monitor commands are available to allow the user program to exercise control over central processor trapping, to modify its memory allocation, and other services, which are described in the monitor programmer's manuals.

Monitor commands are listed in Appendix A.

1.4 OPERATING PROCEDURES

Commands for loading and executing MACRO-10 are contained in Appendix H.

1.5 MACRO STATEMENTS

As previously stated (paragraph 1.1) macro statements consist of a label, an operator, an operand and optional comments.

The assembler interprets and processes these statements, generating one or more binary instructions or data words, or performaing an assembly process. A statement must contain at least one of these elements and may contain all four types. Some statements are written with only one operand; but others may have many. To continue a statement on the following line, the control (CTRL) left arrow (←), echoed as ←↓, is used before the carriage return-line feed sequence (← ↓ or ❩). Examples of program statements are given in Chapter 7, Figures 7-1 and 7-3.

Statement labels, operators and operands may be represented either numerically or symbolically. The assembler interprets all symbols and replaces them with a numeric (binary) value.

1.5.1 Symbols

The programmer may create symbols to use as statement labels, as operators and as operands. A symbol may consist of any

combination of from one to six characters of the following set:

The 26 letters, A-Z
Ten digits, 0-9
Three special characters: $ (Dollar Sign)
% (Percent)
. (Period)

The foregoing character set is the Radix-50 character set.

Any statement character which is not in the Radix-50 set is treated as a symbol delimiter when encountered by the assembler.

If the first characters of a symbol are numeric, the symbol is treated as through the numeric characters were not present. If the first character is a period, it must not be followed by a digit. Spaces must not be embedded in symbols. A symbol may actually have more than six characters, but only the first six are meaningful to MACRO-10.

MACRO-10 accepts programs written using both upper and lower case letters and symbols (e.g., programs written using the Teletype Model 37). Lower case letters are treated as upper case in symbols; additional special characters, and lower case letters elsewhere, are taken without change.

1.5.2 Labels

A label is the symbolic name created by the source programmer to identify a statement. If present, the label is written as the first item in a statement and is terminated by a colon (:). (Refer to paragraph 1.5.1 for a description of how symbolic names are formed.)

1.5.3 Symbolic Addresses

A symbol used as a label to specify a symbolic address must appear first in the statement and must be immediately followed by a colon (:). When used in this way, a symbol is said to be defined. A defined-symbol can reference an instruction or data word at any point in the program.

A label can be defined with only one value; if a programmer attempts to redefine a label with a different value, the second value is

ignored and an error is indicated (see Chapter 4 for error messages). The following are legal labels:

```
$SUM:
ABC: DEF:          (Both labels are legal)
FOO
```

The following are illegal:

```
7ABC:              (First character must not be a digit.)
LAB :              (Colon must immediately follow label.)
```

If too many characters are used in a label, only the first six characters given are used. For example the label ABCDEFGH: is recognized by the assembler as being ABCDEF:.

Labels are used for programmer reference as addresses for jump instructions, for loops and for debugging.

1.5.4 Operators

An operator may be one of the mnemonic machine instruction codes (see DECsystem-10 System Reference Manual), a command to Monitor, or a pseudo-operation code which directs assembly processing. These assembly pseudo-op codes are described in this manual, and listed with all other assembler defined operators in Appendix A.

Programmers may extend the power of the assembler by creating their own pseudo-operators (see OPDEF pseudo-op).

An operator may be a macro name, which calls a user-defined macro instruction. Like pseudo-ops, macros direct assembly processing; but, because of their unique power to handle repetitions and to extend and adapt the assembly language, macros are considered separately (see Chapter 3). Operators are terminated with a space or tab.

1.5.5 Symbolic Operators

Symbols used as operators must be predefined by the assembler or by the programmer. If a statement has no label, the operator may appear first in the statement, and must be terminated by a space, tab, or carriage return. The following are examples of legal operators:

```
MOV     (A mnemonic machine instruction operator.)
LOC     (An assembler pseudo-op.)
ZIP     (Legal only if defined by the user.)
```

1.5.6 Operands

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, or the input data or arguments or a pseudo-op or macro instruction. In each case, the interpretation of operands in a statement depends on the statement operator. Operands are separated by commas, and terminated by a semicolon (;) or by a carriage return-line feed.

In the mnemonic machine instruction and UUO call set, if an operand is followed by a comma (spaces in the line are ignored) then the operand is identified as an accumulator (see instruction format description in paragraph 1.2.1). If an operand is not followed by a comma, then it is viewed as an address (either indexed or indirect if negative).

1.5.7 Symbolic Operands

Symbols used as operands must have a value defined by the user. These may be symbolic references to previously defined labels where the argument to be used by this instruction are to be found, or the values of symbolic operands may be constants or character strings. If the first operand references an accumulator, it must be followed by a comma.

```
TOTAL:  ADD AC1,TAG)
```

The first operand, AC1, specifies an accumulator register, determined by the value given to the symbol AC1 by the user. The second operand references a memory location, whose name or symbolic address is TAG. If the user has equated AC1 to 17, and the assembler has assigned TAG to the binary address, 000537, then the assembler inserts 17 in the accumulator field (bits 9-12) and 000537 in the address field (bits 18-35) of this instruction. If an accumulator is not specified, but the operator requires one, accumulator 0 is assumed by default. If an accumulator is specifies by the value $>17_8$, the four least significant bits are used.

1.5.8 Comments

The programmer may add notes to a statement following a semicolon. Such comments do not affect assembly processing or program execution, but are useful in the program listing for later analysis or debugging. The use of angle brackets (<>) should be avoided in comments because they may affect the assembly.

Each line of a program may contain a comment which explains the purpose of the line and any special action it causes. A line may also consist of only a comment; this is usually done at the beginning of each routine or major program section to explain the major flow of control, entry and exit points and any other pertinent information.

1.6 STATEMENT PROCESSING

The assembler has several symbol tables and corresponding search routines. The symbol tables arranged in the order in which they are searched are:

1. Macro Table - This symbol table contains macros, user-defined operator definitions (op-defs) and synonyms (refer to the description of the SYN pseudo-op, paragraph 2.8.3). The macro table is initially empty; it grows as the user defines items.

2. Op-Code Table - This symbol table contains all of the operator-codes (op-codes), the UUO calls and the assembler pseudo-operators (pseudo-ops). Lists of the foregoing items are given in Appendices A and B. The op-code table is generated by the assembler and is of fixed length; it cannot be changed except by reassembling MACRO.

3. User Symbol Table - This symbol table contains all user-defined symbols other than those which are placed in the Macro Table. This table is initially empty; it grows as the user defines items.

4. Mnemonic Table - This table contains the mnemonics for the CALLI, MTAPE and TTCALL UUO's. The mnemonic table is searched only if all other measures fail. Any symbol found in this table is put into the macro table as an op-def as though the user had defined it. Examples of the mnemonics contains by this table are

   a) RESET as defined by the CALLI Ø,Ø
   b) EXIT as defined in CALLI Ø,12
   c) OUTSRT as defined in TTCALL 3,Ø

Internally, the macro table and the user symbol table occupy the same space; however, the entries of each table are easily distinguishable so no confusion takes place.

1.6.1 Order of Statement Evaluation

The following table shows the order in which the assembler searches each statement field:

Label Field	Operator Field	Operand Field
1. Symbol suffixed by colon. If colon is not found, no label is present.	1. Number	1. Number
	2. Macro/OPDEF	2. Symbol
	3. Machine operator	3. Macro/OPDEF
	4. Assembler operator	4. Machine operator
	5. Symbol	5. Assembler operator
	6. CALL1 mnemonic	

A single symbol could be used as a label, an operator, or an operand, depending on where it is used.

The assembler first checks the operator field for a number, and if found, assumes that no operator is present. Likewise, if a symbol is not a macro, OPDEF, machine operator or assembler operator, the assembler will search the symbol table. If a defined symbol is found, no operator is present.

If a defined operator appears in an operand field, it must generate at least one word of data. Statements that do not generate data may not be used as part of operand expressions. If a statement used in an operand expressions generates more than one word of data, only the first word generated is meaningful.

1.6.2 Order of Expression Evaluation

Expressions are evaluated in the following order:

- (Unary operator)
↑D, ↑O, ↑B, ↑F, ↑L
B Shift, ← Shift
Logical operators
Multiply/Divide
Add/Subtract

At each level, operations are performed left to right.

1.7 USER-DEFINED SYMBOLS

User-defined symbols are of two types: labels and assignments. Labels are generated by entering a symbol followed immediately by a colon (e.g., TAG:). Symbols used as labels cannot be redefined with a different value once they have been defined. The value of a label is the value of the location counter at the time that the label is defined.

Assignments are used to represent, symbolically, numbers or bit patterns. Assignments ease the coding task in that only one line has to be changed (that containing the assignment) in order to change a number or bit pattern which is used throughout the program. Assignment statements may be changed at any time, the current value of an assignment is the last value given to the symbol used.

1.7.1 Direct Assignment Statements

The macro inserts new symbols with their assigned values directly into the symbol table by using a direct assignment statement of the form,

```
symbol=value)
```

where the value may be a number or expression. Note that the equal sign must immediately follow the symbol. For example,

```
ALPHA= 5)
BETA= 17)
```

A direct assignment statement may also be used to give a new symbol the same value as a previously defined symbol:

```
BETA= 17)
GAMMA= BETA)
```

The new symbol, GAMMA, is entered into the symbol table with the value 17.

The value assigned to a symbol may be changed:

```
ALPHA= 7)
```

changes the value assigned in the first example from 5 to 7.

Direct assignment statements do not generate instructions or data in the object program. These statements are used to assign values so that symbols can be conveniently used in other statements.

1.7.2 Local and Global Symbols

User-defined symbols may be used as local and global symbols in addition to beging used as label and assignment symbols.

Local symbols are symbols which are known only to the program in which they are defined. Two separately assembled macro programs may contain local symbols which have the same mnemonic but different definitions; these programs, however, may be loaded and executed without conflict since the symbols are defined as local to each program.

Global symbols are symbols which can be recognized by programs other than the one in which it is defined. The manner in which a global symbol is written or defined depends on where it is located: in the program in which it is defined or the program in which it is a reference to a symbol defined elsewhere.

Global symbols located in the program in which they are defined must be declared as available to other programs by the use of the pseudo-ops INTERN or ENTRY (see paragraphs 2.5.14.1 and 2.5.14.3) or by the use of the delimiter =: in their definition statement. For example, the symbol FLAG may be declared a global symbols by:

a. INTERN FLAG (the symbol FLAG is declared internal),
b. ENTRY FLAG (identifies the entry point of a library subroutine),
c. FLAG=: 2ØØ (FLAG is given the value 200 and is declared internal).

NOTE

The statement in item c of the foregoing examples (i.e., FLAG=: 2ØØ) is equivalent to the series

```
INTERN FLAG
FLAG= 2ØØ
```

Global symbols located in a program in which they are references to symbols defined in other programs must be declared as external symbols by the use of the EXTERN pseudo-op (see paragraph 2.5.14.1) or a ## suffix. For example, the statement

```
EXTERN FLAG
```

declares the symbol FLAG as an external reference. The statement

```
MOVE Ø,FLAG##
```

also declares the symbol FLAG as an external reference; this statement is the equivalent of the series:

```
EXTERN FLAG
MOVE Ø,FLAG
```

1.7.3 Deleted Symbols

Sometimes a programmer may want to define a symbol in MACRO but not have that symbol typed out by DDT (refer to the *DDT Programmer's Reference Manual*). In such a case, the programmer should define that symbol with a double equal sign:

```
FLAG== 2ØØ)
```

FLAG will be assigned the value 200 and will be

a. Fully available in MACRO.

b. Available for type-in with DDT (assuming that symbols were loaded for the program containing FLAG).

c. Unavailable for type-out by DDT.

This is equivalent to defining FLAG by:

```
FLAG= 2ØØ)
```

and then typing

```
FLAG$K        (the symbol $ represents ALT MODE)
```

to DDT.

A symbol may be defined with == and declared internal in the following manner

```
FLAG==:2ØØ)
```

is equivalent to

```
INTERN  FLAG)
FLAG==2ØØ)
```

The programmer may also want to define a label in MACRO but have the output of the label suppressed in DDT. The following constructions may be used:

LABEL:!	LABEL is a suppressed local symbol.
LABEL::!	LABEL is a suppressed internal symbol.

1.8 NUMBERS

Numbers used in source program statements may be signed or unsigned, and are interpreted by the assembler according to the radix specified by the programmer, where

$$2 \le radix \le 10$$

The programmer may use an assembler pseudo-op, RADIX, to set the radix for the numbers which follow. If the programmer does not use a RADIX statement, the assembler assumes a radix of 8 (octal) except in the case of the POINT pseudo-op (see paragraph 2.5.2).

The radix may be changed for a single numeric term, by using the qualifier followed by a letter, D (for decimal), O (for octal), B (for binary), or F (for fixed-point decimal fractions). Note that these are not control characters. Thus,

↑D1Ø	is stored as	1Ø1Ø
↑O1Ø	is stored as	1ØØØ
↑B1Ø	is stored as	ØØ1Ø

The qualifier ↑L is used for bit position determination of a numeric value. ↑Ln generates an octal value equal to the number of 0 bits to the left of the leftmost 1, if the numeric value n were stored in a computer word.

Expression	Resultant Value	
		44_8 zero bits
↑LØ	44	0000000000. . . .0000000000

Expression	Resultant Value	41_8 zero bits
↑L5	41	0000000000. . . .0000000101
↑L-1	0	1111111111. . . .1111111111

The suffixes K, M and G may be added to numbers as a shorthand method of specifying the number of zeros which are to follow the given number. The meaning of each suffix is:

a) K, add three zeros (e.g., 5K = 5000),
b) M, add six zeros (e.g., 5M = 5000000),
c) G, add nine zeros (e.g., 5G = 5000000000).

1.8.1 Arithmetic and Logical Operations

Numbers and defined symbols may be combined using arithmetic and logical operators. The following arithmetic and logical operators may be used.

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Integer Divide
&	AND
!	Inclusive OR

The assembler computes the 36-bit value of a series of numbers and defined symbols connected by arithmetic and logical operators, truncating from the left, if necessary. The following examples show how these arithmetic and logical operators are written in statements.

```
B=      65+X11-3)
MULI    AC1+7,RHO/31)
MOVE    A+3,BETA-5)
```

Combinations of numbers and defined symbols using arithmetic and logical operators are called expressions.

1.8.2 Evaluating Expressions

When combining elements of an expression, the assembler first performs unary operations (leading + or -), then binary shifts. The logical operations are then done from left to right, followed by multiplications

and divisions, from left to right. Division always truncates the fractional part. Finally, additions and subtractions are performed, left to right. All arithmetic operations are performed modulo 2^{35}.

For example, in the statement:

```
TAG:  TRO 3,1+A&C)
```

the first operand field is evaluated first; the comma terminating this operand indicates that this is an accumulator. In the second operand field, the logical AND is performed first, the result is added to one, and the sum is placed in the memory address field of the machine instruction.

To change the normal order of operations, angle brackets may be used to delimit expressions and indicate the order of computation. Angle brackets must always be used in pairs.

Expressions may be nested to any level, with each expression enclosed in a pair of angle brackets. The innermost expression is evaluated first, the outermost is evaluated last. The following are legal expressions:

```
A+B/5
<<C-D+B-29>*<A-41>>+1
A=<B=<C=10>>
```

1.8.3 Numeric Terms

A numeric term may be a digit, a string of digits, or an expression enclosed in angle brackets. The assembler reduces numeric terms to a single 36-bit value. This is useful when specifying operations such as local radix changes and binary shifts, which require single values.

For example, the ↑D operator changes the local radix to decimal for the numeric term that follows it. The number 23_{10} may be represented by

```
↑D23
↑D<5*2+13>
↑D<TEN*2+THREE>
```

but 23_{10} may not be written,

```
↑D1ØØ-77
```

because the ↑D operator affects only the numeric term which follows it, and in this example the second term (77) is taken under the prevailing radix, which is normally octal.

The B shift operator is preceded by a numeric term (the number to be shifted) andis followed by another term (the bit position of the assumed point). The following are legal:

```
↑F167B17
↑B10011B8
 566B5
<MARK + SIGN>B<PT-XXV>
```

A bracketed numeric term may be preceded by a + or a - sign.

1.8.4 Binary Shifting

A number may be logically shifted left or right by following it with the letter B, followed by a numeric term, n, representing the bit position in which the right-hand bit of the number should be placed. The numeric term, n, may be any (decimal) bit position, starting with zero and numbering from left to right. If n is not used, B35 is assumed; n is taken as modulo 256 decimal. Thus, the number ↑D10 is stored as 000000 000012; but ↑D10B32 is shifted left three binary positions and stored as 000000 000120; and D10B4 is shifted left 31 positions, so that its rightmost bit is in bit 4 and stored as 240000 000000.

Binary shifting is a logical operation, rather than an arithmetic one.

The following are legal binary shifts:

```
1B0     400000 000000
1B17    000001 000000
1B35    000000 000001
-1B35   777777 777777  (see explanation below)
-1B53   000000 777777
-1B70   000000 000001
```

Note that the following expressions are equivalent:

```
10B32  ↑010B32 ≡ 10B <42-10>≡ 10B<  ↑D  <42-10>>≡ 10B<↑ D42-  ↑D10>
```

The unary operators preceding a value are interpreted first by the assembler before the binary shift. A leading plus sign has no effect, but a leading minus sign causes the assembler to shift and then to store the 2's complement.

Binary shifting may operate on numeric terms, as defined in Section 1.3.2.

1.8.5 Left Arrow Shifting

If two expressions are combined with the operator "←", i.e., <m>←<n>, the 36-bit value of expression m is shifted V bits (where V is the value of expression n) in the direction of the arrow (left) if V is positive or against the arrow if V is negative. The effective magnitude of V is that of the address of an LSH instruction.

1.8.6 Floating-Point Decimal Numbers

If a string of digits contains a decimal point, it is evaluated as a floating point decimal number, and the digits are taken radix 10. For example, the statement,

```
17.Ø     is stored as 205420 000000.
```

Floating-point decimal numbers may also be written, as in FORTRAN, with the number followed by the letter E, followed by a signed exponent representing a power of 10. The following examples are valid:

```
NUM1:  17.2E-4)
NUM2:  3.85E2)
NUM3:  -567.825E33)
```

1.8.7 Fixed-Point Decimal Numbers

As shown in Section 1.8, ↑D followed by a numeric term, is used to enter decimal integers.

Fixed-point decimal numbers (mixed numbers) are preceded by ↑F followed by a number (not a numeric term, defined below) which normally contains a decimal point. The assembler forms these fixed-point numbers in two 36-bit registers, the integer part in the first and the fractional part in the second. The value is then stored in one storage word in the object program, the integer part to the left of the assumed binary point, the fractional part to the right.

The binary shift (B) operator is used to position the assumed point. The number ↑F123.45B8 is formed in two registers:

```
000000 000173     (the integer part)
346314 631462     (the fraction part, left-justified)
```

The B operator sets the assumed point after bit 8, so the integer part is placed in bits 0-8, and the fraction part in bits 9-35 of the storage word. In this case, the integer part is truncated from the left to fit the 9-bit integer field. The fraction part is moved into the 27-bit field following the assumed point and is truncated on the right. The result is,

```
173346 314631
   ↑
(assumed point)
```

If a B shift operator does not appear in a fixed-point number, the point is assumed to follow bit 35, and the fractional part is lost.

Fixed-point numbers are assumed to be positive unless a minus sign precedes the qualifier:

```
000000 000173     ↑F123.45
000173 346314     ↑F123.45B17
346314 631462     ↑F123.45B-1

777777 777604     -↑F123.45
777604 431463     -↑F123.45B17
431463 146316     -↑F123.45B-1
```

Negative fixed-point numbers, such as the example above, are assembled as if they were positive numbers, complemented, and then logically shifted.

1.9 ADDRESS ASSIGNMENTS

As source statements are processed, the assembler assigns consecutive memory addresses to the instruction and data words of the object program. This is done by incrementing the location counter each time a memory location is assigned. A statement which generates a single object program storage word increments the location counter by one. Another statement may generate six storage words, incrementing the location counter by six.

The mnemonic instruction and monitor command[1] statements generate a single storage word. However, direct assignment statements and some assembler pseudo-ops do not generate storage words, and do not affect the location

[1]The terms monitor command (as used here) and programmed operator are synonymous.

counter. Other pseudo-ops and macros may generate many words in the object program.

1.9.1 Setting and Referencing the Location Counter

The MACRO-10 programmer may set the location counter by using the pseudo-ops, LOC and RELOC, which are described in Chapter 2. He may reference the location counter directly by using the symbol, point (.). For example, he can transfer to the second previously assigned storage word by writing:

```
JRST .-2)
```

1.9.2 Indirect Addressing

The character @ prefixing an operand causes the assembler to set bit 13 in the instruction word, indicating an indirect address. For an explanation of indirect addressing and effective address calculation, see the *PDP-10 System Reference Manual.*

1.9.3 Indexing

If indexing is used to increment the address field, the address of the index register used is entered in parentheses, as the last part of the memory reference operand. This is normally a symbolic name defined by a direct assignment statement, or an octal number in the range 1-17, specifying 1 of the 15 index registers. However, the address of the index register may be any legal expression or an expression element.

This is a symbolic, indirect, indexed, memory reference:

```
A: ADD 4,@NUM(17))
```

NOTE

The parentheses cause the value of the enclosed expression to be interpreted as a 36-bit word with its two halves interchanged, e.g., (17) is effectively 000017000000_8. The 36-bit value is added to the instruction and may modify it. This is often used to generate right half values from left half expressions; for example, the statement

```
TLO AC,(1BØ)
```

which sets the sign bit.

1.10 LITERALS

In a MACRO statement, a symbolic data reference may be replaced by a direct representation of the data enclosed in square brackets ([]). This direct representation is called a literal. The assembler stores data found within brackets in a Literal table, assigns an address to the first word of the data and inserts that address in the machine instruction.

A literal may consist of more than one statement and may generate more than one word of data. A literal must, however, generate at least one word but no more than 18 words. Literals which consist of only pseudo-ops (such as RADIX) which do not generate data or direct assignments are illegal.

Literals may be nested (i.e., bracketed data within other sets of bracketed data) up to 18 levels.

The following is a simple example of the user of literals. Byte instructions must reference by a byte pointer in this manner:

```
    LDB     4,BP)
BP: POINT   1Ø,A+3,14)
```

(POINT is a pseudo-op which sets up a byte pointter word.) A literal can be used to insert the POINT statement directly. For example

```
LDB 4,[POINT 1Ø,A+3,14])
```

Literals are often used as constants as, for example:

a) PUSH 17,[Ø]   (note that Ø generates one word of zero).
b) MOVE L. [3,14]

The following is an example of a multi-line literal:

```
GETCHR: SOSG  IBUF+2                      ;ANY CHARS LEFT?
        PUSHJ P,[IN    N,                 ;NO, READ SOME IN
                 POPJ  P,                 ;NO UNUSUAL CONDITIONS
                 STATZ N,74ØØØØ           ;CHECK FOR ERRORS
                 JRST  [MOVEI E, [SIXBIT /INPUT ERROR/]
                        JRST ERRPNT] ;PUBLISH ERROR MESSAGE
                 JRST  ENDFIL]            ;END OF FILE HANDLER
        ILDB  AC,IBUF+1                   ;PICKUP NEXT CHAR
        POPJ  P,
```

NOTE

The closing right square bracket does not terminate the literal if placed after the semicolon.

The excessive use of literals, especially for small subroutines, is not recommended since they use up assembler space at the rate of four locations per data word generated. Literals also make debugging more difficult and may cause page faults in the KI-10 processor virtual memory allocation.

The PDP-6 version of macro (MACRO-6) only permitted literals to contain one statement but it permitted the right bracket to be dropped. Dropping the right bracket is not permitted by MACRO-10.

Two pseudo-ops MLON and MLOFF provide compatibility with old programs. Use of these pseudo-ops is required since

```
MOVE AC,[SIXBIT/TEXT/)
```

is legal in MACRO-6, even though the closing right bracket (]) of the literal has been omitted. In normal mode, MACRO does not allow such an unterminated literal. The pseudo-op

```
MLON
```

is set at the start of each assembly to cause the assembler to consider all code following a left bracket as part of a literal, until such time as the assembler processes a matching right bracket. Thus, carriage-return, line-feed does not end a literal, but rather the programmer must insert a right bracket. The pseudo-op,

```
MLOFF
```

set by the switch /O, places MACRO into the compatibility mode in which literals may occupy only a single line.

The symbol . (current location) is not changed by the use of literals. It retains the value it had before the literal was entered.

Chapter 2
MACRO-10 Assembler Statements—Pseudo-Ops

Assembler statements or pseudo-ops direct the assembler to perform certain assembler processing operations, such as converting data to binary under a selected radix, or listing selected parts of the assembled object program. In this chapter, these assembler processing operations are fully described.

NOTE

> The pseudo-op name must follow the rules for constructing a symbol (refer to Paragraph 1.5.1) and must be terminated by a character other than those listed in Paragraph 1.5.1 as valid symbolic characters. (Normally, a space or tab is used as a terminator.)

2.1 ADDRESS MODE: RELOCATABLE OR ABSOLUTE

MACRO-10 normally assembles programs with relocatable binary addresses, so that the program can be located anywhere in memory for execution as a function of what has been previously loaded. When desired, the assembler will also assign absolute location addresses, either for the entire program or for selected parts. Two pseudo-ops control the address mode: RELOC and LOC.

RELOC N↵

This statement sets the location counter to n, which may be a number or an expression, and causes the assembler to assign relocatable addresses to the instructions and data which follow. Since most relocatable programs start with the location counter set to 0; the implicit statement,

RELOC Ø↵

begins all programs, and need not be written by the programmer who wants his program assembled with relocatable addresses.

LOC N↵

This statement sets the location counter to n, a number or an expression, and causes the assembler to assign absolute addresses, begin ing with n, to the instructions and data which follow. If the entire program is to be assigned absolute locations, a LOC statement must precede all instructions and data.

If n is not specified

(LOC↵)

zero is assumed initially.

If only a part of the program is to be assembled in absolute locations, the LOC statement is inserted at the point where the assembler begins assigning absolute locations. For example, the statement,

LOC 2ØØ↵

causes the assembler to begin assigning absolute addresses, and the next machine instruction or data word is stored at location 200_8.

To change the address mode back to relocatable, an explicit RELOC statement is required. If the programmer wants the assembler to continue assigning relocatable addresses sequentially, he writes,

RELOC ⤶

To switch back to the next sequential absolute assignment, he writes,

LOC ⤶

Thus, the programmer is not required to insert a location counter value in either a LOC or RELOC statement, and unless he does, both the relocatable coding and the absolute coding will be assigned sequential addresses. This is shown in the following skeleton coding. The single quote mark is used here, and in MACRO-10 listings, to identify relocatable addresses.

```
Location Counter    Program
   000000'         ADD 1,X        ;RELOC 0 IS IMPLICIT.
                   .
                   .
                   .
   000074'         LOC 1000       ;CHANGES TO ABSOLUTE, STARTING
   001000          SUB 5,TOT      ;WITH 001000.
                   .
                   .
                   .
   001034          RELOC          ;SETS LOCATION COUNTER TO 74.
   000074'         ADD 2,XAT
   000075'         LOC            ;SWITCHES LOCATION COUNTER
   001034          EXP A-X+7      ;BACK TO ABSOLUTE SEQUENCE.
```

When operating in the relocatable mode, the assembler determines whether each location in the object program is relocatable or absolute, using an algorithm described in Chapter 5.

2.1.1 Relocation Before Execution - PHASE and DEPHASE Statements

Part of a program can be moved into other locations for execution. This feature is often used to relocate a frequently used subroutine, or iterative loop, into fast memory (accumulators $0\text{-}17_8$) just prior to execution.

To use this feature, the subroutine is assembled at sequential relocatable or absolute addresses along with the rest of the program, but the first statement before the subroutine contains the assembler operator, PHASE, followed by the address of the first location of the block into which the subroutine is to be moved prior to execution. All address assignments in the subroutine are in relation to the argument of the PHASE statement. The subroutine is terminated by a DEPHASE statement, which requires no operands, and which restores the location counter.

In the following example, which is the central loop in a matrix inversion, a block transfer instruction moves the subroutine LOOP into accumulators 11-16.

```
                                    MOVE [XWD LOOPX,LOOP]
Relocatable                         BLT LOOP+4
  Address                           JRST LOOP
                  LOOPX:            PHASE 11
                  LOOP:             MOVN A (X)
                                    FMP MPYR
Absolute                            FADM A (Y)
Address                             SOJGE X, .-3
                                    JRST MAIN
                                    DEPHASE
```

The label LOOP represents accumulator 11, and the point in the SOJGE instruction represents accumulator 14.

Note that the code inside the phase to dephase program segment is loaded into the address following the previous relocatable code; all labels inside the segment, however, have the address corresponding to the phase address. Thus the phased code cannot, in general, be executed until it has been moved to the address for which it was assembled.

2.2 NAMING PROGRAMS

Normally the first statement in a program gives the name of the program using the TITLE pseudo-op. This pseudo-op has the form

TITLE NAME↵

in which the single operand (i.e., NAME) may contain up to 60 characters.

The name given will be printed at the top of each page of the program listing. The first 6 characters of the given title will appear in the assembled program as the program name. If no title is given, the assembler inserts the name .MAIN. The program name given in the TITLE statement is used when debugging with DDT in order to gain access to the program's symbol table.

Only one TITLE pseudo-op is permitted in a program; it can appear anywhere in the program but is normally the first line on the first page. Remember that a name may be longer than 6 characters, however, only the first 6 symbol combinations (within the radix-50 set) will be used for the program name.

2.2.1 Program Subtitles

After the first page of a program listing, the first data line encountered on a page may be a subtitle. Subtitles are generated using the pseudo-op SUBTTL. This pseudo-op has the form

```
        SUBTTL SUBTITLE)
```

in which the single operand (SUBTITLE) may contain up to 40 characters. A subtitle is printed as the first data line on a page and all succeeding pages until the end of the listing or until the subtitle is changed. If the current subtitle is changed by another SUBTTL statement which is the first data line on a page, the new subtitle appears on the new page and all subsequent pages. If the SUBTTL statement is not the first statement on a page, the new subtitle appears on the next page and all subsequent pages.

Subtitles can be changed as often as required; they do not generate data and they do not affect the binary procedure only the listing. They are used for informational purposes only.

2.3 PROGRAM ORIGIN

Initially all programs start with an implicit RELOC Ø which sets the mode to be relocatable and the first address to be Ø. Unless otherwise changed, the code generated will be a single-segment program.

The programmer can change the relocatable nature of the program by using a LOC statement to generate absolute code (normally used for diagnostics) or to generate high-segment code.

High-segment (or two-segment programs) have two logical address spaces; one starting at Ø and increasing, the other starting at 4ØØØØØ (128K) and increasing. Two pseudo-ops, HISEG and TWOSEG control High or two-segment program operation.

2.3.1 HISEG Statements - The HISEG Pseudo-Op Statement

This pseudo-op does not affect the assembly operations in any way except to generate information that directs the Loader to load the current program into the high segment if the program has reentrant (two-segment) capability. (Refer to Block Type 3 Load Into The High Segment, paragraph 6.2.1.1, for additional information.) This pseudo-op should appear at the beginning of the source program.

NOTE

> Whenever possible the pseudo-op TWOSEG should be used instead of HISEG. This pseudo-op provides functions which are superior to those of HISEG.

HISEG may be followed by an optional argument which represents the program high-segment origin address. This argument, when used, must be equal to or greater than 4ØØØØØ and must be a K-bound (even multiple of 2000) value. The code produced by HISEG will execute at either relocatable Ø or relocatable 4ØØØØØ depending on the loading instructions given.

HISEG must not be used if the programmer wishes to reference data in the low segment since locations in the low segment are referenced by absolute addresses only.

2.3.2 TWOSEG Statements

The TWOSEG pseudo-op generates code that directs MACRO and LOADER to assemble and load a two-segment program in one file. This pseudo-op outputs a block type 3 (refer to Paragraph 6.2.1.1) which signals the LOADER to expect two segments. An optional argument may be present

which is the first address in the high segment. If no argument is present, 400000 is assumed.

The high segment code must be preceded by

    RELOC 4ØØØØØ

or greater; the low segment code by

    RELOC Ø

or an argument indicating the low segment. Each RELOC pseudo-op switches the relocation.

The listing produced by the TWOSEG pseudo-op shows high segment addresses as greater than 400000 or the argument of the pseudo-op, and low segment addresses as less than 400000 or the argument of the pseudo-op. All relocatable addresses are flagged with a single quote.

2.4 ENTERING DATA

2.4.1 RADIX Statements

When the assembler encounters a numerical value in a statement, it converts the number to a binary representation reflecting the radix indicated by the programmer. The statement,

    RADIX N↵

where n is a decimal number, $2 \leq n \leq 10$, sets the radix to n for all numerical values that follow, unless another RADIX statement changes the prevailing radix or a local radix change occurs (see below).

For example, if the programmer wants the assembler to interpret his numbers as decimal quantities, then the prevailing radix must be set to decimal before he uses decimal numbers.

    RADIX 1Ø↵

The statement, RADIX 2, sets the prevailing radix to binary.

The implicit statement, RADIX 8, begins every program; if the programmer wants to enter octal numbers, this statement is not necessary.

2.4.2 Entering Data Under the Prevailing Radix

Data is entered under the prevailing radix by typing the data, followed by a carriage return:

```
765432234567↵
```

Data may be labeled and contain expressions:

```
LAB:   456+A+B/<C+D>↵
```

Data may also be entered by first using a direct assignment statement to place a symbol with an assigned value in the symbol table, and then using the symbol to insert the assigned value in the object program. For example, the direct assignment statements,

```
A=2↵
B=5↵
```

cause two entries in the symbol table. The following statement enters the sum of the assigned values in the object program at symbolic address REX.

```
REX:   A+B↵   REX contains 000000 000007
```

The radix can also be changed locally, that is, for a single statement or a single value, after which the prevailing radix is automatically restored, as described in Section 1.3.

2.4.3 DEC and OCT Statements

To change to a local radix for a single statement, the programmer writes:

```
DEC N,N,N,...N↵
```

where all of the numbers and expressions are to be interpreted as decimal numbers. The numbers or expressions following the operator

are separated by commas, and each will generate a word of storage.

```
OCT N,N,N,...N ↵
```

changes the local radix to octal for this statement only, and generates a word of memory for each number or expression.

The statement,

```
DEC  1Ø,4.5,3.1416,6.Ø3E-26,3 ↵
```

generates five decimal words of data.

2.4.4 Changing the Local Radix for a Single Numeric Term

To change the radix for a single number or expression, the numeric term is prefixed with ↑D, ↑O, ↑B, or ↑F, as explained in Chapter 1. If an expression is used, it must be enclosed in angle brackets,

```
↑D<A+B-C/2ØØ> ↵
```

These prefixes may generate a word, or part of an instruction word. The statement,

```
TOTAL2:MOVE  ↑D1Ø,ABZ ↵
```

causes the contents of ABZ to be moved to accumulator 12_8.

When the assembler encounters a numeric term, it forms the binary representation in a 36-bit register under the prevailing or local radix. If the quantity is a part of an instruction, it is truncated to fit in the required field.

For example, the accumulator field must have a final value in the range $0\text{-}17_8$. In the statement,

```
MOVE  ↑D6Ø,ABZ ↵
```

the assembler first interprets the accumulator address in a 36-bit register, forming the integer 000000000074: but takes only the rightmost four bits and places them in the accumulator field of the instruction, which results in the selection of accumulator 14_8.

2.4.5 RADIX 50 Statement

Another radix changing statement is available, but it is used primarily in systems programming. This is RADIX50 n,sym) which is used by the assembler, PDP-10 Loader, DDT, and other systems programs to pack symbolic expressions into 32 bits and add a 4-bit code field n in bits 0-3. This is explained in Appendix F of this manual. (The mnemonic SQUOZE may be used in place of RADIX50.)

2.4.6 EXP Statement

Several numbers and expressions may be entered by using the EXP statement:

```
EXP  X,4,  ↑D65,HALF,B+362-A
```

which generates one word for each expression; five words were generated for the above example.

2.4.7 Z Statement

A zero word can be entered by using the operator, Z.

```
LABEL:  Z
```

generates a full word of all zeros at LABEL. If operands follow the Z, the assembler forms a primary machine instruction, with the operator field and other unknown fields zeroed. In the statement,

```
Z 3,
```

the assembler finds an accumulator field, but no address field, and generates this machine instruction: 000140 000000.

2.5 INPUT DATA WORD FORMATTING

2.5.1 BYTE Statement

To conserve memory, it is useful to store data in less than full 36-bit words. Bytes of any length, from 1 to 36 bits, may be entered by using a BYTE statement.

```
BYTE  (N) X,X,X
```

The first operand (n) is the byte size in bits. It is a decimal number in the range 1-36, and must be enclosed in parentheses. The operands following are separated by commas, and are the data to be stored. If an operand is an expression, it is evaluated and, if necessary, truncated from the left to the specified byte size. Bytes are packed into words,

starting at bit 0, and the words are assigned sequential storage locations. If, during the packing of a word, a byte is too large to fit into the remaining bits, the unused bits are zeroed and the byte is stored left-justified in the next sequential location.

In the following statement, three 12-bit bytes are entered:

```
LABEL:  BYTE (12)5,177,N ↵
```

This assembles at LABEL as, 0005 0177 0316, where N=316.

The byte size may be altered by inserting a new byte size in parentheses immediately following any operand. Notice that the parentheses serve as delimiters, so commas must not be written when a new byte size is inserted. The following are legal:

```
BYTE  (6)5(14)NT(3)6,2,5 ↵
```

where 6 is entered in a 6-bit byte, NT in the following 14-bit byte, 6 in the following 3-bit byte, followed by 2 and 6 in 3-bit bytes. A BYTE statement can be used to reserve null fields of any byte size. If two consecutive delimiters are found, a null field is generated.

```
BYTE  (18),5 ↵
```

When the assembler finds two delimiters, it assembles a null byte. In this case, 000000 000005. To enter ASCII characters in a byte, the characters are enclosed in quotation marks.

```
BYTE  (7)"A" ↵
```

Text handling pseudo-ops are discussed in paragraph 2.5.5.

2.5.2 POINT Statement - Handling Bytes

Five machine instructions are available for byte manipulation. These instructions reference a byte pointer word, which is generated by the assembler from a POINT statement of the form,

```
LABEL:POINT s, address, b ↵   (s and b are decimal)
```

where the first operand s is a decimal number indicating the byte size, the second operand is the address of the memory location which contains the byte, and the third operand, b, is the bit position in the word of the right-hand bit of the byte (if b is not specified, the bit position is the nonexistent bit to the

left of bit 0). The address specified in the second operand may be indirect and indexed. If the byte size is not specified, MACRO-10 assumes 36 bits.

In the following example, an LDB (load a byte from a memory location into an accumulator) and an ILDB instructions are used, along with three assembler statements. The ILDB instruction "increments" AC to look like AB, then does a load byte; the effect of the two instructions is the same.

```
000000' 050000  000000   AA:     BYTE   (6)5
000001' 360600  000000'  AB:     POINT  6,AA,5
000002' 440600  000000'  AC:     POINT  6,AA

000003' 135140  000001'  START:  LDB    3,AB
000004' 134140  000002'          ILDB   3,AC
```

The first statement enters the quantity 5 in a 6-bit byte at symbolic address AA which is 0. The second statement is for reference by the load byte instruction. When the LDB is executed, the machine goes to AB for the byte size, its address, and bit position. In this case, it finds that the byte size is 6 bits, the byte is located in the word AA, and the right-hand bit of the byte is in bit 5. The byte is then loaded into accumulator 3, where it looks like this: 000000 000005.

The other byte manipulation mnemonic instructions reference the byte pointer word in similar ways. The deposit byte (DPB) instruction takes a byte from an accumulator and deposits it, in the position specified by the pointer word, in a memory word.

The increment byte pointer (IBP) instruction increments the bit position indicator (the third operand in the referenced POINT word) by the byte size. This is useful when loading or depositing a string of bytes, using the same byte pointer word.

The increment and load byte (ILDB) and increment and deposit byte (IDPB) instructions increment the byte pointer word by the byte size before loading or depositing.

2.5.3 IOWD Statement: Formatting I/O Transfer Words

The assembler generates I/O transfer words in a special format for use in BLKI and BLKO and all four pushdown instructions. The general statement is,

```
IOWD  N,M↵
```

where two operands, which may be numbers or expressions, follow the IOWD operator. This statement generates one data word. The left half of the assembled word contains the 2's complement of the first operand n, and the right half-word contains the value of the second operand m, minus one. For example,

```
IOWD 6,↑D256↵
```

assembles as 777772 000377.

2.5.4 XWD Statement: Entering Two Half-Words of Data

The XWD statement enters two half-words in a single storage word. It is written in the form,

```
XWD  LHW,RHW↵
```

where the first operand is a symbol or expression specifying the left half-word, and the second operand specifies the right half-word. Both are formed in 36-bit registers and the low order 18-bits are placed in the half-words. The high-order 18 bits of each operand are ignored. Three examples follow:

```
XWD  A,B↵
XWD  SUM+2,DES+5↵
XWD  START,END↵
```

XWD statements are used to set up pointer words for block transfer instructions. Block transfer pointer words contain two 18-bit addresses: the left half is the starting location of the block to be moved, and the right half is the first location of the destination. A,,B may also be used to duplicate the results of XWD A,B.

2.5.5 Text Input

The assembler translates text written in full 7-bit ASCII or 6-bit compressed ASCII. It will also format 7-bit ASCII with a null character at the end of text, if desired. These codes are listed in Appendix E.

In all three text modes, the printing symbols in the code set are translated to their binary representation.

To translate and store a single word containing as many as five 7-bit ASCII characters, right-justified, the input characters are enclosed in quotation marks.

```
"AXE")   is stored as
         0 0000000 0000000 1000001 1011000 1000101
         0   null    null     A       X       E
```

Notice that characters are right-justified, and bit 0, which is not used, is set to zero.

Up to six 6-bit ASCII characters may be translated and stored, right-justified, in a single word by enclosing the input characters in single quotation marks.

```
'TABLES' is stored as

110100 100001 100010 101100 100101 110011
  T      A      B      L      E      S
```

NOTE

> The quotation marks (single or double) may only be used to assemble a single word. To input strings of text characters, the following three pseudo-ops must be used.

2.5.5.1 ASCII, ASCIZ, and SIXBIT Statement - To enter strings of text characters, the operators ASCII, SIXBIT, and ASCIZ are used. The delimiter for the string of text characters is the first non-blank character following the character that terminates the operator (refer to the note on page 2.1). The binary codes are left-justified. Unused character positions are set to zero (null). Text is terminated by repeating the initial delimiter. If the initial delimiter is a symbol constituent, the pseudo-op must be followed by a space or a tab.

The statement

```
ASCII "AXE" ↵
```

where the quotation marks are the delimiters, assembles as

```
1000001 1011000 1000101 0000000 0000000 0
   A       X       E      null    null  0
```

The operator ASCIZ (ASCII Zero) guarantees a null character at the end of text. If the number of characters is a multiple of five, another all zero word is added. For example,

```
ASCIZ/"AXE"/↵
```

assembles as,

```
0100010 1000001 1011000 1000101 0100010 0
   "       A       X       E       "
```

followed by another word of zeros.

```
0000000 0000000 0000000 0000000 0000000 0
null
```

When thefull 7-bit ASCII code set is not required, the 64-character 6-bit subset may be entered, using the SIXBIT operator. Six characters are left-justified in sequential storage words. Format of the SIXBIT statement is the same as for ASCII statements. To derive SIXBIT code:

a. Convert lower case ASCII characters to upper case characters.

b. Add 40_8 to the value of the character.

c. Truncate the result to the rightmost six bits.

2.5.6 Reserving Storage

The programmer can reserve single locations, or blocks of many locations for use during execution of his program.

2.5.6.1 Reserving a Single Location - The number sign (#), suffixing a symbol in an operand field, is used to reserve a single location. The symbol is defined, entered in the assembler's symbol table,and can be referenced elsewhere in the program without the number sign. For example,

```
LAB:  ADD 3,TEMP#
```

reserves a location called TEMP at the end of the program, which may be used to store a value entered at some other point in the program. This feature is useful for storing scalars, and other quantities which may change during execution.

The pseudo-op INTEGER may be used to reserve storage locations at the end of the program on a one-per-given name basis. For example the statement

```
INTEGER TEMP,FOO,BAR
```

will reserve 3 locations identified as TEMP, FOO and BAR. The assignment of the locations to the names given is performed on an alphabetical basis by the assembler rather than on the order in which the names are given. For example, the order of the locations reserved by the foregoing INTEGER statement would be BAR, FOO then TEMP.

Multiple word locations may be reserved by the ARRAY pseudo-op. For example, the statement

```
ARRAY FOO[2*3]
```

reserves a 2-word by 3-word array in memory which is identified by the name FOO.

NOTE

> If the pseudo-op TWOSEG is used, the variables reserved by an array statement must be assigned to the low segment only; thus, a VAR pseudo-op is required after a RELOC back to the low segment.

2.5.7 VAR Statements

VAR ↲

This statement causes symbols which have been defined by suffixing with the # sign (array and integer pseudo-ops) in previous statements to be assembled as block statements. This has no effect on subsequent symbol definitions of the same type.

If the LIT and VAR statements do not appear in the program, all literals and variables are stored at the end of the program.

2.5.8 BLOCK Statements

To reserve a block of locations, the BLOCK operator is used. It is followed by a single operand, which may be a number or an expression in the current radix, indicating the number of words to be reserved. The assembler increments the location counter by the value of the operand. For example,

MATRIX: BLOCK N*M ↲

reserves a block of N*M words starting at MATRIX for an array whose dimensions are M and N.

BLOCK is used to reserve words in a specific order; remember that data words should be stored in the low segment in two-segment programs.

2.5.9 END Statements

The END statement must be the last statement in every program. A single operand may follow the END operator to specify the address of the first instruction to be executed. Normally this operand is given only in the main program; since subprograms are called from the main program, they need not specify a starting address.

END START↲ start is the label at the starting address

When the assembler first encounters an END statement, it terminates pass 1 and begins pass 2. The END also terminates pass 2, after which

the assembler automatically assembles all previously defined variables and literals starting at the current location.[1]

The following processing operations can be performed at any point in the program.

2.5.10 LIT Statements

LIT ↵

This statement causes literals that have been previously defined to be assembled, starting at the current location. If n literals have been defined, the next free storage location will be at location counter plus n. Literals defined after this statement are not affected.

If a LIT statement does not appear before the END statement, the literals are XLISTed (refer to paragraph 2.6.3). If the output of literals is desired, the LIT pseudo-op should appear immediately before the END statement.

NOTE

> In a two-segment program LIT must be given in the high segment. The END statement must also be given in the high segment or the literals will go to the low segment.

2.5.11 Multi-Program Assembly

The pseudo-op PRGEND is used to compress many small files into one large file to save space and disk lookups. This pseudo-op has the form PRGEND↵ . PRGEND allows multiprogram assemblies, and is used for assembling library files (LIB40) in which all programs are very short. PRGEND takes the place of all but the last END statement. The output is a binary file which can be loaded in search mode. The use of PRGEND costs assembler space since the symbol tables, literal tables and titles of each of the small files (i.e., programs) involved must be saved at the end of pass 1. Also, since PRGEND is functionally an END statement, macros cannot be used over it (i.e., macros cannot generate PRGEND as part of their expansions).

---

[1]The END statement is also used to specify a transfer word in some output file formats. (See Section 6.2.2.4.)

If the LIT and VAR statements do not appear in the programs, all literals and variables are stored at the end of the program.

2.5.12 PASS2 Statements

PASS2)

This statement switches the assembler to pass 2 processing for the remaining coding. Coding preceding this statement will have been processed by pass 1 only. This is used primarily for debugging, such as testing macros defined in the pass 1 portion.

2.5.13 PURGE Statements

The PURGE statement is used to delete defined symbols. Its general form is:

PURGE symbol, symbol, symbol ↲

where each operand is a user-created label, operator, or macro call which is to be deleted from the assembler's tables. The PURGE statement is normally used at the end of programs to conserve storage and to delete symbols for DDT. Purged symbol table space is reused by the assembler.

If the programmer uses the same symbol for both a macro call and/or OPDEF (refer to Section 2.8.2) and for a label, a PURGE statement deletes the macro call or OPDEF. A repeat of the symbol in the PURGE statement also purges the label. For example, the following statement purges both:

PURGE SOLV,SOLV ↲

The first SOLV purges the macro call; the second SOLV purges the label.

2.5.14 XPUNGE Statements

The XPUNGE pseudo-op deletes all local symbols during pass 2; it has the form:

XPUNGE ↲

The use of this pseudo-op reduces the size of the REL file and speeds up loading (especially of DDT). XPUNGE should be placed just prior to the END statement.

2.5.15 Linking Subroutines

Programs usually consist of subroutines which contain references to symbols in external programs. Since these subroutines may be assembled separately, the loader must be able to identify "global" symbols. For a given subroutine, a global symbol is either a symbol defined internally and available for reference by other subroutines, or a symbol used internally but defined in another subroutine. Symbols defined within a subroutine, but available to others, are considered internal. Symbols which are externally defined are considered external.

These linkages between internal and external symbols are set up by declaring global symbols using the operators EXTERN, INTERN, or ENTRY. The double colon (::) may also be used.

2.5.15.1 EXTERN Statements - The EXTERN statement identifies symbols which are defined elsewhere. The statement,

```
EXTERN SQRT, CUBE,TYPE
```

declares three symbols to be external. External symbols must not be defined within the current subroutine. These external references may be used only as an address or in an expression that is to be used as an address. For example, the square root routine declared above might be called by the statement,

```
PUSHJ  P,SQRT
```

External symbols may be used in the same manner as any other relocatable symbol. Examples:

```
                   EXTERN A
200300  000003* MOVE    6,A+3
000003* 000000* XWD     A+3,A
777777  777771  B=      A-7
                OPDEF   U[XWD B+3,A-5]
777774* 777773* U
```

The external symbols are flagged with asterisks. There are three restrictions for the use of external symbols:

a. Externals may not be used in LOC and RELOC statements.

b. The use of more than one external in an expression is not permitted. Thus, A+B (where A and B are both external) is illegal.

c. Globals may only be additive; therefore, the following are illegal

```
-A          EXP-A
2*A         2*A-A
```

An alternative method for generating external symbols is to use a double pound sign (##) following the symbol name. This method eliminates specifying the EXTERN statement. For example,

```
MOV Ø,JOBREL##
```

is equivalent to

```
EXTERN JOBREL
MOVE Ø,JOBREL
```

2.5.15.2 INTERN Statements - To make internal program symbols available to other programs as external symbols, the operators INTERN or ENTRY are used. These statements have no effect on the actual assembly of the program, but will make a list of symbol equivalences available to other programs at load time. The statement,

```
INTERN MATRIX ↵
```

makes the subroutine MATRIX available to other programs. An internal symbol must be defined within the program as a label, variable, or by direct assignment.

2.5.15.3 ENTRY Statements - Some subroutines have common usage, and it is convenient to place them in a library. In order to be called by other programs, these library subroutines must contain the statement,

```
ENTRY NAME ↵
```

where "name" is the symbolic name of the entry point of the library subroutine.

ENTRY is equivalent to INTERN with the following additional feature. All names in a list following ENTRY are defined as internal symbols and are placed in a list at the beginning of the library of subroutines. If the loader is in library search mode, a subroutine will be loaded if the program to be executed contains an undefined global symbol which matches a name on the library ENTRY list.

If the MATRIX subroutine mentioned before is a library subroutine, it must contain the statement,

```
ENTRY MATRIX ↲
```

Since library subroutines are external to programs using them, the calling program must list them in EXTERN statements.

2.6 SUPPRESSION OF SYMBOLS

When a parameter file is used in assemblies, many symbols get defined but are never used. Unused defined symbols take up space in the binary file and complicate listings of the file. Unused and unwanted symbols may be removed from symbol tables by the use of a pseudo-op, either SUPPRESS or ASUPRESS. These pseudo-ops control a suppress bit in each location of the symbol table; if a suppress bit is on, the symbol in that location is not output. The suppress bit is used in the file S.MAC so that if a bit is on and the symbol in that location is not used later, the symbol is not output in the CREF table.

2.6.1 SUPPRESS SYMBOL Statement

The SUPPRESS statement turns on the suppress bit for the specified symbols.

2.6.2 ASUPPRESS Statement

The ASUPPRESS statement turns on the suppress bit for all the symbols in the symbol table.

2.6.3 Listing Control Statements

Program listings are normally printed on a line printer or a terminal depending on the listing file device specified. Listings are printed as the source program statements are processed during pass 2. A sample listing is shown in Chapter 7.

From left to right the standard columns of a listing contain

a) the location counter,
b) the instruction or data in octal form, and
c) the symbolic instruction or data followed by comments.

Relocatable object-code addresses are suffixed by a single quotation mark (') which may occur in either the left or right half.

Data is displayed in one of several modes depending on the statement format. The possible statement formats are:

1) Halfword - two 18-bit bytes
2) Instruction - a 9-bit op-code, 4-bit accumulator code, 1-bit indirect bit, 4-bit index, and an 18-bit address segment
3) Input/Output - 3-bit I/O indicator, 7-bit I/O device specification, 3-bit operand, 1-bit indirect address bit, 4-bit index and an 18-bit address segment
4) Byte pointer - 6-bit byte position, 6-bit byte size, 1 unused bit, 1-bit indirect address bit, 4-bit index and an 18-bit address segment
5) ASCII - 5 Seven-bit bytes
6) SIXBIT - 6 six-bit bytes.

NOTE

Refer to the DECsystem-10 System Reference Manual for a complete description of word formats.

The listing function is suppressed within macro expansion, therefore only the macro call and any succeeding lines that generate code are

listed. Line printer listings always begin at the top of a page and up to 55 lines are printed on each page. Consecutive page numbers are printed in the upper right-hand corner of each page. Each page also contains a title and a subtitle.

The standard listing operations can be augmented and modified by using the following listing control statements.

STATEMENT	DESCRIPTION
PAGE ↵	This statement causes the assembler to skip to the top of the next page. (A form feed character in the input text has the same effect and is preferred.
XLIST ↵	This statement causes the assembler to stop listing the assembled program. The listing printout actually starts at the beginning of pass 2 operations. Therefore, to suppress all program listing, XLIST must be the first statement in the program. If only a part of the program listing is to be suppressed, XLIST is inserted at any point to stop listing from that point. Literals are XLISTed if no LIT statement is seen before the END statement.
LIST ↵	Normally used following an XLIST statement to resume listing at a particular point in the program. The LIST function is implicitly contained in the END statement.
LALL ↵	This statement causes the assembler to list everything that is processed including all text, macro expansions and list control codes suppressed in the standard listing.
XALL ↵	Normally used following a LALL statement to resume standard listing.
SALL ↵	This causes suppression of all macro and repeat expansions and their text; only the input file and the binary generated will be listed. SALL can be nullified by either XALL or LALL and the /M switch can be used instead of SALL.
NOSYM ↵	The assembler normally prints out the symbol table at the end of the program, but the NOSYM statement suppresses the symbol table printout.

STATEMENT	DESCRIPTION

TAPE ↲

This pseudo-op causes the assembler to begin assembling the program contained in the next source file in the MACRO command string. For example,

```
.R MACRO
*DSK:BINAME,LPT:←TTY:,DSK:MORE
PARAM=6
TAPE
;THIS COMMENT WILL BE IGNORED
↑Z
```

would set the symbol PARAM equal to 6 and then assemble the remainder of the program from the source file DSK:MORE. Since MACRO is a 2-pass assembler, the TTY: file would probably be repeated for pass 2.

```
END OF PASS 1
PARAM=6
TAPE
↑Z
```

Note that all text after the TAPE pseudo-op is ignored.

PRINTX MESSAGE↲

This statement, when encountered, causes the single operand following the PRINTX operator to be typed out on the TTY. This statement is frequently used to print out conditional information. PRINTX statements are also used in very long assemblies to report the progress of the assembler through pass 1.

REMARK COMMENTS↲

On pass 1 the message is printed on both the list device and TTY. On pass 2 it is printed on the TTY, but only if it is not the list device.

The REMARK operator is used for statements which contain only comments. Such statements may also be started with a semi-colon.

COMMENT↲

This pseudo-op treats the text between the first non-blank character (delimiter) and the next occurrence of the <u>same</u> character as a comment. If the first occurrence of the delimiter is a right (left) angle bracket, the next occurrence of the delimiter must also be a right (left) angle bracket. The text may include the carriage return, line feed sequence. For example,

```
COMMENT/THIS IS A COMMENT
THAT IS MORE THAN ONE LINE LONG
/
```

Internally, the pseudo-op functions as ASCII, but no binary is produced.

2.7 CONDITIONAL ASSEMBLY

Parts of a program may be assembled, or not assembled, on an optional basis depending on conditions defined by an assembler IF statement. The general form is,

```
IF N, <............>
```

where the coding within angle brackets is assembled only if the first operand, N, meets the statement requirement.

The IF statement operators and their conditions are listed below:

```
     Operator                  Assemble angle-bracketed coding IF:

IFE N, <...>                   N=0, or blank
IFG N, <...>                   N>0
IFGE N, <...>                  N=0, or N>0
IFL N, <...>                   N<0
IFLE N, <...>                  N=0, or N<0
IFN N, <...>                   N≠0
IF1, <...>                     encountered during pass 1
IF2, <...>                     encountered during pass 2
```

In the following conditional statements, assembly depends on whether or not a symbol has been defined. The coding enclosed in angle brackets is assembled if,

```
IFDEF SYMBOL, <...>            this symbol is defined
IFNDEF SYMBOL, <...>           this symbol is not defined
```

NOTE

SYMBOL can be an op-code or pseudo-op as well as a user symbol.

The following conditional statements operate on character strings. Arguments are interpreted as 7-bit ASCII character strings, and the assembler makes a logical comparison, character-by-character to determine if the condition is met.

The coding within the third set of angle brackets is assembled if the character strings enclosed by the first two sets of angle brackets:

```
IFIDN <A-Z>  <A-Z>,<...>       (1) are identical
IFDIF <A-Z>  <A-Z>,<...>       (2) are different
```

These statements, IFIDN and IFDIF, are usually used in macro expansions (see Chapter 3) where one or both arguments are dummy variables.

An alternate form is to use delimiters as in ASCII. For example:

```
IFDIF/A-Z/"A-Z',<--->
```

This allows the use of > inside the character string. If the first non-blank (space or tab) character is a < character, then the < > method is used; otherwise, the character is used as a delimiter.

The last pair of conditional statements is followed by a single bracketed character string, which is either blank or not blank, and which is followed by conditional coding in brackets.

The coding enclosed in the second set of angle brackets is assembled if,

```
IFB <...>,<....>          the first operand is blank
IFNB <...>,<.....>        the first operand is not blank
```

A blank field is either an empty field or a field containing only the ASCII characters space (40_8) or tab (11_8).

Again, delimiters can be used as in

```
IFB / .... / , <.....>
```

2.8 ASSEMBLER CONTROL STATEMENTS

2.8.1 REPEAT Statements

The statement

```
REPEAT N,  <...>
```

causes the assembler to repeat the coding enclosed in angle brackets n times. If more than one instruction or data word is to be repeated, each is delimited by a carriage return. For example,

```
ADDX:    REPEAT 3,  <ADD 6,X(4)
                     ADDI 4,1>
```

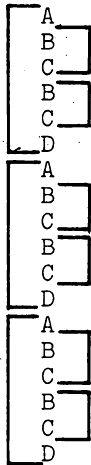
assembles as,

```
ADDX:  ADD 6,(4)
       ADDI 4,1
       ADD 6,X(4)
       ADDI 4,1
       ADD 6,X(4)
       ADDI 4,1
```

Notice that the label of a REPEAT statement is placed on the first line of the assembled coding. REPEAT statements may be nested to any level. The following simplified example shows how a nested REPEAT statement is interpreted.

```
REPEAT 3,<A
REPEAT 2,<B
         C>
         D>
```

assembles as,



NOTE

Brackets indicate repetition.

2.8.2 OPDEF Statements

The programmer can define his own operators using an OPDEF statement, which is written in the form:

```
OPDEF SYM [STATEMENT]
```

where the first operand is defined as an operator, whose function is defined by the second operand, which is enclosed in square brackets. The second operand is evaluated as a statement, and the

result is stored in a 36-bit word. For example,

```
OPDEF CAL1 [Ø3ØØØØ ØØØØØØ]
```

defines CAL1 as an operator, with the value 030000 000000. CAL1 may now be used as a statement operator.

```
Ø3Ø14Ø ØØ1234  CAL1 3,1234
```

which is equivalent to,

```
Ø3Ø14Ø ØØ1234  Z  3,1234(3ØØØØ)
```

When MACRO-10 encounters a user-defined operator, it assembles a single object-program storage word in the format of a primary instruction word (see Chapter 1). The defined 36-bit value is modified by accumulator, indirect, memory address and index fields as specified by the user-defined operator.

For example,

```
OPDEF CAL [MOVE 1,@SYM(2)]↩
CAL 1,BOL(2)↩
```

The CAL statement is equivalent to:

```
MOVE 2,@SYM+BOL(4)↩
```

In this modification the accumulator fields are added, the indirect bits are logically ORed, the memory address fields are added, and the index register addresses are added.

2.8.3 SYN Statements

The statement

```
SYN symbol, symbol
```

defines the second operand as synonymous with the first operand, which must have been previously defined. Either operand may be a symbol or a macro name. If the first operand is a symbol, the second is defined as a symbol with the same value. If the first is

a macro name, the second becomes a macro name which operates identically. If the first is a machine, assembler, or user-defined operator, the second will be interpreted in the same manner. If the first operand in a SYN statement has been previously defined as both a label and as an operator, the second operand is synonymous with the label.

The following are legal SYN statements:

```
SYN K,X↵            ;IF K=5, X=5
SYN FAD,ADD↵
SYN END,XEND↵
```

2.8.4 Extended Instruction Statements

For programming convenience, some extended operation codes are provided in the MACRO-10 Assembler. Primarily, these are used to replace those DECsystem-10 instructions where the combination of instruction mnemonic and accumulator field is used to denote a single instruction. For example:

```
JRST 4
```

is equivalent to a halt instruction. In addition, they are used to replace certain commonly used I/O instruction-device number combinations.

The extended instruction statements are exactly like the primary instruction statements or I/O instruction statements, except that they may not have an accumulator field or device field.

The operator field must have one of the following extended mnemonics:

Extended Instructions	Equivalent Machine Instructions	Meaning
JEN	JRST 12,	Jump and enable the PI (priority interrupt) system
HALT	JRST 4,	Halt
JRSTF	JRST 2,	Jump and restore flags
JOV	JFCL 1Ø,	Jump on overflow and clear
JCRYØ	JFCL 4,	Jump on CRYØ and clear
JCRY1	JFCL 2.	Jump on CRY1 and clear
JCRY	JFCL 6,	Jump on CRYØ or CRY1 and clear
JFOV	JFCL 1,	Jump on floating overflow
RSW	DATAI Ø	Read the console switches

2.9 MULTI-FILE ASSEMBLY

2.9.1 UNIVERSAL Name

UNIVERSAL files may be used to generate data, however, they are normally used to generate symbols, macros and opdef's (user-defined operators). The symbols generated by UNIVERSAL files need not be declared as INTERNAL symbols since all local symbols in files of this type are made available to all programs permitted access to the file.

UNIVERSAL files used to generate data can save time by being set up for a one-pass operation since symbol definition needs to be assembled on one pass only. This one-pass operation can be accomplished in either of two ways:

```
1)  UNIVERSAL NAME
    PASS 2
    .
    .
    .
    END

2)  UNIVERSAL NAME
    IF 2, <END>
    .
    .
    .
    END
```

The first generates a listing; the second does not.

If the UNIVERSAL pseudo-op is seen in a program, the NAME is stored in a table and a flag is set. When the END statement is seen, the symbol table is moved to just after the pushdown stacks and buffers; therefore, the pushdown stacks and buffers cannot be increased during assembly. The first assembly should use the maximum of I/O devices to be used later. The free core pointer is moved to after the top of the moved symbol table, and pointers are stored to enable the table to be scanned.

When assembling is done from indirect files, the universal files must be recompiled by the /COMPIL switch. Otherwise if a REL file later than the source exists, the universal file will not be compiled, and the symbol table will not be available. In addition, if the universal routine is modified, all routines which use it must be recompiled by either using /COMPIL or deleting all REL files.

2.9.2 SEARCH Name

The SEARCH statement opens the specified symbol table for MACRO to scan if the required symbol is not found in the current symbol table. Multiple symbol tables may be specified by separating them with commas; they are searched in the order specified. A maximum of ten symbol tables may be specified since each name requires four words of core. This maximum may be redefined with the symbol .UNIV in MACRO.

When the SEARCH pseudo-op is seen, the specified names are compared with the UNIVERSAL table. If the specified names cannot be found, the message

    CANNOT FIND UNIVERSAL name

is output. If the specified names are found, a table of searching sequence is built. This sequence is to search the universal symbol tables in the order specified whenever a symbol is not found in the current symbol table. This search is to continue until the symbol is found or all the tables have been searched. When a symbol is found in an auxiliary symbol table, it is moved into the current symbol table. This procedure saves time on future references at the expense of core.

Universal files may search other universal files as long as all names in the search list have been assembled. The table of universal names is cleared on each RUN or START, but is not cleared when MACRO responds with an asterisk.

Chapter 3
Macros

When writing a program, certain coding sequences are often used several times with only the arguments changed. If so, it is convenient if the entire sequence can be generated by a single statement. To do this, the coding sequence is defined with dummy arguments as a macro instruction. A single statement referring to the macro by name, along with a list of real arguments, generates the correct sequence.

3.1 DEFINITION OF MACROS

The first statement of a macro definition must consist of the operator DEFINE followed by the symbolic name of the macro. The name must be constructed by the rules for constructing symbols. The macro name may be followed by a string of dummy arguments enclosed in parentheses. The dummy arguments are separated by commas and may be any symbols that are convenient--single letters are sufficient. A comment may follow the dummy argument list.

The character sequence, which constitutes the body of the macro, is delimited by angle brackets. The body of the macro normally consists of a group of complete statements.

For example, this macro computes the length of a vector:

```
DEFINE VMAG (A,B)      ;ROUTINE FOR THE LENGTH OF A VECTOR
<MOVE Ø,A              ;GET THE FIRST COMPONENT
FMP Ø                  ;SQUARE IT
MOVE 1,A+1             ;GET THE SECOND COMPONENT
FMP 1,1                ;SQUARE IT
FAD 1                  ;ADD THE SQUARE OF THE SECOND
MOVE 1,A+2             ;GET THE THIRD COMPONENT
FMP 1,1                ;SQUARE IT
FAD 1                  ;ADD THE SQUARE OF THE THIRD
JSR FSQRT              ;USE THE FLOATING SQUARE ROOT ROUTINE
MOVEM B                ;STORE THE LENGTH>
```

NOTE

Storing comments in a macro takes up space. If the comments start with a double semicolon (;;) the comment will not be stored; therefore, it lists in the original definition but does not list when the macro is expanded.

3.2 MACRO CALLS

A macro may be called by any statement containing the macro name followed by a list of arguments. The arguments are separated by commas and may be enclosed with parentheses. If parentheses are used (indicated by an open parenthesis following the macro name), the argument string is ended by a closed parenthesis. If there are n dummy arguments in the macro definition, all arguments beyond the first n, if any, are ignored. If parentheses are omitted, the argument string ends when all the dummy arguments of the macro definitions have been assigned, or when a carriage return or semicolon delimits an argument.

The arguments must be written in the order in which they are to be substituted for dummy arguements. That is, the first argument is substituted for each appearance of the first dummy argument; the second argument is substituted for each appearance of the second dummy arguemnt, etc. For example the appearance of the statement:

```
VMAG VEC, LENGTH
```

in a program generates the instruction sequence defined above for the macro VMAG. The character string VECT is substituted for each occurrence in the coding of the dummy argument A, and the character string LENGTH is substituted for the single occurrence of B in the coding.

Statements with a macro call may have label fields. The value of the label is the location of the first instruction generated.

CAUTION

MACRO arguments are terminated only by COMMA, CARRIAGE RETURN, SEMICOLON or CLOSE PARENTHESIS (when the entire argument string was started with an open parenthesis). These characters may not be included in arguments unless <> are used. Specifically, spaces or tabs do not terminate arguments; they will be treated as part of the argument itself. The symbol does not terminate arguments, it just permits commas and other symbols to be used as part of an argument.

3.3 MACRO FORMAT

a. Arguments must be separated by commas. However, arguments may also contain commas. For example:

```
DEFINE JFQ(A,B,C)
<MOVE [A]
CAMN B
JRST C>
```

If the data in location B is equal to A (a literal), the program jumps to C. If A is to be the instruction ADD 2,X, the calling macro instruction would be written

```
JEQ<ADD 2,X>,B,INSTX
```

The angle brackets surrounding the argument are removed, and the proper coding is generated.

The general rule is: If an argument contains commas, semicolons, or any other argument delimiters, the argument must be enclosed in angle brackets. For every level of nesting, one set of angle brackets is removed; therefore, to pass arguments containing commas to nested macros the argument should be enclosed by one set of angle brackets for each level of nesting. The > does not terminate the argument, a comma must be used.

b. A macro need not have arguments. The instruction:

```
DATAO PIP,PUNBUF(4)
```

which causes the contents of PUNBUF, indexed by register 4, to be punched on paper tape, may be generated by the macro:

```
DEFINE PUNCH
<DATAO PIP,PUNBUF(4)>
```

The calling macro instruction could be written:

```
PUNCH
```

PUNCH calls for the DATAO instruction contained in the body of the macro.

c. The macro name, followed by a list of arguments, may appear anywhere in a statement. The string within the angle brackets of the macro definition exactly replaces the macro name and argument string. For example:

```
DEFINE L(A,B)<3*<B-A+1>>
```

gives an expression for the number of items in a table where three words are used to store each item. A is the address of the first item, and B is the address of the last item. To load an index register with the table length, the macro can be called as follows:

```
MOVEI X,L(FIRST,LAST)
```

3.4 CREATED SYMBOLS

When a macro is called, it is often convenient to generate symbols without explicitly stating them in the call, for example, symbols for labels within the macro body. If it is not necessary to refer to these labels from outside the macro, there is no reason to be concerned as to what the labels are. Nevertheless, different symbols must be used for the labels each time the macro is called. Created symbols are used for this purpose.

Each time a macro that requires a created symbol is called, a symbol is generated and inserted into the macro. These generated symbols are of the form..hijk, that is, two decimal points followed by four digits. The user is advised not to use symbols starting with two points. The first created symbol is ..0001, the next is ..0002, etc.

If a dummy symbol in a definition statement is preceded by a percent sign (%), it is considered to be a created symbol. When a macro is called, all missing arguments that are of the form %X are replaced by created symbols. However, if there are sufficient arguments in the calling list that some of the arguments are in a position to be assigned to the dummy arguments of the form %X, the percent sign is overruled and the stated argument is assigned in the normal manner.

Null arguments are not considered to be the same as missing arguments. For example, suppose a macro has been defined with the dummy string:

```
(A,%B,%C)
```

If the macro were called with the argument string:

```
(OPD,) or OPD,,
```

The second argument would be considered to have been declared as null string. This would override the % prefixed to the second dummy argument and would substitute the null string for each appearance of the second dummy argument in the statement. However, the third argument is missing. A label would be created for each occurrence of %C. For example:

```
DEFINE TYPE(A,%B)
<JSR TYPEOUT
JRST %B
SIXBIT/A/
%B:>
```

This macro types the text string substituted for A on the console Teletype. TYPEOUT is an output routine. Labeling the location following the text is appropriate since A may be text of indefinite length. A created symbol is appropriate for this label since the programmer would not normally reference this location. This macro might be called by:

```
TYPE HELLO
```

which would result in typing HELLO when the assembled macro is executed. If the call had been:

```
TYPE HELLO,BX
```

the effect would be the same. However, BX would be substituted for %B, overruling the effect of the percent sign.

3.5 CONCATENATION

The apostrophe character or single quote (') is defined as the concatenation operator. A macro argument need not be a complete symbol Rather, it may be a string of characters which form a complete symbol or expression when joined to characters already contained in the macro definition. This joining, called concatenation, is performed by the assembler when the programmer writes an apostrophe between the strings to be so joined. As an example, the macro:

```
DEFINE J(A,B,C)
<JUMP'A B,C>
```

When called, the argument A is suffixed to JUMP to form a single symbol. If the call were:

```
J (LE,3,X+1)
```

the generated code would be:

```
JUMPLE 3,X+1
```

The concatenation (') may be used in nested macros. The assembler removes one operator when it performs concatenation if it is next to (before or after) a dummy argument.

3.6 DEFAULT ARGUMENTS

Missing arguments in macros are generally replaced by nulls. For example, the macro

```
DEFINE FOO (A,B,C)>
EXP A,B,C>
```

when called by FOO(1) would generate three words of 1, Ø, and Ø.

Default arguments may be supplied to override missing arguments. When supplied, default arguments are written within angle brackets (<>) after each argument. For example, the addition of default arguments 222 and 333 to arguments B and C of the foregoing example macro would be written as

```
DEFINE FOO (A,B<222>, <<333>)
EXP A,B,C>
```

If the foregoing macro is called by FOO(1) it would generate the number 1,222,333.

The following example program illustrates the use of defined default arguments.

```
.MAIN    MACRO 47(113) 1Ø:14 28-MAR-72 PAGE 1
FOO      MAC      28-MAR-72 1Ø:13
                                      DEFINE FOO1 (A,B,C)<
                                      EXP A,B,C>
                                      DEFINE FOO2 (A<111>,B<222>,C<333
>)<
                                      EXP A,B,C>
                                      FOO1 (1)↑
        ØØØØØØ' ØØØØØØ  ØØØØØ1        EXP1,,↑
        ØØØØØ1' ØØØØØØ  ØØØØØØ
        ØØØØØ2' ØØØØØØ  ØØØØØØ        FOO2 (1)↑
        ØØØØØ3' ØØØØØØ  ØØØØØ1        EXP 1,222,333↑
        ØØØØØ4' ØØØØØØ  ØØØ222
        ØØØØØ5' ØØØØØØ  ØØØ333
                                      END
NO ERRORS DETECTED
PROGRAM BREAK IS ØØØØØ6
2K CORE USED
```

3.7 INDEFINITE REPEAT

It is often convenient to be able to repeat a macro one or more times for a single call, each repetition substituting successive arguments in the call statement for specified arguments in the macro. This may be done by use of the indefinite repeat operator, IRP. The operator IRP is followed by a dummy argument, which may be enclosed in parentheses. This argument must also be contained in the DEFINE statement's list. This argument is broken into subarguments. When the macro is called, the range of the IRP is assembled once for each subargument, the successive subarguments being substituted for each appearance of the dummy argument within the range of the IRP. For example, the single argument:

```
<ALPHA,BETA,GAMMA>
```

consists of the subarguments ALPHA,BETA, and GAMMA. The macro definition:

```
DEFINE DOEACH(A),
<IRP A
<A>>
```

and the call:

```
DOEACH<ALPHA,BETA,GAMMA>
```

produce the following coding:

```
ALPHA
BETA
GAMMA
```

An opening angle bracket must follow the argument of the IRP statement to delimit the range of the IRP since the argument is one argument to the macro. A closing angle bracket must terminate the range of the IRP. IRPC is like IRP except it takes only one character at a time; each character is a complete argument. An example of a program that uses an IRPC is given in Chapter 7, Figure 7-4.

It is sometimes desirable to stop processing an indefinite repeat depending on conditions given by the assembler. This is done by the operator STOP1. When the STOP1 is encountered, the macro processor finishes expanding the range of the IRP for the present argument and terminates the repeat action. An example:

```
DEFINE CONVERT (A)
<IRP A<IFE K-A,<STOPI
CONV1 A>>
```

Assume that the value of K is 3: then the call:

```
CONVERT Ø,1,2,3,4,5,6,7

.

<IRP
IFE K-Ø,<STOPI
CONV1 Ø>
IFE K-1,<STOP1
CONV1 1>
IFE K-2,<STOPI
CONV1 2>
IFE K-3,<STOPI
CONV1 3>
```

The assembly condition is not met for the first three arguments of the macro. Therefore, the STOPI code is not encountered until the fourth argument, which is the number 3. When the condition is met, the STOPI code is processed which prevents further scanning of the arguments. However, the action continues for the current argument and generates CONV1 3, i.e., a call for the macro CONV1 (defined elsewhere) with an argument of 3.

3.8 NESTING AND REDEFINITION

Macros may be nested; that is, macros may be defined within other macros. For ease of discussion, levels may be assigned to these nested macros. The outermost macros, i.e., those defined directly to the macro processor, may be called first level macros. Macros

defined within first level macros may be called second level macros; macros defined within second level macros may be called third level macros; etc.

At the beginning of processing, first level macros are known to the macro processor and may be called in the normal manner. However, second and higher level macros are not yet defined. When a first level macro containing second and higher level macros is called, all its second level macros become defined to the processor. Thereafter, the level of definition is irrelevant, and macros may be called in the normal manner. Of course, if these second level macros contain third level macros, the third level macros are not defined until the second level macros containing them have been called.

When a macro of level n contains a macro of level n+1, calling the macro results in generating the body of the macro into the user's program in the normal manner until the DEFINE statement is encountered. The level n+1 macro is then defined to the macro processor; it does not appear in the ueer's program. When the definition is complete, the macro processor resumes generating the macro body into the user's program until, or unless, the entire macro has been generated.

If a macro name which has been previously defined appears within another definition statement, the macro is redefined, and the original definition is eliminated.

The first example of a macro calculation of the length of a vector may be rewritten to illustrate both nesting and redefinition.

```
DEFINE VMAG (A,B,%C)
<DEFINE VMAG (D,E)
<JSP SJ,VL
EXP C,E>
VMAG (A,B)
      JRST %C
VL:   HRRZ 2, (SJ)
      MOVE (2)
      FMP Ø
      MOVE 1,1(2)
      FMP 1,1
      FAD 1
      MOVE 1,2(2)
      FMP 1,1
      FAD 1
      JSR FSQRT
      MOVEM @1 (SJ)
      JRST 2(SJ)
%C:>
```

The procedure to find the length of a vector has been written as a closed subroutine. It need only appear once in a user's program. From then on it can be called as a subroutine by the JSP instruction.

The first time the macro VMAG is called, the subroutine calling sequence is generated followed immediately by the subroutine itself. Before generating the subroutine, the macro processor encounters a DEFINE statement containing the name VMAG. This new macro is defined and takes the place of the original macro VMAG. Henceforth, when VMAG is called, only the calling sequence is generated. However, the original definition of VMAG is not removed until after the expansion is complete.

Another example of a nested macro is given in Chapter 7, Figure 7-4.

3.8.1 ASCII Interpretation

If the reverse slash (\) is used as the first character of an argument in a macro call, the value of the following symbol is converted to a 7-bit ASCII character in the current radix. If the call is

```
        MAC  \A
```

and if A=500 (in the current radix), this generates the three ASCII character "500".

Chapter 4
Error Detection

MACRO-10 makes many error checks as it processes source language statements. If an apparent error is detected, the assembler prints a single letter code in the left-hand margin of the program listing (and on the TTY, unless the listing is on the TTY), on the same line as the statement in question.

The programmer should examine each error indication to determine whether or not correction is required. At the end of the listing, the assembler prints a total of errors found; this is printed even if no listing is requested.

Each error code indicates a general class of errors. These errors, however, are all caused by illegal usage of the MACRO-10 language, as described in the preceding three chapters of this manual.

4.1 SINGLE-LETTER ERROR CODES

Table 4-1 lists the single-letter error codes output by the assembler.

TABLE 4-1

Error Codes

Error Code	Meaning	Explanation
A	Argument error in pseudo-op	This is a broad class of errors which may be caused by an improper argument in a pseudo-op. The following represent the majority of the conditions which would cause an A code error. a. Symbol used is improperly formed. For example AB?CD would result in an A code since the character ? is not in the Radix 50 character set. b. IFIDN comparison string is too large. c. OPDEF of macro is SYN,. d. OPDEF, no code generated. e. Invalid SIXBIT character in SIXBIT/TEST Tab/ f. Byte size too big in byte (>4D36). g. Radix 50 code not absolute, that is Radix 50 FOO,BAR where FOO is not Ø-74 absolute. h. End of line on IFx SYM reached before an < character is seen. i. Assignment made in an address field (e.g., MOVE A=1Ø). j. Assignment of a label (e.g., TAG: TAG=1). k. Missing symbol in SYN SYM1,. l. Unknown symbol in SYN,. m. Missing right parenthesis ()) in index (e.g., MOVE 1,(2...). n. Missing left parenthesis in BYTE statement (e.g., BYTE 3 1, 1, 1). o. No comma after repeat count (e.g., REPEAT 3 <). p. IRP not in a macro.

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		q. Argument for IRP is not a dummy symbol; for example DEFINE FOO (A) < IRP(B), <>> r. IRP argument is a created symbol. s. STOP1 not in IRP.
D	Multiply-defined symbolic reference error	This statement contains a tag which refers to a multiply-defined symbol. It is assembled with the first value defined.
E	External symbol error	Improper usage of an external symbol. The following represent the majority of the conditions which will cause an E code error. a. Attempting to use the same symbol as both an external and an internal symbol. For example, the statement EXT: EXTERN TXT,BRT,EXT attempts to use EXT as both an external and an internal symbol. b. Using an external symbol for an AC or index. c. Using an external symbol for IFx. d. Using an external symbol in a LOC, RELOC, PHASE, HISEG or TWOSEG pseudo-op. e. Using an external symbol in the left half of IOWD. f. Using an external symbol in an ARRAY size statement. g. Using an external symbol in a REPEAT count.
L	Literal error	A literal is improper. A literal must generate 1 to 18 words. EXP [SIXBIT //];NO CODE GENERATED
M	Multiply-defined symbol	A symbol is defined more than cnce. The symbol retains its first definition, and the error message M is typed out during pass 1. If this type of error occurs during pass 2, it is a phase error (see below).

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		If a symbol is first defined as a #-sign suffixed tag, and later as a label, it retains the label definition. Examples: A: ADD 3,X; A: MOVE ,C; M ERROR A: ADD 3,X#; X: MOVE ,C; X IS ASSIGNED THE CURRENT VALUE OF THE LOCATION COUNTER Multiple appearances of the TITLE pseudo-op (which generates both a title line and program name) are flagged as "M" (Multiple definition) errors.
N	Number error	A number is improperly entered. The following represent the majority of the conditions which would cause an N-type error. a. The number exceeds the permitted range (e.g., ↑F13.33E38). b. A number does not follow a B shift operator (e.g., ↑D15BZ). c. The number exceeds the current radix (e.g., if radix is 8 the single character 9 is acceptable but the number 19 is not acceptable). d. The binary shift given does not represent an absolute numeric. For example, 4B<sym> is illegal if sym is relocatable. e. The character given after an up arrow (↑) is not B, O, F, L or D. f. The expression given after E was not a signed (±) number.
O	Operation code undefined	The operation field of this statement is undefined. It is assembled with a numeric code of Ø.
P	Phase error	A symbol is assigned a value as a label during pass 2 different from that which it received during pass 1. In general, the assembler should generate the same number of program locations in pass 1 and pass 2, and any discrepancy causes a phase error.

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		For example, if an assembly conditional, IF1, generates three instructions, a phase error results unless another conditional, such as IF2, generates three program locations during pass 2.
Q	Questionable	This is a broad class of possible errors in which the assembler finds ambiguous language. Q-errors may or may not generate correct code; the assembler will attempt to do what the programmer intended. The following represent the majority of the conditions which would cause a Q-type error. a. More than 5 ASCII characters are detected by the assembler before a closing " symbol is detected (e.g., "ABCDEFG" or "ABC). When more than 5 characters are detected, only the first 5 are stored. b. More than 6 SIXBIT characters are detected by the assembler before a closing " symbol is detected. As in item a, only the first 6 characters are stored when more than 6 are detected. c. A given number is too big; in such cases, the high-order bits of the number are lost. d. E in a number is followed by something other than a signed (±) numeric (e.g., 1.ØEX). e. An illegal control character is detected in a line. ASCII characters Ø-4Ø are not permitted except for HT, LF, VT, EF, CR and ESC. f. A comma is detected in a statement after all of the required fields have been filled (e.g., MOVE 1,2,) g. Relocatable code is generated by the assembler before either the pseudo-op HISEG or TWOSEG is found by the assembler.

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		h. An instruction address pointer is detected by the assembler which does not have either all Ø's or all 1's in the left half of its word location.
R	Relocation error	A LOC or RELOC pseudo-op is used improperly. All of the following conditions will cause an R-type error. a. An expression or assignment is made in which relocation is not Ø or 1 (e.g., A+B, A*Z, 1/B, or X=3*B where a and B are relocatable). b. A BLOCK statement is written with a relocatable size (e.g., BLOCK: A where A is relocatable). c. A relocatable variable is used to specify an accumulator (e.g., MOVE A,1 where A is relocatable).
U	Undefined symbol	A symbol is undefined.
V	Value previously undefined	A symbol used to control the assembler is undefined prior to the point at which it is first used. Causes error message in pass 1. For example, BLOCK:A where A is undefined.
X	Macro definition error	An error occurred in defining or calling a macro.

Error messages printed during pass 1 consist of two parts. The page and sequence number, if used, plus the most recently used label is printed on the first line. This material is then followed by +n, where n is the (decimal) number of lines of coding between the labeled statement and the statement containing an error. The second line of the error message is a copy of the erroneous line of coding, with a letter code in the left-hand margin to indicate the type of error. If more than one type of error occurs on the same line, more than one letter is printed; but if the same type of error occurs more than once in the same line, a single letter code is printed.

During pass 2, as the listing is printed out, lines containing errors are marked by letter codes, and a total of errors found is printed at the end of the listing.

4.2 ERROR MESSAGES

The following error messages may be typed out on the user's terminal. Any error message preceded by a question mark (?) is treated as a fatal error when running under the BATCH processor (the run is terminated by BATCH).

Message	Meaning
END OF PASS 1	This message indicates that manual loading is required to start pass 2. This message is issued when the input is paper tape, cards or keyboard.
LOAD THE NEXT FILE	This message indicates that manual loading is required when the files to be input are on paper tape, cards or being input from the terminal.
?COMMAND ERROR	This message indicates that an error was found in the last command string input.
?INSUFFICIENT CORE	Not enough core is available.
?.PDL OVERFLOW,TRY/P	This message indicates that the pushdown list is too small. The use of a /P switch increases the size of the pushdown list by 80 locations. As many /P switches may be used as desired.
?DEV NOT AVAILABLE	The specified device cannot be initialized because another user is using it.
?N ERRORS DETECTED ?1 ERROR DETECTED NO ERRORS DETECTED	These three statements indicate the number of errors detected by MACRO during assembly (errors marked by letter codes on the listing. Under BATCH if any error occurs, the run is terminated.
?NO END STATEMENT ENCOUNTERED ON INPUT FILE	This message is followed by one of the following: IN LITERAL IN DEFINE IN TEXT IN CONDITIONAL OR REPEAT IN CONDITIONAL IN MACRO CALL

and

ON PAGE xxx AT yyy

where xxx = a page number and yyy = a sequence number or TAG+offset.

NOTE

The foregoing type of message usually indicates some error other than a missing END statement. For example:

```
ASCIZ/TEXT
    :
END
```

where TEXT has not been closed or

```
JRST [statements
    :
END
```

where the literal has not been closed.

?PRGEND ERROR — This error message indicates that the macro failed to restore the symbol table for one of the programs.

?TOO MANY UNIVERSALS — This error message indicates that too many universal programs have been assembled. The number of universal programs permitted is a Macro parameter; to prevent this error from reoccurring, the user must reassemble macro with a new parameter which will permit the desired assembly.

?CANNOT FIND UNIVERSAL xxx — This message indicates that a search has been made for UNIVERSAL program xxx but it was not found (i.e., it was not assembled). To clear this error the program xxx must be assembled.

xxx UNASSIGNED DEFINED AS IF EXTERNAL — This message indicates that an undefined symbol was found and that it has been treated as if it was an external symbol.

PROGRAM BREAK IS xxx — Where xxx is the length of the low segment.

HI-SEG BREAK IS xxx — Where xxx is the length of the relocated high segment.

ABSOLUTE BREAK IS xxx — Where xxx is the highest absolute address seen over 14Ø.

xK CORE USED — Message indicates the size of the low segment used to assemble the source program.

?UNIVERSAL PROGRAM(S) MUST HAVE SAME OUTPUT SPECIFICATIONS AS OTHER FILES

This error message indicates that a universal program was found which did not have either a binary or a listing device specified but all of the following files had such specifications. For example the sequence

```
*,←UNIV
*rel,List←file
```

is illegal. The legal sequence would be

```
*rel, LIST←UNIV
*REL,LIST←FILE
```

?ERROR WHILE EXPANDING xxx — This error message indicates that the assembler experienced an internal error while expanding the macro identified as xxx. Errors of this type are extremely rare; if it occurs the user should rewrite the macro involved.

4.2.1 LOOKUP Errors

The following error messages can occur during a monitor LOOKUP, RENAME or ENTER request on disk. The form of the error messages is:

? filename.ext then one of the following

(Ø) FILE WAS NOT FOUND or (Ø) ILLEGAL FILE NAME (used for enter errors only)

(1) NO DIRECTORY FOR PROJECT-PROGRAMMER NUMBER

(2) PROTECTION FAILURE

(3) FILE WAS BEING MODIFIED

(4) RENAME FILE NAME ALREADY EXISTS

(5) ILLEGAL SEQUENCE OF UUOS

(6) BAD UFD OR BAD RIB

(7) NOT A SAV FILE

(1Ø) NOT ENOUGH CORE

(11) DEVICE NOT AVAILABLE

(12) NO SUCH DEVICE

(13) NOT TWO RELOC REG. CAPABILITY

(14) NO ROOM OR QUOTA EXCEEDED

(15) WRITE LOCK ERROR

(16) NOT ENOUGH MONITOR TABLE SPACE
(17) PARTIAL ALLOCATION ONLY
(2Ø) BLOCK NOT FREE ON ALLOCATION
(21) CAN'T SUPERSEDE (ENTER) AN EXISTING DIRECTORY
(22) CAN'T DELETE (RENAME) A NON-EMPTY DIRECTORY
(23) SFD NOT FOUND
(24) SEARCH LIST EMPTY
(25) SFD NESTED TOO DEEPLY
(26) NO-CREATE ON FOR SPECIFIED SFD PATH

If the error code (V) is greater than 26_8, the error message:

?(V) LOOKUP,ENTER, OR RENAME ERROR

is printed.

4.2.2 MACRO I/O Error Messages

The following error messages are generated for error conditions found during input or output operations with peripheral devices. The messages are self-explanatory.

```
?OUTPUT WRITE-LOCK ERROR DEVICE xxx
?OUTPUT DATA ERROR DEVICE xxx
?OUTPUT CHECKSUM OR PARITY ERROR DEVICE xxx
?OUTPUT QUOTA EXCEEDED ON DEVICE xxx
?OUTPUT BLOCK TOO LARGE DEVICE xxx
?MONITOR DETECTED SOFTWARE INPUT ERROR DEVICE xxx
?INPUT DATA ERROR DEVICE xxx
?INPUT CHECKSUM OR PARITY ERROR DEVICE xxx
?INPUT BLOCK TOO LARGE DEVICE xxx
```

Chapter 5
Relocation

The MACRO-10 assembler will create a relocatable object program. This program may be loaded into any part of memory as a function of what has been previously loaded. To accomplish this, the address field of some instructions must have a relocation constant added to it. This relocation constant, added at load time by the PDP-10 Loader, equals the difference between the memory location an instruction is actually loaded into and the location it is assembled into. If a program is loaded into cells beginning at location 1400_8, the relocation constant k would be 1400_8.

Not all instructions must be modified by the relocation constant. Consider the two instructions:

```
MOVEI 2,.-3
MOVEI 2,1
```

The first is used in address manipulation and must be modified; the second probably should not. To accomplish the relocation, the actual expression forming an address is evaluated and marked for modification by the Linking Loader. Integer elements are absolute and not modified. Point elements (.) are relocatable and are always

modified.[1] Symbolic elements may be either absolute or relocatable. If a symbol is defined by a direct assignment statement, it may be relocatable or absolute depending on the expression following the equal sign (=). If a symbol is defined as a macro, it is replaced by the string and the string itself is evaluated. If it is defined as a label or a variable (#), it is relocatable.[1] Finally, references to literals are relocatable.[1]

To evaluate the relocatability of an expression, consider what happens at load time. A constant, k, must be added to each relocatable element and the expression evaluated. Consider the expression:

X - A+2*B-3*C + D

where A,B,C, and D are relocatable. Assume k is the relocation constant. Adding this to each relocatable term we get:

X_R = (A+K)+2*(B+K)-3*(C+K)+(D+K)

This expression may be rearranged to separate the k's, yielding:

X_R = A+2*B-3*C+D+K

This expression is suitable for relocation since it involves the addition of a single k. In general, if the expression can be rearranged to result in the addition of

Ø*K	The expression is legal and fixed.
1*K	The expression is legal and relocatable.
N*K	Where n is any positive or negative integer other than 0 or 1, the expression is illegal.

Finally, if the expression involves k to any power other than 1, the expression is illegal. This leads to the following conventions:

a. Only two values of relocatability for a complete expression are allowed (e.g., nK where n = Ø or +1).

b. An element may not be divided by a relocatable element.

c. Two relocatable elements may not be multiplied together.

d. Relocatable elements may not be combined by the Boolean operators.

---

[1]Except under the LOC code or PHASE code which specifies absolute addressing.

If any of these rules is broken, the expression is illegal and the assembled code is flagged.

If A, C, and B are relocatable symbols, then:

```
A+B-C    is relocatable
A-C      is fixed
A+2      is relocatable
2*A-B    is relocatable
2&A-B    is illegal
```

A storage word may be relocatable in the left half as well as in the right half. For example:

```
XWD A,B
```

Chapter 6
Assembly Output

There are two MACRO-10 outputs, a binary program and a program listing. The listing is controlled by the listing control pseudo-ops, which were described in Chapter 2.

6.1 ASSEMBLY LISTING

All MACRO-10 programs begin with an implicit LIST statement.

Each page begins with a TITLE line; this line contains the program's name, the assembler version, the time of assembly, the date of assembly and a page number. The page number is incremented by a Form-Feed or PAGE pseudo-op.

If the code listed requires more than one page, the basic page number given on the title line does not change but a subpage number is added and incremented for each additional page (e.g., 6-1, 6-2, 6-3, etc.).

The second line printed on each page is the SUBTITLE line. This line contains the program filename and extensions, creation time, creation date and any given subtitle.

From left to right, the columns on a listing page contains:

a. The 6-digit address of each storage word in the binary program. These are normally sequential location counter assignments. In the case of a block statement, only the address of the first word allocated is listed. An apostrophe following the address indicates that the address is relocatable.

b. The assembled instructions and data words shown in one of several forms for easier reading (see paragraph 2.6.3).

c. The source program statement, as written by the programmer, followed by comments, if any.

If an error is detected during assembly of a statement, an error code is printed on that statement's line, near the left edge of the page. If multiple errors of the same type occur in a particular statement, the error code is printed only once; but if several errors, each of a different type, occur in a statement, an error code is printed for each error. The total number of errors is printed at the end of the listing.

The program break is also printed at the end of the listing. This is the highest relocatable location assembled, plus one. This is the first location available for the next program or for patching.

6.2 BINARY PROGRAM OUTPUT

The assembler produces binary program output in four formats. The choice depends on whether the program is relocatable or absolute, and on the loading procedure to be used to load the program for execution.

6.2.1 Relocatable Binary Programs - LINK Format

Most binary programs are output in LINK format. Like the RELOC statement, the LINK format output is implicit and is automatically produced for all relocatable MACRO-10 programs unless another format (RIM, RIM10, RIM10B) is explicitly requested. The LINK format is the only format that may be used with the Linking Loader.

The Linking Loader loads subprograms into memory, properly relocating each one and adjusting addresses to compensate for the relocation.

It also links external and internal symbols to provide communication between independently assembled subprograms. Finally, the Linking Loader loads required subroutines while in Library Search Mode.

Data for the Linking Loader is formatted in blocks. All blocks have an identical format. The first word of a LINK block consists of two halves. The left half is a code for the block type, and the right half is a count of the number of data words in the block. The data words are grouped in sub-blocks of 18 items. Each 18-word sub-block is preceded by a relocation word. This relocation word consists of 18 2-bit bytes. Each byte corresponds to one word in the sub-block, and contains relocation information regarding that word.

If the byte value is:

0	no relocation occurs
1	the right half is relocated
2	the left half is relocated
3	both halves are relocated

These relocation words are not included in the count; they always appear before each sub-block of 18 words or less to ensure proper relocation.

All relocatable programs may be stored in LINK format, including programs on paper tape, DECtape, magnetic tape, punched cards, and disks. This format is totally independent of logical divisions in the input medium. It is also independent of the block type.

6.2.1.1 LINK Formats for the Block Types - Block Type 1 Relocatable or Absolute Programs and Data

WORD 1	The location of the first data word in the block
WORD 2	A contiguous block of program or data words (18 or less)
.	
.	
.	
WORD N	(N, from 1 to 18, must be less than 2000,000 octal)

Block Type 2 Symbols

	Consists of word pairs
1ST WORD	Bits 0-3 code bits
1ST WORD	Bits 4-35 radix 50 representation of symbol (see below)
2ND WORD	Data (value or pointer)
CODE Ø4:	Global (internal) definition
2ND WORD	Bits 0-35 value of symbol
CODE 1Ø:	Local definition
2ND WORD	Bits 0-35 value of symbol
CODE 6Ø:	Chained global requests:
2ND WORD	Bits 0-17=0
2ND WORD	Bits 18-35 pointer to first word of chain requiring defintion (refer to the LOADER manual)
CODE 6Ø:	Global symbol additive request: (refer to the LOADER manual)

Block Type 3 Load Into High Segment

When block type 3 is present in a relocatable binary program, the Loader loads the program into the high segment if the system has re-entrant (two-segment) capability. When used, block type 3 appears immediately after the name block (type 6).

The first word is

```
XWD 3,,2
```

The second word is the relocation word

```
2ØØØØØ,,Ø
```

The third word is

```
XWD HISEG BREAK,,TWOSEG ORIGIN
```

where twoseg origin is 400000 by default.

With the TWOSEG pseudo-op, the left half of the third word is negative. On a two-segment machine,this is ignored except to set a LOADER flag. On a one-segment machine, the difference is assumed to be the maximum length of the high segment. A one-pass assembler does not know this legth at the start of pass 1, therefore

XWD 4ØØØØØ,,4ØØØØØ

is used to signal two segments to a two-segment machine.

On a one-segment machine, this instruction gives the error message

TWO SEGMENTS ILLEGAL

since the LOADER does not know how much space to reserve for the high segment.

Block Type 4 Entry Block

This block contains a list of Radix 50 symbols, each of which may contain a 0 or 1 in the high-order code bit. Each represents a series of logical AND conditions. If all the globals in any series are requested, the following program is loaded. Otherwise, all input is ignored until the next end block. This block must be the first block in a program.

Block Type 5 End Block

This is the last block in a program. It contains two words, the first of which is the program break, that is, the location of the first free register above the program. (Note: This word is relocatable.) It is the relocation constant for the following program loaded. The second word is the highest absolute location seen (if greater than 140). In a two-segment program, the two words are:

1) the high segment break followed by
2) the low segment break.

Block Type 6 Name Block

The first word of this block is the program name RADIX 50). It must appear before any type 2 blocks. The second word, if it appears, defines the length of common. The left half of the second word is used to describe the compiler type that produced the binary file, 0 in the case of MACRO.

Block Type 7 Starting Address

The first word of this block is the starting address of the program. The starting address for a relocatable program may be relocated by means of the relocation bits.

Block Type 10 Internal Request

Each data word is one request. The left half if the pointer to the program. The right half is the value. Either quantity may be relocatable.

6.2.2 Absolute Binary Programs

Three output formats are available for absolute (non-relocatable) binary programs. These are requested by the RIM, RIM10, and RIM10B statements.

6.2.2.1 RIM10B Format - If a program is assembled into absolute locations (not relocatable), a RIM10B statement following the LOC statement at the beginning of the source program causes the assembler to write out the object program in RIM10B format. This format is designed for use with the PDP-10 hardware read-in feature.

The program is punched out during pass 2, starting at the location specified in the LOC statement. If the first two statements in the program are:

```
LOC 1ØØØ)
RIM1ØB )
```

the assembler assembles the program with absolute addresses starting at 1000, and punches out the program in RIM10B format, also starting at location 1000. The programmer may reset the location counter duringassembly of his program, but only one RIM10B statement is needed to punch out the entire program.

In RIM10B format (see Figures 6-1 and 6-2), the assembler punches out the RIM10B Loader (Figure 6-2), followed by the program in 17-word (or less) data blocks, each block separated by blank tape. The assembler inserts an I/O transfer word (IOWD) preceding each data block, and also inserts a 36-bit checksum following each data

block as shown in Figure 6-1. The word count in the IOWD includes only the data words in the block, and the checksum is the simple 36-bit added checksum of the IOWD and the data words.

Data blocks may contain less than 17 words. If the assembler assigns a non-consecutive location, the current data block is terminated, and an IOWD containing the next location is inserted, starting a new data block.

The transfer block consists of two words. The first word of the transfer block is an instruction obtained from the END statement (see Section 6.2.2.4) and is executed when the transfer block is read. The second is a dummy word to stop the reader.

6.2.2.2 RIM10 Format - Binary programs in RIM10 format are absolute, unblocked, and not checksummed. When the RIM10 statement follows a LOC statement in a program, the assembler punches out each storage word in the object program, starting at the absolute address specified in the LOC statement.

RIM10 writes an arbitrary "paper tape". If it is in the format below, it can be read in by the PDP-10 Read-In-Mode hardware.

```
IOWD N,FIRST)
```

where n is the length of the program including the transfer instruction at the end, and FIRST is the first memory location to be occupied. The last location must be a transfer instruction to begin the program, such as:

```
JRST 4,GO)
```

For example, if a program with RIM10 output has its first location at START and its last location at FINISH, the programmer may write

```
IOWD FINISH-START+1,START)
```

NOTE

In cases where the location counter is increased but no binary output occurs (such as with BLOCK, LOCn, and LIT pseudo-ops), MACRO inserts a zero word into the binary output file for each location skipped by the location counter.

6.2.2.3 RIM Format - This format, which is primarily used in PDP-6 systems, consists of a series of paired words. The first word of each pair is a paper-tape read instruction giving the core memory address of the second word. The second word is the data word.

```
DATAI PTR,LOC
DATA WORD
```

The last pair of words is a transfer block. The first word is an instruction obtained from the END statement (see Section 6.2.2.4) and is executed when the transfer block is read. The second word is a dummy word to stop the reader.

The loader that reads this format is:

```
        LOC 2Ø

        CONO PTR,6Ø
A:      CONSO PTR,1Ø
        JRST .-1
        DATAI PTR,B
        CONSO PTR,1Ø
        JRST .-1
B:      Ø
        JRST A
```

This loader is normally toggled into memory and started at location 20.

6.2.2.4 END Statements - When the programmer wants output in either RIM or RIM10B format, he may insert an instruction or starting address as the first word in the two-word transfer block by writing the instruction or address as an argument to the END statement. The second word of the transfer block is zero. In RIM10 assemblies, this argument is ignored.

If bits 0 through 8 of the instruction are zero, MACRO will insert the instruction JRST 4,0, causing a halt when executed. The END statements

```
END SA)     OR  END JRST SA)
```

will start automatically at address SA.

Some other examples:

```
                1st Transfer Block Word

          END@XCT 1234     XCT@1234
          END Z4,SA        JRST 4,SA
          END              JRST 4,Ø
```

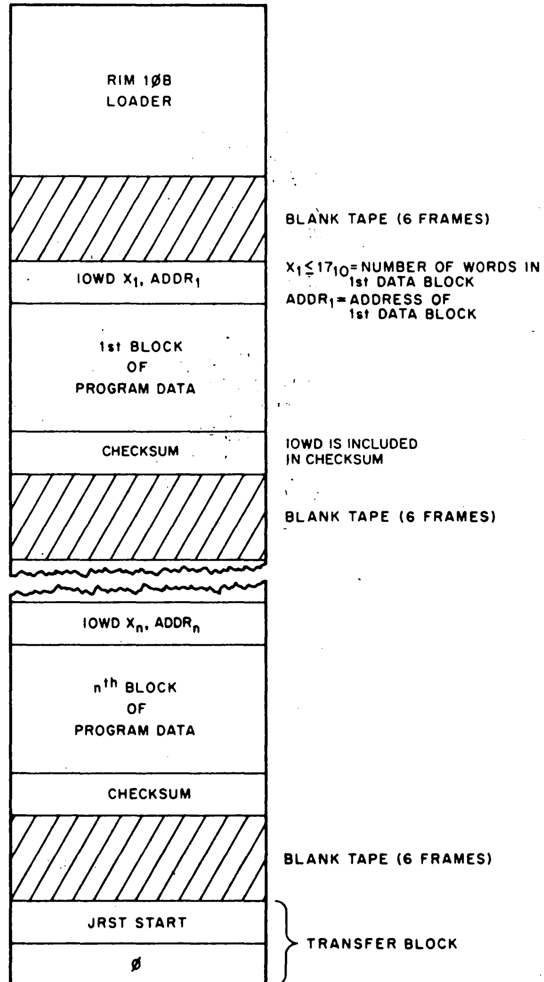


Figure 6-1 General RIM10B Format

```
                XWD -16,Ø
ST:             CONO PTR,6Ø
ST1:            HRRI A,RD+1
RD:             CONSO PTR,1Ø
                JRST
                DATAI PTR,@TBL1-RD+1(A)
                XCT         TBL1-RD+1(A)
                XCT         TBL2-RD+1(A)
A:              SOJA A,
TBL1:           CAME CKSM,ADR
                ADD CKSM,1 (ADR)
                SKIPL CKSM,ADR
TBL2:           JRST 4,ST
                AOBJN ADR,RD
ADR:            JRST     ST1
CKSM=ADR+1
```

Figure 6-2 RIM10B Loader

Chapter 7
Programming Examples

This chapter contains four examples of macro programs. The first example (Figure 7-1) presents a MACRO-10 routine for calculating the logarithm of a complex argument. This routine begins with an ENTRY statement identifying this library routine as CLOG (Complex Logarithm Function) and uses three external routines, ALOG, ATAN2 and CABS.

The second example (Figure 7-2) is the universal parameter file DEF40.MAC which is used to produce the KA-10 version of LIB40. It contains conditional assembly switches to select either a PDP-6, KA10 or KI10 mode. It defines the accumulator conventions and macros which simulate the KI10 hardware operations on the KA10 processor.

Example 3 (Figure 7-3) uses DEF40 (via the SEARCH pseudo-op) for its accumulators and the macros for DMOVE, DMOVEM and FLADD. The macro FLADD is expanded twice to show the effect of LALL on lines which generate text but no binary. The effect of SALL is also shown.

Example 4 (Figure 7-4) shows nested macros which use IRPC. The desired operation is to take an ASCII text string and store the

characters four per word, left-justified, with the character count stored in the first nine bits of the first word.

The TEXT macro counts the string characters and invokes the CODE macro to store the characters four per word.

The CODE macro invokes a SHIFT macro which left-justifies the last word if it is not already left-justified. The first part of the example shows the normal listing, then SALL is set to show what code the macros are generating.

```
CLOG      MACRO 47(113) 13:54  4-APR-72 PAGE 1
CLOG      MAC       4-APR-72 13:53          APRIL 3, 1972

                                                      TITLE CLOG
                                                      SUBTTL APRIL 3, 1972

                                            COMMENT ;COMPLEX LOGARITHMIC FUNCTION
                                            THIS ROUTINE CALCULATES THE LOGARITHM OF A COMPLEX ARGUMENT
                                            Z = X+1*Y WITH THE FOLLOWING ALGORITHM

                                            LOG8Z = LOG 8ABSF (Z) + I*THETA
                                            WHERE ABSF 8Z) = SQRT (X↑2 + Y↑2)
                                            AND THETA IS THE COMPLEX ANGLE ATAN(Y/X)

                                            THE ROUTINE IS CALLED IN THE FOLLOWING MANNER:
                                                    JSA     Q,CLOG
                                                    EXP     ARG
                                            THE REAL PART OF THE ANSWER IS RETURNED IN ACCUMULATOR A
                                            AND THE IMAGINARY PART IS RETURNED IN ACCUMULATOR B;

                                                    ENTRY   CLOG
                                                    EXTERN  ALOG,ATAN2,CABS

                            ØØØØØØ                  A=Ø
                            ØØØØØ1                  B=1
                            ØØØØ1Ø                  C=1Ø
                            ØØØØ11                  D=11
                            ØØØØ16                  Q=16

          ØØØØØØ' ØØØØØØ  ØØØØØØ            CLOG:   Ø                       ;ENTRY TO COMPLEX LOG ROUTINE
          ØØØØØ1' 2Ø1 1Ø 1 16 ØØØØØØ                MOVEI   C,@(Q)          ;GET ADDRESS OF COMPLEX ARGUMENT
          ØØØØØ2' 2ØØ 11 Ø 1Ø ØØØØØ1                MOVE    D,1(C)          ;GET REAL PART OF ARGUMENT
          ØØØØØ3' 2ØØ 1Ø Ø 1Ø ØØØØØØ                MOVE    C,(C)           ;GET REAL PART OF ARGUMENT
          ØØØØØ4' 266 Ø1 Ø ØØ ØØØØØØ*               JSA     1,CABS          ;CALCULATE MAGNITUDE OF Z
          ØØØØØ5' ØØØØØØ  ØØØØØØ                    EXP     C               ;ADDRESS OF COMPLEX ARGUMENT
          ØØØØØ6' 266 16 Ø ØØ ØØØØØØ*               JSA     Q,ALOG          ;CALCULATE LOG(ABSF (Z))
          ØØØØØ7' ØØØØØØ  ØØØØØØ                    EXP     A               ;ADDRESS FOR LOG ROUTINE
          ØØØØ1Ø' 25Ø ØØ Ø ØØ ØØØØ1Ø                EXCH    A,C             ;SWAP ANSWER WITH REAL PART
          ØØØØ11' 266 16 Ø ØØ ØØØØØØ*               JSA     Q,ATAN2         ;CALCULATE ANGLE AS ATAN(Y/X)
          ØØØØ12' ØØØØØØ  ØØØØ11                    EXP     D               ;ADDRESS OF Y
          ØØØØ13' ØØØØØØ  ØØØØØØ                    EXP     A               ;ADDRESS OF C
          ØØØØ14' 2ØØ Ø1 Ø ØØ ØØØØØØ                MOVE    B,A             ;PUT THETA IN IMAGINARY PART
          ØØØØ15' 2ØØ ØØ Ø ØØ ØØØØ1Ø                MOVE    A,C             ;RESTORE REAL PART
          ØØØØ16' 267 16 Ø 16 ØØØØØ1                JRA     Q,1(Q)          ;EXIT
                                                    END
NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØ17

2K CORE USED
```

Figure 7-1 MACRO Program CLOG

```
CLOG     MACRO 47(113) 13:54  4-APR-72 PAGE 2
CLOG     MAC      4-APR-72 13:53         SYMBOL TABLE

A                 ØØØØØØ
ALOG              ØØØØØ6' EXT
ATAN2             ØØØØ11' EXT
B                 ØØØØØ1
C                 ØØØØ1Ø
CABS              ØØØØØ4' EXT
CLOG              ØØØØØØ' ENT
D                 ØØØØ11
Q                 ØØØØ16
```

```
UNIVERSAL DEF4Ø PARAMETER FILE FOR FORTRAN IV LIBRARY
SUBTTL  V32(343)        23-NOV-71       /TWE

IFNDEF PDP6.<IFNDEF KA1Ø,<IFNDEF KI1Ø,<KA1Ø==1>>>
IFNDEF PDP6,<PDP6==Ø>               ;CONDITIONAL ASSEMBLY PARAMETERS
IFNDEF KA1Ø,<KA1Ø==Ø>
IFNDEF KI1Ø,<KI1Ø==Ø>
IFN <PDP6!KA1Ø!KI1Ø-PDP6-KA1Ø-KI1Ø>,
        <PRINTX MACHINE PARAMETERS DEFINED WRONG>



;ACCUMULATOR ASSIGNMENTS
A=Ø
B=1
C=2
D=3
E=4
F=5
G=6
H=7

Q=16    ;FOR JSA AND ARG ADDRESS FOR PUSHJ
P=17    ;PUSH DOWN POINTER

IFE KA1Ø,<
DEFINE DOUBLE (A,B)<
        A
        B>
>

IFN KA1Ø,<
DEFINE DOUBLE (A,B)<
ƵƵ1,==A& 777ØØØ,,Ø>
IFL ƵƵ1.,<ƵƵ1.==-ƵƵ1.-<1ØØØ,,Ø>>
ƵƵ1,==ƵƵ1.-<Ø33ØØØ,,Ø>
IFE B,<ƵƵ1.==Ø>
ƵƵ2,==ƵƵ1.+<<B+2ØØ>←-B>&<ØØØ777,,777777>
IFL ƵƵ1.,<ƵƵ2.==Ø>
        A
        ƵƵ2
SUPPRESS ƵƵ1,,ƵƵ2.>
DEFINE DMOVE(AC,M)<
        IFL <Ƶ M>-<@>,<
        MOVE AC,M
        MOVE AC+1,1+M>

        IFGE <Ƶ M><@>,<
        MOVEI AC+1,M
        MOVE AC,(AC+1)
        MOVE AC+1,1(AC+1)>
>

DEFINE  DMOVN(AC,M)<
        DMOVE AC,M
        DFN AC,AC+1>

DEFINE  DMOVEM(AC,M)<
        MOVEM AC,M
        MOVEM AC+1,1+M

>
```

Figure 7-2   Universal Parameter File DEF40.MAC

```
DEFINE FLMUL (AC,M,%OV)<
        MOVEM AC,AC+2
        FMPR AC+2,1+M
        JFCL (2)
        FMPR AC+1,M
        JFCL (2)
        UFA AC+1,AC+2
        JFCL
        FMPL AC,M
        JOV %OV
        UFA AC+1,AC+2
        FADL AC,AC+2
%OV:>

DEFINE FLDIV(AC,M,%OV)<
        FDVL AC,M
        JOV %OV
        MOVN AC+2,AC
        FMPR AC+2,1+M
        JFCL (2)
        UFA AC+1,AC+2
        FDVR AC+2,M
        JFCL
        FADL AC,AC+2
%OV:>

DEFINE FLADD(AC,M,%OV)<
        UFA AC+1,1+M
        FADL AC,M
        JOV %OV
        UFA AC+1,AC+2
        FADL AC,AC+2
%OV:>

>       ;END OF KA1Ø CONDITIONAL
IFN KI1Ø,<
OPDEF   FLADD   [DFAD]
OPDEF   FLMUL   [DFMP]
OPDEF   FLDIV   [DFDV]

DEFINE DFN (A,B)<        DMOVN A,A
IFN  <<A+1>&17-<B>>,<PRINTX "DMOVN A,A" CAN'T REPLACE "DFN A,B">
>
>       ;END OF KI1Ø CONDITIONAL

END
```

```
TEST SOME MACROS         MACRO 47(113) 13:48  5-APR-72 PAGE 1
TEST    MAC      5-APR-72 13:4Ø           %1 5-APR-72
                                                  TITLE TEST SOME MACROS
                                                  SUBTTL  %1 5-APR-72

                                                  SEARCH  DEF4Ø

        ØØØØØØ'                           START:
                                                  DMOVE   A,(Q)
        ØØØØØØ' 2ØØ ØØ Ø 16 ØØØØØØ                MOVE A,(Q)            ↑;SIMPLE DOUBLE MOVE
        ØØØØØ1' 2ØØ ØØ Ø 16 ØØØØØ1                MOVE A+1,1+(Q)

                                                  DMOVE   A, (Q)         ;THIS ONE INDEXED
        ØØØØØ2' 2Ø1 Ø1 1 16 ØØØØØØ                MOVEI A+1, (Q)
        ØØØØØ3' 2ØØ ØØ Ø Ø1 ØØØØØØ                MOVE A,(A+1)
        ØØØØØ4' 2ØØ Ø1 Ø Ø1 ØØØØØ1                MOVE A+1,1(A+1)

                                                  DMOVEM  A,E            ;STORE TO MEMORY
        ØØØØØ5' 2Ø2 ØØ Ø ØØ ØØØØØ4                MOVEM A,E
        ØØØØØ6' 2Ø2 Ø1 Ø ØØ ØØØØØ5                MOVEM A+1,1+#

                                                  FLADD   A,E
        ØØØØØ7' 13Ø Ø1 Ø ØØ ØØØØØ5                UFA A+1,1+E
        ØØØØ1Ø' 141 ØØ Ø ØØ ØØØØØ4                FADL    A,E
        ØØØØ11' 255 1Ø Ø ØØ ØØØØ14'               JOV ,.ØØØ1
        ØØØØ12' 13Ø Ø1 Ø ØØ ØØØØØ2                UFA A+1,A+2
        ØØØØ13' 141 ØØ Ø ØØ ØØØØØ2                FADL A,A+2

                                                  LALL                   ;LIST EVERYTHING
                                                  FLADD   A,#
        ØØØØ14' 13Ø Ø1 Ø ØØ ØØØØØ5                UFA A+1,1+E
        ØØØØ15' 141 ØØ Ø ØØ ØØØØØ4                FADL A,E
        ØØØØ16' 255 1Ø Ø ØØ ØØØØ21'               JOV ,.ØØØ2
        ØØØØ17' 13Ø Ø1 Ø ØØ ØØØØØ2                UFA A+1,A+2
        ØØØØ2Ø' 141 ØØ Ø ØØ ØØØØØ2                FADL A,A+2
        ØØØØ21'                          ..ØØØ2:  ↑

                                                  SALL                   ;CALL AND BINARY ONLY
        ØØØØ21' 2ØØ ØØ Ø ØØ ØØØØØ4                DMOVE   A,E
                        ØØØØØØ'          END      START

NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØ23

2K CORE USED
```

Figure 7-3 Test Some Macros

```
TEST SOME MACROS        MACRO 47(113) 13:48  5-APR-72 PAGE 2
TEST    MAC      5-APR-72 13:4Ø           SYMBOL TABLE

A               ØØØØØØ
E               ØØØØØ4
Q               ØØØØ16
START           ØØØØØØ'
..ØØØ1          ØØØØ14'
..ØØØ2          ØØØØ21'
```

```
STORE TEXT CHARACTER BY CHARACTER        MACRO 47(113) 14:45 5-APR-72 PAGE 1
TEXT     MAC        5-APR-72 14:44       %1 5-APR-72

                                                 TITLE STORE TEXT CHARACTER BY CHARACTER
                                                 SUBTTL  %1 5-APR-72

                                         DEFINE TEXT (C)<
                                         N==Ø                       ;;INITIAL CONDITION
                                         IRPC C,<N==N+1>            ;;COUNT CHARACTERS
                                         CODE (N,C)                 ;;CALL MACRO TO STORE TEXT
                                         >                          ;END OF TEXT MACRO

                                         DEFINE CODE (N,C)<
                                         ZZ==N                      ;;INITIAL CONDITION
                                         IRPC C,<
                                         IFN ZZ&777B8,<
                                                 EXP ZZ             ;;WORD FULL
                                                 ZZ==Ø>             ;;START AGAIN WITH Ø
                                         ZZ==ZZ←9+"C">              ;;END OF IRPC
                                         IFN ZZ,<DEFINE SHIFT       ;;REMAINDER LEFT
                                          IFE ZZ&777B8,>            ;;IF NOT LEFT JUSTIFIED
                                         ZZ==ZZ←9                   ;;MOVE LEFT 9 BITS
                                         SHIFT>                     ;;RECURSE
                                         >                          ;;END OF DEFINE
                                         SHIFT                      ;;START MOVING LEFT
                                         EXP ZZ>                    ;;END OF IFN ZZ
                                         >                          ;END OF CODE MACRO

                                         TEXT (ABCDEFGHIJKL)↑
         ØØØØØØ' Ø141Ø1  1Ø21Ø3                  EXP ZZ
         ØØØØØ1' 1Ø41Ø5  1Ø61Ø7                  EXP ZZ
         ØØØØØ2' 11Ø111  112113                  EXP ZZ
         ØØØØØ3' 114ØØØ  ØØØØØØ          EXPZZ>

                                                 LALL
                                         TEXT (ABCDEFGHIJKL)↑
                         ØØØØØØ          N==Ø
                                         IRPC
                         ØØØØØ1          N==N+1

                         ØØØØØ2          N==N+1

                         ØØØØØ3          N==N+1
```

Figure 7-4 Store Text Character by Character

```
ØØØØØ4          N==N+1
ØØØØØ5          N==N+1
ØØØØØ6          N==N+1
ØØØØØ7          N==N+1
ØØØØ1Ø          N==N+1
ØØØØ11          N==N+1
```

```
STORE TEXT CHARACTER BY CHARACTER        MACRO 47(113) 14:45  5-APR-72 PAGE 1-1
TEXT    MAC      5-APR-72 14:44          %1 5-APR-72

                        ØØØØ12           N==N+1

                        ØØØØ13           N==N+1

                        ØØØØ14           N==N+1


                                         CODE (N,ABCDEFGHIJKL)↑
                        ØØØØ14           ƵƵ==N
                                         IRPC

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                        Ø141Ø1           ƵƵ==ƵƵ←9+"A"

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                ØØØØ14  1Ø11Ø2           ƵƵ==ƵƵ←9+"B"

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                Ø141Ø1  1Ø21Ø3           ƵƵ==ƵƵ←9+"C"

                                         IFN ƵƵ&777B8,<
        ØØØØØ4' Ø141Ø1  1Ø21Ø3                   EXP ƵƵ
                        ØØØØØØ                   ƵƵ==Ø>
                        ØØØ1Ø4           ƵƵ==ƵƵ←9+"D"

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                        1Ø41Ø5           ƵƵ==ƵƵ←9+"E"
```

```
                          IFN ZZ&777B8,<
                                EXP ZZ
                                ZZ==Ø>
ØØØ1Ø4  1Ø51Ø6            ZZ==ZZ←9+"F"

                          IFN ZZ&777B8,<
                                EXP ZZ
                                ZZ==Ø>
1Ø41Ø5  1Ø61Ø7            ZZ==ZZ←9+"G"
```

```
STORE TEXT CHARACTER BY CHARACTER       MACRO 47(113) 14:45 5-APR-72 PAGE 1-2
TEXT    MAC       5-APR-72 14:44        %1 5-APR-72

                                        IFN ZZ&777B8,<
        ØØØØØ5' 1Ø41Ø5  1Ø61Ø7                  EXP ZZ
                        ØØØØØØ                  ZZ==Ø>
                        ØØØ11Ø          ZZ==ZZ←9+"H"

                                        IFN ZZ&777B8,<
                                                EXP ZZ
                                                ZZ==Ø>
                        11Ø111          ZZ==ZZ←9+"I"

                                        IFN ZZ&777B8,<
                                                EXP ZZ
                                                ZZ==Ø>
                ØØØ11Ø  111112          ZZ==ZZ←9+"J"

                                        IFN ZZ&777B8,<
                                                EXP ZZ
                                                ZZ==Ø>
                11Ø111  112113          ZZ==ZZ←9+"K"

                                        IFN ZZ&777B8,<
        ØØØØØ6' 11Ø111  112113                  EXP ZZ
                        ØØØØØØ                  ZZ==Ø>
                        ØØØ114          ZZ==ZZ←9+"L"

                                        IFN ZZ, DEFINE SHIFT
                                        <IFE ZZ&777B8,<
                                        ZZ==ZZ←9
                                        SHIFT>
                                        >
                                        SHIFT   ↑IFE ZZ&777B8,<
                        114ØØØ          ZZ==ZZ←9
                                        SHIFT   ↑IFE ZZ&777B8,<
                ØØØ114  ØØØØØØ          ZZ==ZZ←9
                                        SHIFT   ↑IFE ZZ&777B8,<
                114ØØØ  ØØØØØØ          ZZ==ZZ←9
                                        SHIFT   ↑IFE ZZ&777B8,<
                                        ZZ==ZZ←9
                                        SHIFT>
```

```
                                        ↑
                                        ↑
                                        ↑
                                        ↑
        ØØØØØ7' 114ØØØ  ØØØØØØ          EXP ZZ
                                        ↑
                                        ↑
                                        END
```

```
STORE TEXT CHARACTER BY CHARACTER     MACRO 47(113) 14:45 5-APR-72 PAGE 1-3
TEXT    MAC       5-APR-72 14:44      %I 5-APR-72

NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØ1Ø

2K CORE USED
```

Appendix A
Op Codes, Pseudo-Ops, and Monitor I/O Commands

This appendix contains a complete list of assembler defined operators including machine instruction mnemonic codes, assembler pseudo-ops, monitor programmed operators, and FORTRAN programmed operators. A programmed operator, or unimplemented user operation code is called a UUO.

A.1 ASSEMBLER PSEUDO-OPS AND MONITOR COMMANDS

The notes specify which pseudo-ops generate data, and which do not. Pseudo-ops that generate data may be used within literals, and in address operand fields.

The initial values given by MACRO-10 to I/O instructions and FORTRAN UUO's for which the octal op code is not shown, are given in the notes and are useful in checking listings.

ARRAY, pseudo-op, generates data
ARG, 320, no-op (same as JUMP)
ASCII, pseudo-op, generates data
ASCIZ, pseudo-op, generates data
ASUPPRESS, pseudo-op, no data generated
BLOCK, pseudo-op, no data generated
BYTE, pseudo-op, generates data
CALL, 040, monitor UUO
CALLI, 047, monitor UUO
CLOSE, 070, monitor UUO
COMMENT, no data generated
DATA, 020, FORTRAN UUO
DEC, pseudo-op, generates data
DEC., 033, FORTRAN UUO
DEFINE, pseudo-op, no data generated
DEPHASE, pseudo-op, no data generated

ENC., 034, FORTRAN UUO
END, pseudo-op , no data generated
ENTER, 077, monitor UUO
ENTRY, pseudo-op, no data generated
EXP, pseudo-op, generates data
EXTERN, pseudo-op, no data generated
FIN., 021, FORTRAN UUO
GETSTS, 062, monitor UUO
HISEG, pseudo-op, no data generated
IF1, conditional pseudo-op
IF2, conditional pseudo-op
IFB, conditional pseudo-op
IFDEF, conditional pseudo-op
IFDIF, conditional pseudo-op
IFE, conditional pseudo-op
IFG, conditional pseudo-op
IFGE, conditional pseudo-op
IFIDN, conditional pseudo-op
IFL, conditional pseudo-op
IFLE, conditional pseudo-op
IFN, conditional pseudo-op
IFNB, conditional pseudo-op
IFNDEF, conditional pseudo-op
IN, 056, monitor UUO
IN., 016, FORTRAN UUO
INBUF, 064, monitor UUO
IN., 026, FORTRAN UUO
INIT, 041, monitor UUO
INPUT, 066, monitor UUO
INTEGER, pseudo-op, generates data
INTERN, pseudo-op, no data generated
IOWD, pseudo-op, generates data
IRP, pseudo-op, no data generated
IRPC, pseudo-op, no data generated
LALL, pseudo-op, no data generated
LIST, pseudo-op, no data generated
LIT, pseudo-op, generates data
LOC, pseudo-op, no data generated
LOOKUP, 076, monitor UUO
MLOFF, pseudo-op, no data generated
MLON, pseudo-op, no data generated
MTAPE, 072, monitor UUO
MTOP., 024, FORTRAN UUO
NLI., 031, FORTRAN UUO
NLO., 032, FORTRAN UUO
NOSYM, pseudo-op, no data generated
OCT, pseudo-op, generates data
OPDEF, pseudo-op, no data generated
OPEN, 050, monitor UUO
OUT, 057, monitor UUO
OUT., 017, FORTRAN UUO
OUTBUF, 065, monitor UUO
OUTF., 027, FORTRAN UUO
OUTPUT, 067, monitor UUO
PAGE, pseudo-op, no data generated
PASS2, pseudo-op, no data generated
PHASE, pseudo-op, no data generated
POINT, pseudo-op, generates data
PRINTX, pseudo-op, no data generated
PURGE, pseudo-op, no data generated
RADIX, pseudo-op, no data generated
RADIX50, pseudo-op, generates data
RELEAS, 071, monitor UUO
RELOC, pseudo-op, no data generated
REMARK, pseudo-op, no data generated
RENAME, 055, monitor UUO
REPEAT, pseudo-op, no data generated
RERED., 030, FORTRAN UUO
RESET., 015, FORTRAN UUO
RIM, pseudo-op, no data generated
RIM10, pseudo-op, no data generated
RIM10B, pseudo-op, no data generated
RTB., 022, FORTRAN UUO
SEARCH, pseudo-op, no data generated
SETSTS, 060, monitor UUO
SIXBIT, pseudo-op, generates data
SLIST., 025, FORTRAN UUO
SQUOZE, same as RADIX50
STATO, 061, monitor UUO
STATUS, 062, monitor UUO
STATZ, 063, monitor UUO
STOPI, pseudo-op, no data generated
SUBTTL, pseudo-op, no data generated
SUPPRESS, pseudo-op, no data generated
SYN, pseudo-op, no data generated
TAPE, pseudo-op, no data generated
TITLE, pseudo-op, no data generated
TTCALL, 051, monitor UUO
TWOSEG, pseudo-op, no data generated
UGETF, 073, monitor UUO
UJEN, 100, monitor UUO
UNIVERSAL, pseudo-op, no data generated
USETI, 074, monitor UUO
USETO, 075, monitor UUO
VAR, pseudo-op, generates data
WTB., 023, FORTRAN UUO
XALL, pseudo-op, no data generated
XLIST, pseudo-op, no data generated
XWD, pseudo-op, generates data
Z, pseudo-op, generates data

.CREF, pseudo-op, no data generated
.XCREF, pseudo-op, no data generated
.HWFRMT, pseudo-op, no data generated
.MFRMT, pseudo-op, no data generated

A.2 MACHINE MNEMONICS AND OCTAL CODES

The following are machine mnemonics and corresponding octal codes:

ADD	270	CAMGE	315	FSBRI	155	HRREM	572	MOVEM	202	SETMI	415	TLCA	645
ADDB	273	CAML	311	FSBRM	156	HRRES	573	MOVES	203	SETMM	416	TLCE	643
ADDI	271	CAMLE	313	FSC	132	HRRI	541	MOVM	214	SETO	474	TLCN	647
ADDM	272	CAMN	316	HALT	254-4,	HRRM	542	MOVMI	215	SETOB	477	TLN	601
AND	404	CLEAR	400	HLL	500	HRRO	560	MOVMM	216	SETOI	475	TLNA	605
ANDB	407	CLEARB	403	HLLE	530	HRROI	561	MOVMS	217	SETOM	476	TLNE	603
ANDCA	410	CLEARI	401	HLLEI	531	HRROM	562	MOVN	210	SETZ	400	TLNN	607
ANDCAB	413	CLEARM	402	HLLEM	532	HRROS	563	MOVNI	211	SETZB	403	TLO	661
ANDCAI	411	CONI	7-24	HLLES	533	HRRS	543	MOVNM	212	SETZI	401	TLOA	665
ANDCAM	412	CONO	7-20	HLLI	501	HRRZ	550	MOVNS	213	SETZM	402	TLOE	663
ANDCB	440	CONSO	7-34	HLLM	502	HRRZI	551	MOVS	204	SKIP	330	TLON	667
ANDCBB	443	CONSZ	7-30	HLLO	520	HRRZM	552	MOVSI	205	SKIPA	334	TLZ	621
ANDCBI	441	DATAI	7-04	HLLOI	521	HRRZS	553	MOVSM	206	SKIPE	332	TLZA	625
ANDCBM	442	DATAO	7-14	HLLOM	522	IBP	133	MOVSS	207	SKIPG	337	TLZE	623
ANDCM	420	DFN	131	HLLOS	523	IDIV	230	MUL	224	SKIPGE	335	TLZN	627
ANDCMB	423	DIV	234	HLLS	503	IDIVB	233	MULB	227	SKIPL	331	TRC	640
ANDCMI	421	DIVB	237	HLLZ	510	IDIVI	231	MULI	225	SKIPLE	333	TRCA	644
ANDCMM	422	DIVI	235	HLLZI	511	IDIVM	232	MULM	226	SKIPN	336	TRCE	642
ANDI	405	DIVM	236	HLLZM	512	IDPB	136	OR	434	SOJ	360	TRCN	646
ANDM	406	DPB	137	HLLZS	513	ILDB	134	ORB	437	SOJA	364	TRN	600
AOBJN	253	EQV	444	HLR	544	IMUL	220	ORCA	454	SOJE	362	TRNA	604
AOBJP	252	EQVB	447	HLRE	574	IMULB	223	ORCAB	457	SOJG	367	TRNE	602
AOJ	340	EQVI	445	HLREI	575	IMULI	221	ORCAI	455	SOJGE	365	TRNN	606
AOJA	344	EQVM	446	HLREM	576	IMULM	222	ORCAM	456	SOJL	361	TRO	660
AOJE	342	EXCH	250	HLRES	577	IOR	434	ORCB	470	SOJLE	363	TROA	664
AOJG	347	FAD	140	HLRI	545	IORB	437	ORCBB	473	SOJN	366	TROE	662
AOJGE	345	FADB	143	HLRM	546	IORI	435	ORCBI	471	SOS	370	TRON	666
AOJL	341	FADL	141	HLRO	564	IORM	436	ORCBM	472	SOSA	374	TRZ	620
AOJLE	343	FADM	142	HLROI	565	JCRY	255-6,	ORCM	464	SOSE	372	TRZA	624
AOJN	346	FADR	144	HLROM	566	JCRY0	255-4,	ORCMB	467	SOSG	377	TRZE	622
AOS	350	FADRB	147	HLROS	567	JCRY1	255-2,	ORCMI	465	SOSGE	375	TRZN	626
AOSA	354	FADRI	145	HLRS	547	JEN	254-12,	ORCMM	466	SOSL	371	TSC	651
AOSE	352	FADRM	146	HLRZ	554	JFCL	255	ORI	435	SOSLE	373	TSCA	655
AOSG	357	FDV	170	HLRZI	555	JFFO	243	ORM	436	SOSN	376	TSCE	653
AOSGE	355	FDVB	173	HLRZM	556	JFOV	255-1,	POP	262	SUB	274	TSCN	657
AOSL	351	FDVL	171	HLRZS	557	JOV	255-10,	POPJ	263	SUBB	277	TSN	611
AOSLE	353	FDVM	172	HRL	504	JRA	267	PUSH	261	SUBI	275	TSNA	615
AOSN	356	FDVR	174	HRLE	534	JRST	254	PUSHJ	260	SUBM	276	TSNE	613
ASH	240	FDVRB	177	HRLEI	535	JRSTF	254-2,	ROT	241	TDC	650	TSNN	617
ASHC	244	FDVRI	175	HRLEM	536	JSA	266	ROTC	245	TDCA	654	TSO	671
BLKI	7-00	FDVRM	176	HRLES	537	JSP	265	RSW	7-04	TDCE	652	TSOA	675
BLKO	7-10	FMP	160	HRLI	505	JSR	264	SETA	424	TDCN	656	TSOE	673
BLT	251	FMPB	163	HRLM	506	JUMP	320	SETAB	427	TDN	610	TSON	677
CAI	300	FMPL	161	HRLO	524	JUMPA	324	SETAI	425	TDNA	614	TSZ	631
CAIA	304	FMPM	162	HRLOI	525	JUMPE	322	SETAM	426	TDNE	612	TSZA	635
CAIE	302	FMPR	164	HRLOM	526	JUMPG	327	SETCA	450	TDNN	616	TSZE	633
CAIG	307	FMPRB	167	HRLOS	527	JUMPGE	325	SETCAB	453	TDO	670	TSZN	637
CAIGE	305	FMPRI	165	HRLS	507	JUMPL	321	SETCAI	451	TDOA	674	UFA	130
CAIL	301	FMPRM	166	HRLZ	514	JUMPLE	323	SETCAM	452	TDOE	672	XCT	256
CAILE	303	FSB	150	HRLZI	515	JUMPN	326	SETCM	460	TDON	676	XOR	430
CAIN	306	FSBB	153	HRLZM	516	LDB	135	SETCMB	463	TDZ	630	XORB	433
CAM	310	FSBL	151	HRLZS	517	LSH	242	SETCMI	461	TDZA	634	XORI	431
CAMA	314	FSBM	152	HRR	540	LSHC	246	SETCMM	462	TDZE	632	XORM	432
CAME	312	FSBR	154	HRRE	570	MOVE	200	SETM	414	TDZN	636		
CAMG	317	FSBRB	157	HRREI	571	MOVEI	201	SETMB	417	TLC	641		

Appendix B
Summary of Pseudo-Ops

B.1 PSEUDO-OPS

A list of pseudo-ops and their functions follows:

ARRAY	Reserve multiple words of storage.
ASCII	Seven-bit ASCII test
ASCIZ	Seven-bit ASCII test, with null character guaranteed at end
ASUPPRESS	Turns on suppress bit for all symbols
BLOCK	Reserves block of storage cells
BYTE	Input bytes of length 1-36 bits
COMMENT	No binary produced; same as seven-bit ASCII
DEC	Input decimal numbers
DEFINE	Defines macro
DEPHASE	Terminates PHASE relocation mode
END	Last statement of the program
ENTRY	Entry point for subroutine library
EXP	Input expressions
EXTERN	Identifies external symbols

HISEG	Load into high segment
INTEGER	Reserve one word of storage per argument
INTERN	Define internal symbols
IOWD	Set up I/O transfer word
IRP	Indefinite repeat of macro arguments
IRPC	Indefinite repeat of one character
LALL	List all; expanded listing of macros
LIST	List in normal mode
LIT	Assemble literals
LOC	Assign absolute addresses
MLOFF	Turn off multiline literal feature
MLON	Turn on multiline literal feature
NOSYM	Suppress symbol table listing
OCT	Input octal numbers
OPDEF	Defines user-created operator; generates only one word
PAGE	Start a new listing page
PASS2	Terminates pass 1, remaining statement are processed pass 2 only
PHASE	Following coding relocated at execution time
POINT	Sets up byte pointer word
PRGEND	Allows multiprogram assemblies, end one such program
PRINTX	Output on terminal or listing device the rest of the line
PURGE	Remove symbol from table
RADIX	Sets prevailing radix to 2-10
RADIX5Ø	Compresses 36-bit words, primarily for system use
RELOC	Implied first statement; assigns relocatable addresses
REMARK	Comments only statement
REPEAT	Repeat n times
RIM	Prepare output in RIM paper-tape format
RIM1Ø	Absolute, unblocked, output format; no checksums
RIM1ØB	Absolute, blocked, checksummed output format

SALL	Suppress listing of macros; lists only call and binary generated
SEARCH	Opens symbol tables of universal program
SIXBIT	Input text in compressed 6-bit ASCII
SQUOZE	Same as RADIX 50 above
STOPI	Stop indefinite repeat of macro arguments
SUBTTL	Subtitle on listing
SUPPRESS	Turns on suppress bit for specified symbols
SYN	Make synonymous
TAPE	Stop processing the current file
TITLE	Title on listing and to DDT
TWOSEG	Assembles and loads two segment programs
UNIVERSAL	Makes symbol table available to other programs
VAR	Assemble variables suffixed with # or ARRAY or INTEGER
XALL	Stop expanded listing, resume normal list mode
XLIST	Stop listing
XPURGE	Purges local symbols on pass 2
XWD	Input two 18-bit half words
Z	Input zero word
.CREF	Resume outout pf CREF information
.XCREF	Stop output of CREF information
.HWFRMT	List binary in half word format (old)
.MFRMT	List binary in multi-format (new)

B.1.1 Conditional Assembly Statements

These conditional assembly statements in the first column are assembled if the conditions in the second column exist.

IF1	Encountered during pass 1
IF2	Encountered during pass 2
IFB	Blank
IFDEF	Defined
IFDIF	Different
IFE	Zero

IFG	Positive
IFGE	Zero, or positive
IFIDN	Identical
IFL	Negative
IFLE	Zero, or negative
IFN	Non-zero
IFNB	Not blank
IFNDEF	Not defined

Appendix C
Summary of Character Interpretations

The characters listed below have special meaning in the contexts indicated. These interpretations do not apply when these characters appear in text strings, or in comments.

Character	Meaning	Example
:	Colon. Immediately follows all labels.	LABEL: Z
;	Semicolon. Precedes all comments.	;THIS IS A COMMENT
.	Point. Has current value of the location counter or indicates floating point number.	JRST .+5 JUMP FORWARD FIVE LOCATIONS 1.Ø
,	Comma. General operand or argument delimiter.	DEC 1Ø,5,6 EXP A+B,C-D
	Accumulator field delimiter.	MOVEI 1,TAG
	References accumulator 0. The comma is optional.	MOVEI ,TAG
	Delimits macro arguments.	MACRO (A,B,C)
!	Inclusive OR } Logical Operators	
&	AND } Logical Operators	

Character	Meaning	Example
*	Multiplication	Arithmetic Operators (*, /, +, -)
/	Division	
+	Add (+A outputs the value of A)	
-	Subtract	
1st character of text string	In ASCII, ASCIZ and SIXBIT comment text strings, the first non-blank character is the delimiter.	ASCII/STRING/;
B	Follows number to be shifted and precedes binary shift count.	7B2
E	Exponent. Precedes decimal exponent in floating-point numbers.	F22.1E5 EXPONENT IS 5.
( )	Parentheses. Enclose index fields.	ADD AC1,X (7) MOVEI A,(SIXBIT/ABC/)
	Enclose the byte size in BYTE statements.	BYTE (6) 8, 8, 7
	Enclose the dummy argument string in macro DEFINE statements.	DEFINE MAC(A,B,C)
< >	Angle brackets. In an expression, enclose a numeric quantity.	<A-B+5ØØ/C>
	In conditional assembly statements, contain a single argument, and the conditional coding.	IF1, MOVE ACØ, TAX
	In REPEAT statements, contain coding to be repeated.	REPEAT 3, <SUB 17, TAG>
	In macros, enclose the macro definition.	DEFINE PUNCH DATAO PTP, PUNBUF (4)
[ ]	Square brackets. Delimit literals.	ADD 5,[MOVEI 3,TAX]
	In OPDEF statement, contain new operator; in ARRAY the size.	OPDEF CAL [MOVE] ARRAY FOO[212]
=	Equal sign. Direct assignment.	SYM=6 SYM-A+B*D
==	Equal sign. Direct assignment but no output to DDT.	SYM==6
=:	Equal sign and colon. Direct assignment but automatically made internal.	FLAG=:2ØØ
:!	Colon and exclamation point. Direct assignment of label, no output to DDT, and automatically made internal.	LABEL:!

Character	Meaning	Example
==:	Equal sign and colon. Direct assignment, no output to DDT, and automatically made internal.	LOOP==:32
::!	Double colon and exclamation point. Direct assignment of label, no output to DDT, and automatically made internal.	NAME::!
"..."	Quotation marks enclose 7-bit ASCII text, right justified, from one to five characters.	"ABCDE"
'...'	Single quotation marks enclose 6-bit ASCII text, right justified, from one to six characters.	'TABLES'
#	Number sign, Defines a symbol used as a tag. Variable.	ADD 3,TAG#
##	Alternate method of generating external symbols.	MOVE Ø,JOBREL##
'	Apostrophe or single quote. Concatenation character, used within macro definitions or SIXBIT data.	DEFINE MAC (A,B,C); <JUMP'A B, C> 'SIXBIT'
\	Reverse slash. If used as the first character of an argument in a macro call, the value of the following symbol is converted to an ASCII symbol in the current radix.	MAC \ A IF A=5ØØ, THIS GENERATES THREE 7-BIT ASCII CHARACTERS, ASCII/5ØØ/
↑←	Control left arrow. Line continuation.	
←	Left arrow. N M shift N left (or right) M bit positions,	1ØØ←3=1ØØØ 1ØØ←←3=1Ø
@	Indicates indirect addressing. Causes the indirect bit in an instruction to be set.	MOV AC,@ADDR

Appendix D
Storage Allocation

MACRO allocates storage in two directions:

1) the symbol table (user symbols and macro names) grows downward from top of the low segment (.JBREL)

2) Macros, literals, etc., grow upward from free space (.JBFF).

All entries in the symbol table are two words long. The first word is the symbol name in SIXBIT. The second word is flags in left half and either value or pointer in right half.

Most symbols have a value less than 18 bits and so can be represented by just the two words in the symbol table. Symbols with a 36-bit value (e.g., -1) have the value stored in a 1 word in free storage and a pointer to this value stored in the symbol table.

External symbols have two words in free storage, the first is the value (i.e., the last reference in a chain of references to the symbol). The second is the sixbit name of the symbol. This is so that additive global fixups can be output.

Opdefs tend to have 36-bit values and are stored like other 36-bit value symbols.

Macro names are stored in the symbol table, the value is a pointer to the stored text string.

The text string is stored in four (assembly parameter) word blocks which have the general form

```
1)   link to next block, [Ø if last] ,, 2 characters
2)                     5 characters
3)                     5 characters
4)                     5 characters
```

However, the first such block is special

```
1)   link to next block   ,,   link to last block
2)   pointer to default arg; ,, <number or args expected>+9+reference
3)                     5 characters                              count
4)                     5 characters
```

The number of args expected is the number of arguments in the define statement.

The reference count is incremented when the macro is called and decremented when exiting from the macro. When this count goes to zero the macro is removed from free space.

The actual arguments to a macro are stored in the same linked block, but are not in the symbol table. Repeats (2 or more times) are also stored the same way. The text blocks are removed when the macro exits or the repeat exits since the reference count has gone to zero.

The addresses of the actual argument blocks are stored in a pushdown stack in order of generation.

Default arguments are stored the same way except the list is in free core. The pointer to this default arg list is stored in the left half of the second word of the first block of the macro definition.

The text body is stored as is, except that dummy arguments are replaced by special symbols.

The ASCII character RUBOUT (177) is used to signal a special character text.

These characters are

```
ØØ1     ;end of macro
ØØ2     ;end of dummy symbol
ØØ3     ;end of Repeat
ØØ4     ;end of IRP or IRPC
```

If the character is 4<ch<77 it is illegal.
If the character is <100 then it is a dummy symbol, the value of the character is ANDed with 37 to get the dummy symbol number and the corresponding pointer retrieved from the stack of pointers.

If th- symbol was not specified (i.e., no pointer) then if the 40 bit is on this is to be a created symbol and one is created, otherwise the argument is ignored.

Verbose macros can eat up a lot of storage space.

Literals are stored in four words/block per word generated (three words if old format used).

Words are

```
-3:     form word
-2:     relocation bits
-1:     code
 Ø:     pointer to next
```

The pointer points to the Ø word of the next block. The code is the generated code. Relocation is either the relocation bits Ø or 1 per half word or external pointers if externs used.

Form word is the word used for listing, this word is not checked when comparing literals so that different forms that produce the same code are classed as equal.

Long literals are both slow and take up extra storage, they should be written as subroutines or inline.

Single quotes can also be used to indicate SIXBIT words, however, one pair of single quotes is removed by the assembler if the pair encloses a dummy argument. For example, in the macro

```
DEFINE    SXBT (A)<
MOVSI     1,"A"
MOVSI     2,"B"
>
```

B is not a dummy argument so it can be enclosed in single quotes. A, however, is a dummy argument and must be enclosed in double quotes since one pair of quotes (the inner pair) will be removed by the assembler.

Appendix E
Text Codes

This appendix contains a summary of MACRO-10 text codes.

SIXBIT	Character	ASCII 7-Bit*	SIXBIT	Character	ASCII 7-Bit*	Character	ASCII 7-Bit*
00	Space	040	40	@	100	`	140
01	!	041	41	A	101	a	141
02	"	042	42	B	102	b	142
03	#	043	43	C	103	c	143
04	$	044	44	D	104	d	144
05	%	045	45	E	105	e	145
06	&	046	46	F	106	f	146
07	'	047	47	G	107	g	147
10	(	050	50	H	110	h	150
11	)	051	51	I	111	i	151
12	*	052	52	J	112	j	152
13	+	053	53	K	113	k	153
14	,	054	54	L	114	l	154
15	-	055	55	M	115	m	155
16	.	056	56	N	116	n	156
17	/	057	57	O	117	o	157
20	0	060	60	P	120	p	160
21	1	061	61	Q	121	q	161
22	2	062	62	R	122	r	162
23	3	063	63	S	123	s	163
24	4	064	64	T	124	t	164
25	5	065	65	U	125	u	165
26	6	066	66	V	126	v	166
27	7	067	67	W	127	w	167
30	8	070	70	X	130	x	170
31	9	071	71	Y	131	y	171
32	:	072	72	Z	132	z	172
33	;	073	73	[	133	{	173
34	<	074	74	\	134	|	174
35	=	075	75	]	135	}	175
36	>	076	76	↑	136	~	176
37	?	077	77	←	137	Delete	177

*MACRO-10 also accepts five of the 32 control codes in 7-bit ASCII:

Horizontal Tab 011 — Line Feed 012 — Vertical Tab 013 — Form Feed 014 — Carriage Return 015

Appendix F
Radix 50 Representation

Radix 50_8 representation is used to condense 6-character symbols into 32 bits. Each character of a symbol is subscripted in descending order from left to right; i.e., the symbols are of the form

$$L_6 L_4 L_5 L_3 L_2 L_1$$

If C_n denotes the octal code for L_n, the radix 50_8 representation is generated by the following

$$(((((C_6*5Ø)+C_5)*5Ø+C_4)*5Ø*C_3)*5Ø+C_2)*5Ø+C_1$$

where all numbers are octal.

The code numbers corresponding to the characters are:

Code (Octal)	Characters
00	Null character
01-12	0-9
13-44	A-Z
45	.
46	$
47	%

The top four bits are taken from the four leftmost bits of a 6-bit octal number (i.e., Ø4-74).

Appendix G
Summary of Rules for Defining and Calling Macros

G.1 ASSEMBLER INTERPRETATION

MACRO-10 assembles macros by direct and immediate character substitutions. When a macro call is encountered, in any field, the character substitution is made, the characters are processed, and the assembler continues its scan with the character following the delimiter of the last argument, except when it is delimited by a semicolon. Macros can appear any number of times on a line.

G.2 CHARACTER HANDLING

G.2.1 Blanks

A macro symbol is delimited by one blank or one tab; the character following the delimiter is the start of the argument string even if it is also a blank or tab. Other than the first delimiter, blanks and tabs are treated as standard characters in the argument string.

G.2.2 Brackets

Angle brackets are only significant in the argument fields if the first character of any field is a left angle bracket. In this case,

no terminator or parenthesis tests are made between the left angle bracket and its matching right bracket. The matching brackets are removed from the string but the scan continues until a standard delimiter is found.

G.2.3 Parentheses

Parentheses serve only to terminate an argument scan. They are significant only when the first character following the blank or tab delimiter is a left parenthesis. In this case, the left parenthesis is removed and, if it matching right parenthesis is encountered prior to the normal termination of theargument scan, it is removed and the scan discontinued.

G.2.4 Commas

When a comma is encountered in an argument scan, it acts as the delimiter of the current argument. If it delimits the last argument, the character following it will be the first scanned after the subsitution is processed.

G.2.5 Semicolons

When a semicolon is encountered in an argument scan, the scan is discontinued. If an argument has not been satisfied, the remainder is considered null. It is saved, however, and will be the first character scanned after the substitution is made, normally acting as a comment flag.

G.2.6 Carriage Return

A carriage return, except when pre-empted by angle brackets (see Section G.2.2), will terminate the scan similar to the semicolon. This can be circumvented, if desired, by the control left arrow key described elsewhere.

G.2.7 Back-Slash

If the first character of any argument is a back-slash, it must be directly followed by a numeric term. The value of the numeric term is broken down into a string of ASCII digits of the current radix, just the reverse of a fixed-point number computation. The value is

considered to be a 36-bit positive number having a value of 0 to 777777 777777. Leading zeros are suppressed except in the case of 0, in which case the result is one ASCII 0. The ASCII string is substituted and the scan continued in the normal manner (no implied terminator).

The default listing mode is XALL, in which case the initial macro call and all lines within its range that produce binary code are listed. The pseudo-op LALL will cause all lines to be listed. Substituted arguments are bracketed by ↑'s by the assembler.

G.3 CONCATENATION

The rule for concatenation is as follows:

For each string of apostrophes, one is removed if and only if it is next to (either before or after) a dummy argument to that macro.

Appendix H
Operating Instructions

H.1 REQUIREMENTS

The following are MACRO-10 operating requirements:

Minimum Core	7K pure plus 1K impure
Additional Core	Automatically requests additional core assignments from the timesharing monitor as needed.
Equipment	One input device (source program input); up to two output devices (machine language program output and listing output). If the listing output is to be used as input to the Cross Reference (CREF) program, it must not be TTY, DIS or LPT.

H.2 INITIALIZATION

The following are commands and corresponding indications:

.R MACRO)	Loads the MACRO-10 Assembler into core.
*	The Assembler is ready to receive a command.

H.3 COMMANDS

H.3.1 General Command Format

MACRO-10 general commands are as follows:

```
objprog-dev:filename.ext,list-dev:filename.ext source-dev:filename.ext,......source-n)
```

objprog-dev: The device on which the object program is to be written.

MTAn:	(magnetic tape)
DTAn:	(DECtape)
PTP:	(paper-tape punch)
DSK:	(disk)

list-dev: The device on which the assembly listing is to be written.

MTAn:	(magnetic tape)	Must be one of these if input to CREF[1]
DTAn:	(DECtape)	
DSK:	(disk)	
LPT:	(line printer)	
TTY:	(Teletype)	
PTP:	(paper-tape punch)	

source-dev: The device(s) from which the source-program input to assembly is to be read.

MTAn:	(magnetic tape)
CDR:	(card reader)
DTAn:	(DECtape)
DSK:	(disk)
PTR:	(paper-tape reader)
TTY:	(Teletype)

If more than one file is to be assembled from a magnetic tape, card reader, or paper tape reader, dev: is followed by a comma for each file beyond the first.

Input via the Teletype is terminated by typing CTRL Z (↑Z) to enter pass 1; the entries must be retyped at the beginning of pass 2.

filename.ext (DSK: and DTAn: only) The filename and filename extension of the object program file, the listing file, and the source file(s).

The object program and listing devices are separated from the source device by the left arrow symbol.

H.3.2 Disk File Command Format

MACRO-10 disk file commands are as follows:

```
DSK:filename.ext [proj,prog]
```

[1] If /C switch is given, but no list-dev: is specified, DSK:CREF.CRF is assumed.

[proj,prog] Project-programmer number assigned to the disk area to be searched for the source file(s) if other than the user's project-programmer number.

The installation standard protection is assigned to any disk file specified as output.

NOTE

If object coding output is not desired (e.g., a program is being scanned for source language errors), objprog-dev: is omitted. If an assembly listing is not desired, list-dev: is omitted. If device is not specified, DSK is assumed.

Examples:

```
.R MACRO
*DTA3:OBJPRG,LPT: CDR:
```

Assemble one source program file from the card reader; write the object code on DTA3 and call the file OBJPRG; write the assembly listing on the line printer.

```
END OF PASS 1
```

The source program cards must be manually re-fed for pass 2.

```
?2 ERRORS DETECTED
PROGRAM BREAK IS ØØ2537
2K CORE USED
```

Number of source errors; size of object program; core used by assembler.

```
*↑C
```

Return to the monitor.

```
.R MACRO
*MTA3:,MTA2: MTA1:,,
```

Assemble the next three source files located at the present position of MTA1; write the object program on MTA3; write the listing on MTA2 for later printing.

```
NO ERRORS DETECTED
PROGRAM BREAK IS ØØ3552
2K CORE USED
```

```
*,LPT: DTA1:FILE1,FILE2,FILE5
NO ERRORS DETECTED
PROGRAM BREAK IS ØØ1Ø27
2K CORE USED
```

Assemble the source files named FILE1, FILE2, and FILE5 from DTA1; produce no object coding; write the listing on the line printer.

```
*,←DSK:FILE1.MAC[14,12]
NO ERRORS DETECTED
PROGRAM BREAK IS ØØØ544
2K CORE USED
```

Scan the source program called FILE1.MAC, located in area 14, 12 on the disk, for source language errors; produce no object coding or assembly listing; print all error diagnostics on the terminal.

```
*↑C
```

Return to the monitor.

```
.R MACRO
```

```
*MTA1:,TTY: TTY:

        JMP     R
R:      AOS     G
G:      JFCL
        END
```

Assemble a source file from the terminal; write the object code program on MTA1 and print the assembly listing on the terminal.

```
↑Z
```

Terminate input.

```
     END OF PASS 1
         JMP    R
```

Reenter terminal input.
Type first statement again.

```
.MAIN     MACRO          1Ø:14       2Ø-DEC-67       PAGE1)    Page heading.
0         ØØØØØØ  ØØØØØØ   ØØØØØ1'     JMP           R)        First assembled.
R:        AOS            G                                     Reenter second.
          ØØØØØ1  35ØØØØ   ØØØØØ2'   R: AOS          G)        Second assembled.
G:        JFCL)                                                Reenter third.
          ØØØØØ2  255ØØØ   ØØØØØØ    G: JFCL)                  Third assembled.
          END)                                                 Reenter fourth.
                                       END)                    Fourth assembled.

?1 ERROR DETECTED)                                             Typeout of symbol
PROGRAM BREAK IS ØØØØØ3)                                       table.
.MAIN     MACRO          1Ø:14       2Ø-DEC-67       PAGE2)

          SYMBOL TABLE)
G         ØØØØØ2')
R         ØØØØØ1')

2K CORE USED)
*↑C)                                                           Return to the monitor.
```

H.4 SWITCHES

Switches are used to specify such options as:

a. Magnetic tape control
b. Macro call expansion
c. Listing suppression
d. Pushdown list expansion
e. Cross-reference file output.

All switches are preceded by a slash (/) or enclosed in parentheses, and usually occur prior to the left arrow (see Table H-1).

Table H-1

MACRO-10 Switch Options

Switch	Meaning
A	Advance magnetic tape reel by one file.
B	Backspace magnetic tape reel by one file.
C	Produce listing file in a format acceptable as input to CREF; unless the file is named, CREF. CRF is assigned as the filename; if no extension is given, .CRF is assigned; if no list-dev: is specified, DSK: is assumed. /C must appear between the comma and the left-arrow.
E	List macro expansions (same function as LALL pseudo-op).
F	New format for output binary listing (.MFRMT pseudo-op).
G	Old format for output binary listing (.HWFRMT pseudo-op).
H	Print Help text (i.e., this list of switches and explanations).
L	Reinstate listing (used after list suppression by either the XLIST pseudo-op or 5 switch).
M	List only call, no binary, in macro expansion (same .SALL pseudo-op).
N	Suppress error printouts on the terminal.
O	Sets the pseudo-op MLOFF which allows literals to occupy on a single line. This means literals may be terminated with a carriage return, line feed instead of a right bracket.
P	Increase the size of the pushdown list. This switch may appear as many times as desired (pushdown list is initially set to a size of 80_{10} locations; each /P increases its size by 80_{10}). /P must appear on the left of the left arrow.
Q	Suppress Q (questionable) error indications on the listing; Q messages indicate assumptions made during pass 1. /Q must appear on the left of the left-arrow.
S	Suppress listing (same function as XLIST pseudo-op).
T	Skip to the logical end of the magnetic tape.
W	Rewind the magnetic tape.
X	Suppress all macro expansions (same function as XALL pseudo-op).
Z	Zero the DECtape directory.

NOTE

Switches A through C and T, W, X, and Z must immediately follow the device or file to which the individual switch refers.

Examples:

```
.R MACRO)
*MTA1:,DTA3:,/C←PTR:)
```

Assemble one source file from the paper tape reader; write the object code on MTA1; write the assembly listing on DTA3 in cross-reference format and call the file CREF.CRF.

```
END OF PASS 1 )
```

The paper tape must be re-fed by the operator for pass 2.

```
?3 ERRORS DETECTED)
PROGRAM BREAK IS ØØØ4Ø1)
2K CORE USED)
```

End-of-assembly messages.

```
*DTA2:ASSEMB.ONE/Z,LPT:
        MTA4:/W,)
```

Rewind MTA4 and assemble the first two source files on it; write the object code on DTA2, after zeroing the directory, and call the file ASSEM.ONE; write the assembly listing on the line printer.

```
NO ERRORS DETECTED)
PROGRAM BREAK IS ØØ5231)
3K CORE USED)
```

```
*MTA1:/W,LPT:←MTA3:
     /W,(AA),(BB))
```

Rewind MTA1 and MTA3 and assemble files 1, 4, and 3 (in that order) from MTA3; print the assembly listing on the line printer; write the object code on MTA1.

```
?1 ERROR DETECTED)
PROGRAM BREAK IS ØØØ655)
2K CORE USED)
```

```
*FOO,/C FOO)
NO ERRORS DETECTED)
PROGRAM BREAK IS ØØØ765)
2K CORE USED)
```

Assemble source file FOO on DSK:; write the assembly listing on DSK in cross-reference format calling the file CREF.CRF. Write object code on DSK calling it FOO.REL.

```
*↑C)
.
```

Return to the monitor.

DEC-10-MRRC-D

decsystem10 MONITOR CALLS

This manual reflects the software as of the 5.06 release of the monitor.

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Printing June 1971
2nd Printing (Rev) January 1972
3rd Printing (Rev) June 1972
4th Printing (Rev) March 1973

The material in this manual is for informational purposes and is subject to change without notice.

Copyright © 1971, 1972, 1973 by Digital Equipment Corporation

Actual distribution of the software described in this specification will be subject to terms and conditions to be announced at some future date by Digital Equipment Corporation.

DEC assumes no responsibility for the use or reliability of its software on equipment which is not supplied by DEC.

The software described in this manual is furnished to purchaser under a license for use on a single computer system and can be copied (with inclusion of DEC's copyright notice) only for use in such system, except as may otherwise be provided in writing by DEC.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

DEC	PDP
FLIP CHIP	FOCAL
DIGITAL	COMPUTER LAB

CONTENTS

			Page
CHAPTER 1 MEMORY FORMAT			
1.1	Memory Protection and Relocation		359
1.2	User's Core Storage		360
	1.2.1	Job Data Area (JOBDAT)	360
	1.2.2	Vestigial Job Data Area	365
CHAPTER 2 INTRODUCTION TO USER PROGRAMMING			
2.1	Processor Modes		367
	2.1.1	User Mode	367
	2.1.2	User I/O Mode	367
	2.1.3	Executive Mode	368
2.2	Programmed Operators (UUOs)		368
	2.2.1	Operation Codes 001-037 (User UUOs)	369
	2.2.2	Operation Codes 040-077 and 000 (Monitor UUOs)	369
		CALL and CALLI	371
		Suppression of Logical Device Names	382
		Restriction on Monitor UUOs in Reentrant User Programs	382
	2.2.3	Operation Codes 100-127 (Unimplemented Op Codes)	383
	2.2.4	Illegal Operation Codes	383
	2.2.5	Naming Conventions for Monitor Symbols	383
CHAPTER 3 NON-I/O UUOS			
3.1	Execution Control		385
	3.1.1	Starting	385

CONTENTS (Cont)

			Page
		SETDDT AC, or CALLI AC, 2	385
	3.1.2	Stopping	385
		Illegal Instructions (700-777, JRST 10, JRST 14) and Unimplemented OP Codes (101-127)	385
		HALT or JRST 4	386
		EXIT AC, or CALLI AC, 12	386
	3.1.3	Trapping	387
		APRENB AC, or CALLI AC, 16	387
		Error Intercepting	388
	3.1.4	Suspending	391
		SLEEP AC, or CALLI AC, 31	391
		HIBER AC, or CALLI AC, 72	391
		WAKE AC, or CALLI AC, 73	392
3.2	Core Control		393
	3.2.1	Definitions	393
	3.2.2	LOCK AC, or CALLI AC, 60	394
		KA10 Systems	396
		Core Allocation Resource	397
		UNLOK. AC, or CALLI AC, 120	397
	3.2.3	CORE AC, or CALLI AC, 11	401
	3.2.4	SETUWP AC, or CALLI AC, 36	403
3.3	Segment Control		403
	3.3.1	RUN AC, or CALLI AC, 35	403
	3.3.2	GETSEG AC, or CALLI AC, 40	407
	3.3.3	REMAP AC, or CALLI AC, 37	408
	3.3.4	Testing for Sharable High Segments	408
	3.3.5	Modifying Shared Segments and Meddling	409
3.4	Program and Profile Identification		410
	3.4.1	SETNAM AC, or CALLI AC, 43	410
	3.4.2	SETUUO AC, or CALLI AC, 75	410
	3.4.3	LOCATE AC, or CALLI AC, 62	412
3.5	Inter-Program Communication		412
	3.5.1	TMPCOR AC, or CALLI AC, 44	412
		CODE = 0 (.TCRFS), Obtain Free Space	413

CONTENTS (Cont)

			Page
		CODE = 1 (.TCRRF), Read File	413
		CODE = 2 (.TCRDF), Read and Delete File	413
		CODE = 3 (.TCRWF), Write File	414
		CODE = 4 (.TCRRD), Read Directory	414
		CODE = 5 (.TCRDD), Read and Clear Directory	414
3.6	Environmental Information		414
	3.6.1	Timing Information	414
		DATE AC, or CALLI AC, 14	415
		TIMER AC, or CALLI AC, 22	415
		MSTIME AC, or CALLI AC, 23	415
	3.6.2	Job Status Information	416
		RUNTIM AC, or CALLI AC, 27	416
		PJOB AC, or CALLI AC, 30	416
		GETPPN AC, or CALLI AC, 24	416
		OTHUSR AC, or CALLI AC, 77	416
	3.6.3	Monitor Examination	416
		PEEK AC, or CALLI AC, 33	416
		SPY AC, or CALLI AC, 42	416
		POKE. AC, or CALLI AC, 114	417
		GETTAB AC, or CALLI AC, 41	417
	3.6.4	Configuration Information	442
		SWITCH AC, or CALLI AC, 20	442
		LIGHTS AC, or CALLI AC, -1	442
3.7	DAEMON AC, or CALLI AC, 102		442
	3.7.1	.DCORE Function	442
	3.7.2	.CLOCK Function	443
	3.7.3	Returns	443
3.8	Real-Time Programming		444
	3.8.1	RTTRP AC, or CALLI AC, 57	444
		Data Block Mnemonics	446
		Interrupt Level Use of RTTRP	447
		RTTRP Returns	447
		Restrictions	448
		Removing Devices from a PI Channel	449

CONTENTS (Cont)

			Page
		Dismissing the Interrupt	449
		Examples	449
	3.8.2	RTTRP Executive Mode Trapping	453
		Example	454
	3.8.3	TRPSET AC, or CALLI AC, 25	455
	3.8.4	UJEN (Op Code 100)	457
	3.8.5	HPQ AC, or CALLI AC, 71	457
3.9	METER. AC, or CALLI AC, 111		458
CHAPTER 4	I/O PROGRAMMING		
4.1	I/O Organization		461
	4.1.1	Files	461
	4.1.2	Job I/O Initialization	461
4.2	Device Selection		462
	4.2.1	Nondirectory Devices	462
	4.2.2	Directory Device	463
	4.2.3	Device Initialization	463
		Data Channel	463
		Device Name	463
		Initial File Status	464
		Data Modes	464
		Buffer Header	465
4.3	Ring Buffers		466
	4.3.1	Buffer Structure	466
		Buffer Ring Header Block	466
		Buffer Ring	467
	4.3.2	Buffer Initialization	468
		Monitor Generated Buffers	468
		User Generated Buffers	469
4.4	File Selection (LOOKUP and ENTER)		470
	4.4.1	The LOOKUP Operator	470
	4.4.2	The ENTER Operator	471
	4.4.3	RENAME Operator	472

CONTENTS (Cont)

			Page
4.5	Data Transmission		474
	4.5.1	Unbuffered Data Modes	475
	4.5.2	Buffered Data Modes	476
		Input	476
		Output	477
	4.5.3	Synchronization of Buffered I/O	478
4.6	Status Checking and Setting		479
	4.6.1	File Status Checking	480
	4.6.2	File Status Setting	480
4.7	File Termination		481
	4.7.1	CLOSE D,0	481
	4.7.2	CLOSE D,1 (Bit 35=1, CL.OUT)	482
	4.7.3	CLOSE D,2 (Bit 43=1, CL.IN)	482
	4.7.4	CLOSE D,4 (Bit 33=1, CL.DLL)	482
	4.7.5	CLOSE D,10 (Bit 32=1, CL.ACS)	482
	4.7.6	CLOSE D,20 (Bit 31=1, CL.NMB)	482
	4.7.7	CLOSE D,40 (Bit 30=1, CL.RST)	482
	4.7.8	CLOSE D,100 (Bit 29=1, CL.DAT)	482
4.8	Device Termination and Reassignment		483
	4.8.1	RELEASE	483
	4.8.2	RESDV. AC, or CALLI AC, 117	483
	4.8.3	REASSIGN AC, or CALLI AC, 21	484
	4.8.4	DEVLNM AC, or CALLI AC, 107	484
4.9	Examples		485
	4.9.1	File Reading	485
	4.9.2	File Writing	485
	4.9.3	File Reading/Writing	486
4.10	Device Information		487
	4.10.1	DEVSTS AC, or CALLI AC, 54	487
	4.10.2	DEVCHR AC, or CALLI AC, 4	488
	4.10.3	DEVTYP AC, or CALLI AC, 53	489
	4.10.4	DEVSIZ AC, or CALLI AC, 101	490
	4.10.5	WHERE AC, or CALLI AC, 63	491
	4.10.6	DEVNAM AC, or CALLI AC, 64	491

CONTENTS (Cont)

			Page
CHAPTER 5 I/O PROGRAMMING FOR NONDIRECTORY DEVICES			
5.1	Card Punch		494
	5.1.1	Concepts	494
	5.1.2	Data Modes	494
		ASCII, Octal Code 0	494
		ASCII Line, Octal Code 1	495
		Image, Octal Code 10	495
		Image Binary, Octal Code 13	495
		Binary, Octal Code 14	495
	5.1.3	Special Programmed Operator Service	495
	5.1.4	File Status	496
5.2	Card Reader		496
	5.2.1	Concepts	496
	5.2.2	Data Modes	497
		ASCII, Octal Code 0	497
		ASCII Line, Octal Code 1	497
		Image, Octal Code 10	497
		Image Binary, Octal Code 13	497
		Binary, Octal Code 14	497
		Super-Image, Octal Code 110	497
	5.2.3	Special Programmed Operator Service	498
	5.2.4	File Status	498
5.3	Display with Light Pen		499
	5.3.1	Data Modes	499
	5.3.2	Background	499
	5.3.3	Display UUOs	499
		INPUT D, ADR	499
		OUTPUT D, ADR	499
	5.3.4	File Status	501
5.4	Line Printer		502
	5.4.1	Data Modes	502
		ASCII, Octal Code 0	502
		ASCII Line, Octal Code 1	502
		Image, Octal Code 10	502

CONTENTS (Cont)

			Page
	5.4.2	Special Programmed Operator Service	502
	5.4.3	File Status	502
5.5	Magnetic Tape		503
	5.5.1	Data Modes	503
		ASCII, Octal Code 0	503
		ASCII Line, Octal Code 1	503
		Image, Octal Code 10	503
		Image Binary, Octal Code 13	503
		Binary, Octal Code 14	503
		DR (Dump Records), Octal Code 16	503
		D (Dump), Octal Code 17	504
	5.5.2	Magnetic Tape Format	504
	5.5.3	Special Programmed Operator Service	504
		MTAPE UUO	505
		MTCHR. AC, or CALLI AC, 112	507
	5.5.4	9-Channel Magtape	507
		Digital-Compatible Mode	508
		Industry-Compatible Mode	508
		Changing Modes	509
	5.5.5	File Status	511
5.6	Paper-Tape Punch		512
	5.6.1	Data Modes	512
		ASCII, Octal Code 0	512
		ASCII Line, Octal Code 1	512
		Image, Octal Code 10	512
		Image Binary, Octal Code 13	512
		Binary, Octal Code 14	512
	5.6.2	Special Programmed Operator Service	512
	5.6.3	File Status	513
5.7	Paper-Tape Reader		513
	5.7.1	Data Modes (Input Only)	513
		ASCII, Octal Code 0	513
		ASCII Line, Octal Code 1	513
		Image, Octal Code 10	513

CONTENTS (Cont)

			Page
		Image Binary, Octal Code 13	514
		Binary, Octal Code 14	514
	5.7.2	Special Programmed Operator Service	514
	5.7.3	File Status	514
5.8	Plotter		515
	5.8.1	Data Modes	515
		ASCII, Octal Code 0	515
		ASCII Line, Octal Code 1	515
		IMAGE, Octal Code 10	515
		IMAGE BINARY, Octal Code 13	515
		BINARY, Octal Code 14	515
	5.8.2	Special Programmed Operator Service	515
	5.8.3	File Status	516
5.9	Pseudo-TTY		516
	5.9.1	Concepts	516
	5.9.2	The HIBER UUO	517
	5.9.3	File Status	518
	5.9.4	Special Programmed Operator Service	519
		OUT, OUTPUT UUOs	519
		IN, INPUT UUOs	519
		RELEASE UUO	519
		JOBSTS UUO	519
		CTLJOB UUO	520
5.10	Terminals		521
	5.10.1	Data Modes	522
		ASCII, Octal Code 0 and ASCII Line, Octal Code 1	522
		Image, Octal Code 10	523
	5.10.2	DDT Submode	524
	5.10.3	Special Programmed Operator Service	525
		INCHRW ADR or TTCALL 0, ADR	526
		OUTCHR ADR or TTCALL 1, ADR	526
		INCHRS ADR or TTCALL 2, ADR	527
		OUTSTR ADR or TTCALL 3, ADR	527
		INCHWL ADR or TTCALL 4, ADR	527

CONTENTS (Cont)

		Page
	INCHSL ADR or TTCALL 5, ADR	527
	GETLCH ADR or TTCALL 6, ADR	527
	SETLCH ADR or TTCALL 7, ADR	528
	RESCAN ADR or TTCALL 10, 0	528
	CLRBFI ADR or TTCALL 11, 0	528
	CLRBFO ADR or TTCALL 12, 0	529
	SKPINC ADR or TTCALL 13, 0	529
	SKPINL ADR or TTCALL 14, 0	529
	IONEOU ADR or TTCALL 15, E	529
5.10.4	GETLIN AC, or CALLI AC, 34	529
5.10.5	TRMNO. AC, or CALLI AC, 115	529
5.10.6	TRMOP. AC, or CALLI AC, 116	530
5.10.7	File Status	533
5.10.8	Paper-Tape Input from the Terminal (Full-Duplex Software)	534
5.10.9	Paper-Tape Output at the Terminal (Full-Duplex Software)	534

CHAPTER 6 I/O PROGRAMMING FOR DIRECTORY DEVICES

6.1	DECtape		536
	6.1.1	Data Modes	536
		Buffered Data Modes	536
		Unbuffered Data Modes	536
	6.1.2	DECtape Format	536
	6.1.3	DECtape Directory Format	537
	6.1.4	DECtape File Format	539
		Block Allocation	540
	6.1.5	I/O Programming	540
		LOOKUP D, E	541
		ENTER D, E	542
		RENAME D, E	543
		INPUT, OUTPUT, CLOSE, RELEASE	544
	6.1.6	Special Programmed Operator Service	545
		USETI D, E	545

CONTENTS (Cont)

			Page
		USETO D, E	545
		UGETF D, E	545
		UTPCLR AC, or CALLI AC, 13	546
		MTAPE D, 1 and MTAPE D, 11	546
		DEVSTS UUO	546
	6.1.7	File Status	546
	6.1.8	Important Considerations	548
6.2	Disk		549
	6.2.1	Data Modes	549
		Buffered Data Modes	549
		Unbuffered Data Modes	549
	6.2.2	Structure of Disk Files	549
		Addressing by Monitor	550
		Storage Allocation Table (SAT) Blocks	550
		File Directories	551
		File Format	554
	6.2.3	Access Protection	554
		UFD and SFD Privileges	556
	6.2.4	Disk Quotas	559
	6.2.5	Simultaneous Access	560
	6.2.6	File Structure Names	560
		Logical Unit Names	560
		Physical Controller Class Names	560
		Physical Controller Names	560
		Physical Unit Names	560
		Unit Selection on Output	561
		Abbreviations	561
	6.2.7	Job Search List	562
	6.2.8	User Programming	563
		Four-Word Arguments for LOOKUP, ENTER, RENAME UUOs	564
		Extended Arguments for LOOKUP, ENTER, RENAME UUOs	571
		Error Recovery for ENTER and RENAME UUOs	576
	6.2.9	Special Programmed Operator Service	577
		PATH. AC, or CALLI AC, 110	577

CONTENTS (Cont)

			Page
		USETI and USETO UUOs	584
		SEEK UUO	587
		RESET UUO	587
		DEVSTS UUO	588
		CHKACC UUO	588
		STRUUO AC, or CALLI AC, 50	588
		JOBSTR AC, or CALLI AC, 47	590
		GOBSTR AC, or CALLI AC, 66	591
		SYSSTR AC, or CALLI AC, 46	592
		SYSPHY AC, or CALLI AC, 51	593
		DEVPPN AC, or CALLI AC, 55	593
		DSKCHR AC, or CALLI AC, 45	597
		DISK. AC, or CALLI AC, 121	599
		Simultaneous Supersede and Update	600
	6.2.10	File Status	601
	6.2.11	Disk Packs	602
		Removable File Structures	603
		Identification	603
		IBM Disk Pack Compatibility	603
6.3	Spooling of Unit Record I/O on Disk		603
	6.3.1	Input Spooling	604
	6.3.2	Output Spooling	604

CHAPTER 7 MONITOR ALGORITHMS

7.1	Job Scheduling		605
7.2	Program Swapping		607
7.3	Device Optimization		609
	7.3.1	Concepts	609
	7.3.2	Queuing Strategy	610
		Position-Done Interrupt Optimization	611
		Transfer-Done Interrupt Optimization	611
	7.3.3	Fairness Considerations	611
	7.3.4	Channel Command Chaining	611
		Buffered Mode	611

CONTENTS (Cont)

		Page
	Unbuffered Mode	612
7.4	Monitor Error Handling	612
7.4.1	Hardware Detected Errors	612
7.4.2	Software Detected Errors	613
7.5	Directories	613
7.5.1	Order of Filenames	613
7.5.2	Directory Searches	613
7.6	Priority Interrupt Routines	613
7.6.1	Channel Interrupt Routines	613
7.6.2	Interrupt Chains	614
7.7	Memory Parity Error Analysis, Reporting and Recovery	618

APPENDIX A DECTAPE COMPATIBILITY BETWEEN DEC COMPUTERS

APPENDIX B WRITING REENTRANT USER PROGRAMS

B.1	Defining Variables and Arrays	623
B.2	Example of Two-Segment Reentrant Program	623
B.3	Constant Data	624
B.4	Single Source File	624

APPENDIX C CARD CODES

APPENDIX D DEVICE STATUS BITS

APPENDIX E ERROR CODES

APPENDIX F COMPARISON OF DISK-LIKE DEVICES

APPENDIX G MAGNETIC TAPE CODES

APPENDIX H FILE RETRIEVAL POINTERS

H.1	A Group Pointer	641
H.1.1	Folded Checksum Algorithm	642

CONTENTS (Cont)

		Page
H.2	End-of-File Pointer	642
H.3	Change of Unit Pointer	642
H.4	Device Data Block	642
H.5	Access Block	642

ILLUSTRATIONS

Figure No.	Title	Page
1-1	KA10 User Address Mapping	361
1-2	KI10 User Address Mapping	361
3-1	Locking Jobs In Core on KA10 Systems	398
4-1	User's Ring of Buffers	467
4-2	Detailed Diagram of Individual Buffer	468
5-1	Pseudo-TTY	517
6-1	DECtape Directory Format	537
6-2	Format of a File on Tape	539
6-3	Format of a DECtape Block	540
6-4	Basic Disk File Organization for Each File Structure	551
6-5	Disk File Organization	553
6-6	Directory Paths on a Single File Structure	583
6-7	Directory Paths on Multiple File Structures	583

TABLES

Table No.	Title	Page
1-1	Job Data Area Locations (for user-program reference)	362
1-2	Vestigial Job Data Area Locations	365
2-1	Monitor Programmed Operators	369
2-2	CALL and CALLI Monitor Operations	372
3-1	GETTAB Tables	419
4-1	Buffered Data Modes	465
4-2	Unbuffered Data Modes	465
4-3	File Status Bits	479
5-1	Nondirectory Devices	493
5-2	MTAPE Functions	505
6-1	Directory Devices	535
6-2	LOOKUP Parameters	541
6-3	ENTER Parameters	542

TABLES (Cont)

Table No.	Title	Page
6-4	RENAME Parameters	543
6-5	File Structure Names	562
6-6	Extended LOOKUP, ENTER, and RENAME Arguments	571
6-7	.FSSRC Error Codes	590
7-1	Software States	610
C-1	ASCII Card Codes	629
C-2	DEC-029 Card Codes	629
C-3	DEC-026 Card Codes	630
D-1	Device Status Bits	631
E-1	Error Codes	635
F-1	Disk Devices	637
G-1	ASCII Codes and BCD Equivalents	639

ALPHABETICAL LIST OF MONITOR CALLS

ACTIVATE, 378
APRENB, 387
ATTACH, 379

CHGPPN, 378
CHKACC, 588
CLOSE, 481
CORE, 401
CTLJOB, 520

DAEFIN, 379
DAEMON, 442
DATE, 415
DDTGT, 372
DDTIN, 525
DDTOUT, 525
DDTRL, 372
DEACTIVATE, 378
DEVCHR, 488
DEVGEN, 378
DEVLNM, 484
DEVNAM, 491
DEVPPN, 593
DEVSIZ, 490
DEVSTS, 487
DEVTYP, 489
DISK., 599
DSKCHR, 597
DVRST., 381
DVURS., 381

ENTER, 471
EXIT, 386

FRCUUO, 379
FRECHN, 377

GETCHR, 372
GETLIN, 529
GETPPN, 416
GETSEG, 407
GETSTS, 480
GETTAB, 417
GOBSTR, 591

HIBER, 391
HPQ, 457

IN, 474
INBUF, 468
INIT, 463
INPUT, 474

JBSET., 380
JOBPEK, 379
JOBSTR, 590
JOBSTS, 519

LIGHTS, 442
LOCATE, 412
LOCK, 394
LOGIN, 373
LOGOUT, 373
LOOKUP, 470

METER., 458
MSTIME, 415
MTAPE, 505
MTCHR., 507

OPEN, 463
OTHUSR, 416
OUT, 474
OUTBUF, 468
OUTPUT, 474

PATH., 577
PEEK, 416
PJOB, 416
POKE., 417

REASSIGN, 484
RELEASE, 483
REMAP, 408
RENAME, 472
RESDV., 483
RESET, 461, 587
RTTRP, 444
RUN, 403
RUNTIM, 416

SEEK, 587
SETDDT, 385
SETNAM, 410
SETPOV, 374
SETSTS, 480
SETUUO, 410
SETUWP, 403
SLEEP, 391
SPY, 416
STATO, 480
STATZ, 480
STRUUO, 588
SWITCH, 442
SYSPHY, 593
SYSSTR, 592

TIMER, 415
TMPCOR, 412
TRMNO., 529
TRMOP., 530
TRPJEN, 374
TRPSET, 455
TTCALL, 525

UGETF, 545
UJEN, 457
UNLOK., 397
USETI, 545, 584
USETO, 545, 584
UTPCLR, 546

WAIT, 478
WAKE, 392
WHERE, 491

FOREWORD

DECsystem-10 Monitor Calls is a complete reference document describing the monitor programmed operators (UUOs) and is intended for the experienced assembly language programmer. The information presented in this manual reflects the 5.06 release of the monitor. The monitor calls are grouped in a manner that facilitates easy learning, and once they are mastered, the user can refer to the end of the Table of Contents and to the index for an alphabetical list of the UUOs.

DECsystem-10 Monitor Calls does not include reference material on the operating system commands. This information can be found in DECsystem-10 Operating System Commands (DEC-10-MRDC-D). Included in DECsystem-10 Operating System Commands are discussions on commands processed by both the monitor command language interpreter and the programs in the Batch system. The two manuals, DECsystem-10 Monitor Calls and DECsystem-10 Operating System Commands, supersede the Time-sharing Monitors manual (DEC-T9-MTZD-D) and all of its updates.

A third manual, Introduction to DECsystem-10 Software (DEC-10-MZDA-D), is a general overview of the DECsystem-10. It is written for the person, not necessarily a programmer, who knows computers and computing concepts and who desires to know the relationship between the various components of the DECsystem-10. This manual is not intended to be a programmer's reference manual, and therefore, it is recommended that it be read at least once before reading the above-mentioned reference documents.

SYNOPSIS OF DECsystem-10 MONITOR CALLS

Chapter 1 discusses the format of memory and briefly describes the job data area. Chapter 2 introduces all of the monitor programmed operators available to a user program and the various processor modes in which a user program operates. The UUOs available for non-I/O operations are presented in Chapter 3. These programmed operators are used to obtain execution, core, and segment control; program identification; environmental information; and real-time status. An introduction to I/O programming is presented in Chapter 4; the services the monitor performs for the user and how the user

program obtains these services are also discussed. I/O programming specific to the nondirectory devices and directory devices is explained in Chapters 5 and 6, respectively. Algorithms of the monitor, described in Chapter 7, give the user an insight into system operation. The appendices contain supplementary reference material and tables.

CONVENTIONS USED IN DECsystem-10 MONITOR CALLS

The following conventions have been used throughout this manual:

dev:	Any logical or physical device name. The colon must be included when a device is used as part of a file specification.
list	A single file specification or a string of file specifications. A file specification consists of a filename (with or without a filename extension), a device name, a directory name, and a protection.
job n	A job number assigned by the system.
file.ext	Any legal filename and filename extension.
core	Decimal number of 1K blocks of core (KA10). Decimal number of pages of core (KI10).
adr	An octal address.
C(adr)	The contents of an octal address.
[proj,prog]	Project-programmer numbers; the square brackets must be included in the command string.
fs	Any legal file structure name or abbreviation.
($)	The symbol used to indicate the ESCAPE Key.
↑x	A control character obtained by depressing the CTRL key and then the character key x.
←	A back arrow used in command string to separate the input and output file specifications.
=	A equal sign used in a command string to separate the input and output file specifications.
*	The system program response to a command string.
.	The monitor response to a command string.
↲	The symbol used to indicate that the user should depress the RETURN key. This key must be used to terminate every command to the monitor command language interpreter.
____	Underscoring used to indicate computer typeout.
n	A decimal number.
[directory]	A designation identifying a particular disk area. This designation can be in the form [proj, prog] which identifies a UFD or [proj,prog,sfd, sfd,...] which identifies a sub-file directory path branching from a UFD. The square brackets are required.

CHAPTER 1
MEMORY FORMAT

1.1 MEMORY PROTECTION AND RELOCATION

Each user program is run with the processor in a special mode called the user mode; in this mode the program must operate within an assigned area in core, and certain operations are illegal. Because every user has an assigned area in core, the rest of core is unavailable to him. He cannot gain access to a protected area for either storage or retrieval of information.

The assigned area of each user can be divided into two segments. If this is the case, the low segment (impure segment) is unique for a given user and can be used for any purpose. The high segment (pure segment) can be used by one user or it can be shared by many users. If the high segment is shared, the program is a reentrant program. The monitor usually write-protects the high segment so that the user cannot alter its contents. This is done, for example, when the high segment is a pure procedure to be used reentrantly by many users. One high pure segment can be used with any number of low impure segments. Any user program that attempts to write in a write-protected high segment is aborted and receives an error message. If the monitor defines two segments but does not write-protect the high segment, the user has a two-segment non-reentrant program (refer to Paragraph 3.2.4).

The DECsystem-10 monitor defines the size and position of a user's area. On KA10-based systems (DECsystem-1040, -1050, and -1055), the monitor uses relocation by specifying protection and relocation addresses for the low and high segments. The protection address is the maximum relative address the user can reference. The relocation address is the absolute core address of the first location in the segment, as seen by the monitor in the hardware. The monitor defines these addresses by loading four 8-bit registers (two 8-bit registers in a KA10 based system with the KT10 option instead of the KT10A option), each of which correspond to the left eight bits of an 18-bit PDP-10 address. Thus, segments always contain a multiple of 1024 words.

On KI10-based systems (DECsystem-1070 and -1077), the user's area is page mapped. This means that each page (a page consists of 512 words) of the user's area is associated with a page of physical core memory. Because the assignment of physical pages of core does not need to be contiguous, the monitor has greater freedom in allocating core. The monitor associates (maps) pages in the user's area with physical pages in core in such a way that the user appears to have one or two segments as with a KA10-based system. Therefore, in most cases, the user does not need to be concerned with the type

of processor on which his program is running. However, the unit of core allocation is different on the two processors. The unit of allocation on the KA10 is 1024 words (1K) and on the KI10, is 512 words (1 page).

In general, the term mapping refers to both relocation (KA10) and page mapping (KI10). On either processor, a user address is called a relative or virtual address before it is mapped, and an absolute or physical address after it is mapped.

To take advantage of the fast accumulators, memory addresses $0-17_8$ are not mapped and all users have access to the accumulators. Therefore, relative locations $0-17_8$ cannot be referenced by a user's program. The monitor saves the user's accumulators in this area when the user's program is not running and while the monitor is servicing a UUO from the user. Refer to the PDP-10 System Reference Manual for a more complete description of the relocation and mapping hardware.

1.2 USER'S CORE STORAGE

A user's core storage consists of blocks of memory, the sizes of which are an integral multiple of 1024 (2000_8) words on the KA10-based system and 512_{10} (1000_8) words on the KI10-based system. In a non-reentrant monitor, the user's core storage is a single contiguous block of memory. After mapping, the first address in a block is a multiple of 2000_8 or 1000_8. The relative user and relocated address configurations on the KA10 are shown in Figure 1-1, where P_L, R_L, P_H, and R_H are the protection and relocation addresses for the low and high segments, respectively. If the low segment is more than half the maximum memory capacity ($P_L \geq 400000$), the high segment starts at the first location after the low segment (at $P_L + 2000$). The high segment is limited to 128K. The relative user address configurations on the KI10 are shown in Figure 1-2, where P_L and P_H are the protection addresses for the low and high segments, respectively.

Two methods are available to the user for loading his core area. The simplest way is to load a core image stored on a retrievable device (refer to RUN and GET commands). The other method is to use the relocatable binary loader to link-load binary files. The user can then write the core image on a retrievable device for future use (refer to SAVE command).

1.2.1 Job Data Area (JOBDAT)

The first 140 octal locations of the user's core area are always allocated to the job data area (refer to Table 1-1). Locations in this area are given mnemonic assignments where the beginning characters are .JB. The job data area provides storage for specific information of interest to both the monitor and the user. Some locations, such as .JBSA and .JBDDT, are set by the user's program for use by the monitor. Other locations, such as .JBREL, are set by the monitor and are used by the user's program. In particular, the right half of .JBREL contains the highest legal address set by the monitor when the user's core allocation changes.

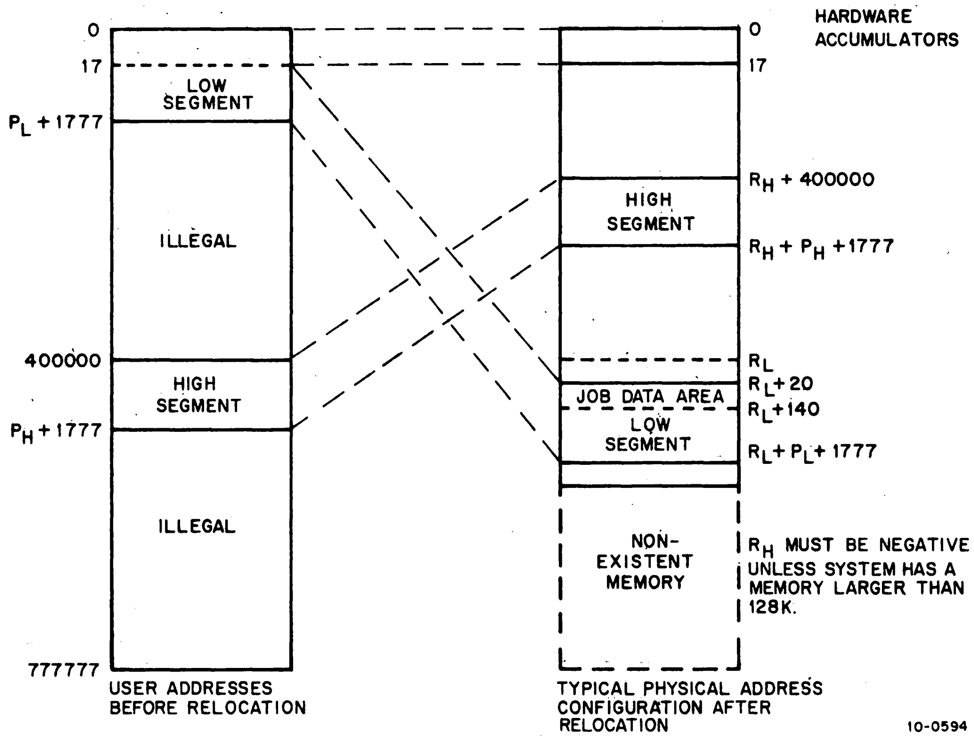


Figure 1-1 KA10 User Address Mapping

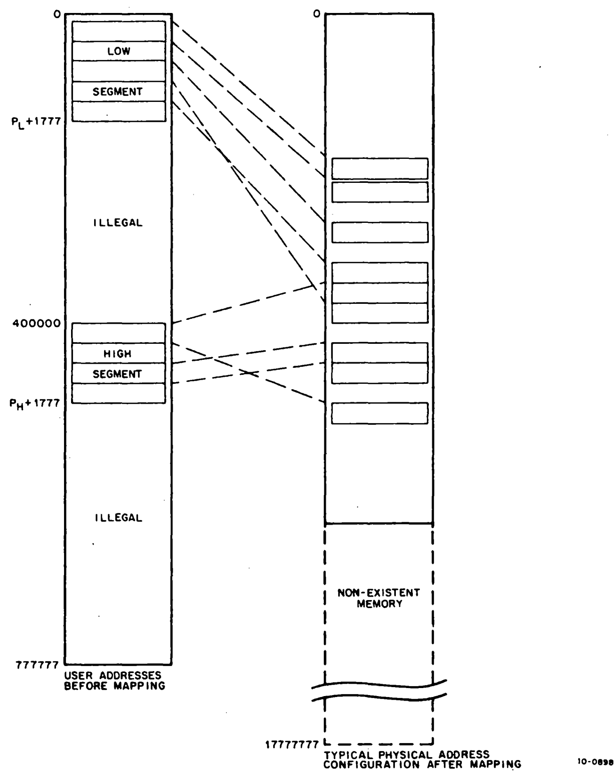


Figure 1-2 KI10 User Address Mapping

Table 1-1
Job Data Area Locations
(for user-program reference)

Name	Octal Location	Description
.JBUUO	40	User's location 40_8. Used by the hardware when processing user UUOs (001 through 037) for storing op code and effective address.
.JB41	41	User's location 41_8. Contains the beginning address of the user's programmed operator service routine (usually a JSR or PUSHJ).
.JBERR	42	Left half: Unused. Right half: Accumulated error count from one system program to the next. System programs should be written to look at the right half only.
.JBREL	44	Left half: Zero. Right half: The highest relative core location available to the user (i.e., the contents of the memory protection register when this user is running).
.JBBLT	45	Three consecutive locations when the LOADER puts a BLT instruction and a CALLI UUO to move the program down on top of itself. These locations are destroyed on every executive UUO by the executive pushdown list.
.JBDDT	74	Left half: The last address of DDT. Right half: The starting address of DDT. If contents are 0, DDT has not been loaded.
.JBCN6	106	Six temporary locations used by the CHAIN program (refer to the LOADER manual) after it releases all I/O channels. .JBCN6 is defined to be in .JBJDA.
.JBPFI	114 (value)	All user I/O must be to locations greater than .JBPFI.
.JBHRL	115	Left half: First relative free location in the high segment (relative to the high segment origin so it is the same as the high segment length). Set by the LOADER and subsequent GETs, even if there is no file to initialize the low segment. The left half is a relative quantity because the high segment can appear at different user origins at the same time. The SAVE command uses this quantity to know how much to write from the high segment. Right half: Highest legal user address in the high segment. Set by the monitor every time the user starts to run or does a CORE or REMAP UUO. The word is $\geqslant$401777 unless there is no high segment, in which case it will be zero. The proper way to test if a high segment exists is to test this word for a non-zero value.

Table 1-1 (Cont)
Job Data Area Locations
(for user-program reference)

Name	Octal Location	Description
.JBSYM	116	Contains a pointer to the symbol table created by linking loader. Left half: Negative of the length of the symbol table. Right half: Lowest address used by the symbol table.
.JBUSY	117	Contains a pointer to the undefined symbol table created by linking loader or defined by DDT. This location has the same format as .JBSYM. There are no undefined symbols if the contents is ≥ 0.
.JBSA	120	Left half: First free location in low segment (set by loader). Right half: Starting address of the user's program.
.JBFF	121	Left half: Zero. Right half: Address of the first free location following the low segment. Set to C $(.JBSA)_{LH}$ by RESET UUO.
.JBREN	124	Left half: Unused. Right half: REENTER starting address. Set by user or by loader and used by REENTER command as an alternate entry point.
.JBAPR	125	Left half: Zero. Right half: Set by user program to trap address when user is enabled to handle APR traps such as illegal memory, pushdown overflow, arithmetic overflow, and clock. See APRENB UUO.
.JBCNI	126	Contains state of APR as stored by CONI APR when a user-enabled APR trap occurs.
.JBTPC	127	Monitor stores PC of next instruction to be executed when a user-enabled APR trap occurs.
.JBOPC	130	The previous contents of the job's last user mode program counter are stored here by monitor on execution of a DDT, REENTER, START, or CSTART command. After a user program HALT instruction followed by a START, DDT, CSTART, or REENTER command, .JBOPC contains the address of the HALT. To proceed at the address specified by the effective address, it is necessary for the user or his program to recompute the effective address of the HALT instruction and to use this address to start. Similarly, after an error during execution of a UUO followed by a START, DDT, CSTART, or REENTER command, .JBOPC points to the address of the UUO. For example, in DDT to continue after a HALT, type .JBOPC/10000,,3010 JRST @ .$X

Table 1-1 (Cont)
Job Data Area Locations
(for user-program reference)

Name	Octal Location	Description
.JBCHN	131	Left half: Zero or the address of first location after first FORTRAN IV loaded program. Right half: Address of first location after first FORTRAN IV Block Data.
.JBCOR	133	Left half: Highest location in low segment loaded with non-zero data. No low file written on SAVE or SSAVE if less than 140. Set by the LOADER. Right half: User argument on last SAVE or GET command. Set by the monitor.
.JBINT	134	Left half: Reserved for the future. Right half: Zero or the address of the error-intercepting block (refer to Paragraph 3.1.3.2).
.JBOPS	135	Reserved for all operating systems.
.JBCST	136	Reserved for customers.
.JBVER	137	Program version number. The bits are defined as follows: Bits 0-2 The group who last modified the program 0 = Digital development group. 1 = Other Digital employees. 2-4 = Reserved for customers. 5-7 = Reserved for customer's users. Bits 3-11 Digital's major version number. Usually incremented by 1 after a release. Bits 12-17 Digital's minor version number. Usually 0, but may be used if an update is needed after work has begun on a new major version. Bits 18-35 Edit number. Usually not reset. The VERSION and the SET WATCH VERSION commands output the version number in standard format. Refer to DECsystem-10 Operating System Commands.
.JBDA	140	The value of this symbol is the first location available to the user.

NOTE: Only those JOBDAT locations of significant importance to the user are given in this table. JOBDAT locations not listed include those that are used by the monitor and those that are unused at present. User programs should not refer to any locations not listed above because such locations are subject to change.

JOBDAT is loaded automatically, if needed, during the loader's library search for undefined global references, and the values are assigned to the mnemonics. JOBDAT exists as a .REL file on device SYS: for loading with user programs that symbolically refer to the locations. User programs should reference locations by the assigned mnemonics, which must be declared as EXTERN references to the assembler. All mnemonics in this manual with a .JB prefix refer to locations in the job data area.

1.2.2 Vestigial Job Data Area

A few constant data in the job data area may be loaded by a two-segment, one-file program without using instructions on a GET command (.JB41, .JBREN, .JBVER), and some locations are loaded by the monitor on a GET (.JBSA, .JBCOR, .JBHRL). The vestigial job data area (the first 10 locations of the high segment) is reserved for these low-segment constants; therefore, a high-segment program is loaded into 400010 instead of 400000 (refer to Table 1-2). With the vestigial job data area in the high segment, the loader automatically loads the constant data into the job data area without requiring a low file on a GET, R, or RUN command, or a RUN UUO. SAVE will write a low file for a two-segment program only if the LH of .JBCOR is 140_8 or greater.

Table 1-2
Vestigial Job Data Area Locations

Symbol	Octal Location†	Description
.JBHSA	0	A copy of .JBSA.
.JBH41	1	A copy of .JB41.
.JBHCR	2	A copy of .JBCOR.
.JBHRN	3	LH: restores the LH of .JBHRL, RH: restores the RH of .JBREN.
.JBHVR	4	A copy of .JBVER.
.JBHNM	5	High segment name set on a SAVE.
.JBHSM	6	A pointer to the high-segment symbols, if any.
	7	Reserved for future use.
.JBHDA	10	First location not used by vestigial job data area.

†Relative to origin of high segment, usually .JBHGH = 400000_8.

CHAPTER 2
INTRODUCTION TO USER PROGRAMMING

2.1 PROCESSOR MODES

In a single-user, non-timesharing system, the user's program is subject only to those conditions inherent in the hardware. The program must

a. Stay within the memory capacity.

b. Observe the hardware restrictions placed on the use of certain memory locations.

c. Observe the restriction on interrupt instructions.

With timesharing, the hardware limits the central processor operations to one of three modes: user mode, user I/O mode, and executive mode.

2.1.1 User Mode

Normally, user programs run with the processor in user mode and must operate within an assigned area of core. In user mode, certain instructions are illegal. User mode is used to guarantee the integrity of the monitor and each user program. The user mode of the processor is characterized by the following:

a. Automatic memory protection and mapping (refer to Chapter 1).

b. Trap to absolute location 40 in the monitor on any of the following:

(1) Operation codes 040 through 077 and operation code 00,

(2) Input/output instructions (DATAI, DATAO, BLKI, BLKO, CONI, CONO, CONSZ, and CONSO),

(3) HALT (i.e., JRST 4,),

(4) Any JRST instruction that attempts to enter executive mode or user I/O mode.

c. Trap to relative location 40 in the user area on execution of operation codes 001 through 037.

2.1.2 User I/O Mode

The user I/O mode (bits 5 and 6 of PC word = 11) of the central processor allows privileged user programs to be run with automatic protection and mapping in effect, as well as the normal execution of

all defined operation codes. The user I/O mode provides some protection against partially debugged monitor routines and permits infrequently used device service routines to be run as a user job. Direct control of special devices by the user program is particularly important in real-time applications.

To utilize this mode, the user must have bit 15 (JB.TRP) set in the privilege word. RESET AC, or CALLI 0 terminates user I/O mode. User I/O mode is not used by the monitor and is normally not available to the timesharing user (refer to Paragraph 3.8.3).

2.1.3 Executive Mode

The monitor operates with the processor in executive mode, which is characterized by special memory protection and mapping (refer to Chapter 1) and by the normal execution of all defined operation codes.

User programs run in user mode; therefore, the monitor must schedule user programs, service interrupts, perform all input and output operations, take action when control returns from a user program, and perform any other legal user-requested operations that are not available in user mode. The services the monitor makes available to user-mode programs and how a user program obtains these services, are described in Chapters 3 and 4.

2.2 PROGRAMMED OPERATORS (UUOs)

Operation codes 000 through 077 in the PDP-10 are programmed operators, sometimes referred to as UUOs. They are software-implemented instructions because from a hardware point of view, their function is not pre-specified. Some of these op-codes trap to the monitor, and the rest trap to the user program.

After the effective address calculation is complete, the contents of the instruction register, along with the effective address, are stored, and an instruction is executed out of the normal sequence.

Although there is one operating system for all configurations of the DECsystem-10, some UUOs may not be included in each DECsystem-10. This is especially true of the DECsystem-1040, the basic system intended for small installations who do not want all of the system's features because of a constraint on core. UUOs are deleted from the DECsystem-1040 by feature test switches defined at MONGEN time. In the standard DECsystem-1040, many of these switches are off, and therefore, the corresponding UUOs are not available. This saves core but limits various features of the operating system. In the UUO descriptions that follow, footnotes indicate if the switch is normally absent in the DECsystem-1040. If not stated, the UUO is available on all configurations of the DECsystem-10.

2.2.1 Operation Codes 001-037 (User UUOs)

Operation codes 001 through 037 do not affect the mode of the central processor; thus, when executed in user mode, they trap to user location 40, which allows the user program complete freedom in the use of these programmed operators.

If a user's undebugged program accidentally executes one of these op-codes when the user did not intend to use it, the following error message is normally issued:

HALT AT USER PC addr

This message is given because the user's relative location 41 contains HALT (unless his program has overtly changed it) which is provided by the loader; addr is the location of the user UUO.

2.2.2 Operation Codes 040-077 and 000 (Monitor UUOs)

Operation codes 040 through 077 and 000 trap to absolute location 40, with the central processor in executive mode. These programmed operators are interpreted by the monitor to perform I/O operations and other control functions for the user's program.

Operation code 000 always returns the user to monitor mode with the error message:

?ILLEGAL UUO AT USER PC addr

Table 2-1 lists the operation codes 040 through 077 and their mnemonics.

Table 2-1
Monitor Programmed Operators

Op Code	Call	Function
040	CALL AC, [SIXBIT/NAME/] , or NAME AC,	Programmed operator extension (refer to Paragraph 2.2.2.1).
041	INIT D, MODE SIXBIT/DEV/ XWD OBUF, IBUF error return normal return	Select I/O device (refer to Paragraph 4.2.3).
042		No operation
043		No operation
044		No operation
045		No operation
046		No operation

(042–046: Reserved for installation-dependent definition.)

Table 2-1 (Cont)
Monitor Programmed Operators

Op Code	Call	Function
047	CALLI AC, N	Programmed operator extension (refer to Paragraph 2.2.2.1).
050	OPEN, D, E error return normal return E: EXP STATUS SIXBIT /DEV/ XWD OBUF, IBUF	Select I/O device (refer to Paragraph 4.2.3).
051	TTCALL AC, ADR	Extended operations on job-controlling terminal (refer to Paragraph 5.10.3).
052		Reserved for future expansion by DEC.
053		Reserved for future expansion by DEC.
054		Reserved for future expansion by DEC.
055	RENAME D, E error return normal return E: SIXBIT /FILE/ SIXBIT /EXT/ EXP <PROT> B8+DATE XWD PROJ, PROG	Rename or delete a file (see Section 4.4.3).
056	IN D, normal return error or EOF return	INPUT and skip on error or EOF (see Section 4.5).
057	OUT D, normal return error return	OUTPUT and skip on error or EOT (see Section 4.5).
060	SETSTS D, STATUS	Set file status (see Section 4.6.2).
061	STATO D, BITS R0: NO SELECTED BITS = 1 R1: SOME SELECTED BITS = 1	Skip if file status bits = 1 (see Section 4.6.1).
062	GETSTS D, E	Copy file status to E (see Section 4.6.1).
063	STATZ D, BITS R0: SOME SELECTED BITS = 1 R1: ALL SELECTED BITS = 0	Skip if file status bits = 0 (see Section 4.6.1).
064	INBUF D, N	Set up input buffer ring with N buffers (refer to Paragraph 4.3.2).
065	OUTBUF D, N	Set up output buffer ring with N buffers (refer to Paragraph 4.3.2).
066	INPUT D,	Request input or request next buffer (refer to Paragraph 4.5).

Table 2-1 (Cont)
Monitor Programmed Operators

Op Code	Call	Function
067	OUTPUT D,	Request output or request next buffer (refer to Paragraph 4.5).
070	CLOSE D,	Terminate file operation (refer to Paragraph 4.7).
071	RELEAS D,	Release device (refer to Paragraph 4.8.1).
072	MTAPE D, N	Perform tape positioning operation (refer to Paragraphs 5.5.3 and 6.1.6.5).
073	UGETF D,	Get next free block number on DECtape (refer to Paragraph 6.1.6.3).
074	USETI D, E	Set next input block number (refer to Paragraphs 6.1.6.1 and 6.2.9.2).
075	USETO D, E	Set next output block number (refer to Paragraphs 6.1.6.2 and 6.2.9.2).
076	LOOKUP D, E error return normal return E: SIXBIT /FILE/ SIXBIT /EXT/ 0 XWD PROJ, PROG	Select a file for input (refer to Paragraph 4.4.1).
077	ENTER D, E error return normal return E: SIXBIT /FILE/ SIXBIT /EXT/ 0 XWD PROJ, PROG	Select a file for output (refer to Paragraph 4.4.2).
100	UJEN	Dismiss real-time interrupt (refer to Paragraph 3.8.4).

2.2.2.1 CALL and CALLI - Operation codes 040 through 077 limit the monitor to 40_8 operations. The CALL operation extends this set by specifying the name of the operation by the contents of the location specified by the effective address (e.g., CALL [SIXBIT /EXIT/]). This capability provides for indefinite extendability of the monitor operations, at the overhead cost to the monitor of a table lookup.

The CALLI operation eliminates the table lookup of the CALL operation by having the programmer or the assembler perform the lookup and specify the index to the operation in the effective address of the CALLI. Table 2-2 lists the monitor operations specified by the CALL and CALLI operations.

Table 2-2
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, -2 ... -n		Customer defined	Reserved for definition by each customer installation.
CALLI AC, -1	LIGHTS	CALL AC, [SIXBIT/LIGHTS/]	Display AC in console lights (refer to Paragraph 3.6.4.2).
CALLI AC, 0	RESET	CALL [SIXBIT/RESET/] return	Reset I/O device (refer to Paragraph 4.1.2).
CALLI AC, 1	DDTIN	MOVEI AC, BUFFER CALL AC, [SIXBIT/DDTIN/] only return	DDT mode console input (refer to Paragraph 5.9.2).
CALLI AC, 2	SETDDT	MOVEI AC, DDT-start-adr CALL AC, [SIXBIT/SETDDT/] only return	Set protected DDT starting address (refer to Paragraph 3.1.1.1).
CALLI AC, 3	DDTOUT	MOVEI AC, BUFFER CALL AC, [SIXBIT/DDTOUT/] only return	DDT mode console output (refer to Paragraph 5.9.2).
CALLI AC, 4	DEVCHR	MOVE AC, [SIXBIT/dev/] or MOVEI AC, channel no. CALL AC, [SIXBIT/DEVCHR/] only return C(AC) = 0 if no such device C(AC) = DEVMOD word of device data block if device is found.	Get device characteristics (refer to Paragraph 4.10.2).
CALLI AC, 5	DDTGT	CALL AC, [SIXBIT/DDTGT/] only return	No operation, historical UUO.
CALLI AC, 6	GETCHR	AC: = SIXBIT/DEV/ CALL AC, [SIXBIT/GETCHR/] only return	Same as CALLI AC, 4.
CALLI AC, 7	DDTRL	CALL AC, [SIXBIT/DDTRL/] only return	No operation; historical UUO.
CALL AC, 10	WAIT	AC field is software channel number. CALL AC, [SIXBIT/WAIT/] only return	Wait until device is inactive (refer to Paragraph 4.5.3).
CALLI AC, 11	CORE	MOVE AC, [XWD HIGH ADR or 0, LOW ADR or 0] CALL AC, [SIXBIT/CORE/] error return, assignment unchanged normal return, new assignment AC: = max. core available (in 1K blocks) on error or normal return.	Allocate core (refer to Paragraph 3.2.3).

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, 12	EXIT	CALL AC, [SIXBIT/EXIT/] return If AC ≠ 0, devices are not released and CONT and CCONT commands are effective.	Stop job, may release devices depending on contents of AC (refer to Paragraph 3.1.2.3).
CALLI AC, 13	UTPCLR	AC field is software channel number CALL AC, [SIXBIT/UTPCLR/] only return	Clear DECtape directory (refer to Paragraph 6.1.6.4).
CALLI AC, 14	DATE	CALL AC, [SIXBIT/DATE/] only return AC: = date in compressed format	Return date (refer to Paragraph 3.6.1.1).
CALLI AC, 15	LOGIN††	MOVE AC, [XWD -N, LOC] CALL AC, [SIXBIT/LOGIN/] R0: return Does not return if C(R0) is a HALT instruction.	Privileged UUO in that the calling job must not be logged in. Is a no-op if executed by a job already logged-in.
CALLI AC, 16	APRENB	MOVEI AC, BITS CALL AC, [SIXBIT/APRENB/] return	Enable central processor traps (refer to Paragraph 3.1.3.1).
CALLI AC, 17	LOGOUT††	CALL AC, [SIXBIT/LOGOUT/] no return	Privileged UUO available only to system-privileged programs. Is treated like an EXIT UUO if executed by a non-system-privileged program.
CALLI AC, 20	SWITCH	CALL AC, [SIXBIT/SWITCH/] return AC: contents of console data switches	Read console data switches (refer to Paragraph 3.6.4.1).
CALLI AC, 21	REASSI	MOVE AC, job number MOVE AC+1, [SIXBIT/DEV/] CALL AC, [SIXBIT/REASSI/] return If C(AC) = 0 on return, the job specified has not been initialized. If C(AC+1) = 0 on return, the device is not assigned to calling job, or device is TTY.	Reassign device (refer to Paragraph 4.8.3).
CALLI AC, 22	TIMER	CALL AC, [SIXBIT/TIMER/] return AC: = time in jiffies, right justified.	Read time of day in clock ticks (refer to Paragraph 3.6.1.2).
CALLI AC, 23	MSTIME	CALL AC, [SIXBIT/MSTIME/] return AC: = time in milliseconds, right-justified.	Read time of day in milliseconds (refer to Paragraph 3.6.1.3).

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, 24	GETPPN	CALL AC, [SIXBIT/GETPPN/] normal return alternate return AC: = XWD proj. no., prog. no. of this job. Alternate return is taken only if job is privileged and the same proj-prog number occurs twice in the table of jobs logged in.	Return project-programmer number of job (refer to Paragraph 3.6.2.3).
CALLI AC, 25	TRPSET	MOVE AC, [XWD N, LOC] CALL AC, [SIXBIT/TRPSET/] error return normal return LOC: JSR TRAP	Set trap for user I/O mode (refer to Paragraph 3.8.3).
CALLI AC, 26	TRPJEN	CALL [SIXBIT/TRPJEN/]	Illegal UUO; replaced by UJEN (op code 100).
CALLI AC, 27	RUNTIM	MOVE AC, job number or 0 CALL AC, [SIXBIT/RUNTIM/] only return AC: = running time of job AC: = 0 if non-existent job	Return the jobs running time in milliseconds (refer to Paragraph 3.6.2.1).
CALLI AC, 30	PJOB	CALL AC, [SIXBIT/PJOB/] return AC: = job number, right-justified	Return job number (refer to Paragraph 3.6.2.2).
CALLI AC, 31	SLEEP	MOVE AC, time to sleep in seconds CALL AC, [SIXBIT/SLEEP/] return	Stop job for specified time in seconds (refer to Paragraph 3.1.4.1).
CALLI AC, 32	SETPOV	CALL AC, [SIXBIT/SETPOV/] return	Superseded by APRENB UUO.
CALLI AC, 33	PEEK	MOVEI AC, exec adr CALL AC, [SIXBIT/PEEK/] return AC: = C(exec-adr)	Return contents of executive address (refer to Paragraph 3.6.3.1).
CALLI AC, 34	GETLIN	CALL AC, [SIXBIT/GETLIN/] return AC: = SIXBIT TTY name, left-justified (e.g., CTY, TTY27)	Return SIXBIT name of attached terminal (refer to Paragraph 5.9.4).
CALLI AC, 35	RUN	MOVSI AC, start adr increment HRRI AC, E RUN AC, error return normal return	Transfer control to selected program (refer to Paragraph 3.3.1).

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI[†] Mnemonic	CALL	Function
CALLI AC, 35 (continued)	RUN	E: SIXBIT/DEVICE/ SIXBIT/FILE/ SIXBIT/EXT/ 0 XWD proj no. prog no XWD 0; optional core assignment	
CALLI AC, 36	SETUWP	MOVEI AC, BIT SETUWP AC, error return normal return	Set or clear user mode write protect for high segment (refer to Paragraph 3.2.4).
CALLI AC, 37	REMAP	MOVEI AC, highest adr. in low seg REMAP AC, error return normal return	Remap top of low segment into high segment (refer to Paragraph 3.3.3).
CALLI AC, 40	GETSEG	MOVEI AC, E GETSEG AC, error return normal return E: SIXBIT/DEVICE/ SIXBIT/FILE/ SIXBIT/EXT/ 0 XWD proj no, prog no 0	Replace high segment in user's addressing space (refer to Paragraph 3.3.2).
CALLI AC, 41	GETTAB	MOVSI AC, job no. or index no. HRRI AC, table no. GETTAB AC, error return normal return C(AC) unchanged on error return AC: = table entry if table is defined and index is in range.	Return contents of monitor table or location (refer to Paragraph 3.6.3.4).
CALLI AC, 42	SPY	MOVEI AC, highest physical adr. desired SPY AC, error return normal return	Make physical core be high segment for examination of monitor (refer to Paragraph 3.6.3.2).
CALLI AC, 43	SETNAM	MOVE AC, [SIXBIT/NAME/] SETNAM AC, return	Set program name in monitor job table (refer to Paragraph 3.4.1).
CALLI AC, 44	TMPCOR	MOVE AC, [XWD CODE, BLOCK] TMPCOR, error return normal return	Allow temporary in-core file storage for job (refer to Paragraph 3.5.1).

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, 44 (continued)	TMPCOR	BLOCK: XWD NAME, 0 IOWD BUFFLEN,BUFFER AC: = value depending on CODE and whether error or normal return is taken.	
CALLI AC, 45	DSKCHR	MOVE AC, [XWD+N, LOC] DSKCHR AC, error return normal return AC: = XWD status, configuration LOC: = SIXBIT/NAME/ 0 } 0 } values returned 0 }	Return disk characteristics (refer to Paragraph 6.2.9.13).
CALLI AC, 46	SYSSTR	MOVEI AC, 0 or MOVE AC, [SIXBIT/FSNAME/] SYSSTR AC, error return - not a file structure normal return AC: = next file structure name in SIXBIT, left-justified	Return next file structure name (refer to Paragraph 6.2.9.10).
CALLI AC, 47	JOBSTR	MOVE AC, [XWD N, LOC] JOBSTR AC, error return normal return AC: = argument Contents — Use LOC/SIXBIT/NAME/or — 1 arg LOC+1/XWD proj no. prog. no. — value LOC+2/status bits — value	Return next file structure name in the jobs search list (refer to Paragraph 6.2.9.8).
CALLI AC, 50	STRUUO	MOVE AC, [XWD N, LOC] STRUUO AC, error return normal return AC: = status or error code Contents — Use LOC:function numbers — arg LOC+1: — arg depending on function number . :	Manipulate file structures (refer to Paragraph 6.2.9.7).
CALLI AC, 51	SYSPHY	MOVEI AC, 0 or last unit name SYSPHY AC, error return normal return	Return all physical disk units (refer to Paragraph 6.2.9.11).

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, 52	FRECHN		Reserved for future use.
CALLI AC, 53	DEVTYP	MOVE AC, [SIXBIT/dev/] or MOVEI AC, channel no. DEVTYP AC, error return normal return	Return properties of device (refer to Paragraph 4.10.3).
CALLI AC, 54	DEVSTS	MOVEI AC, channel no. of device DEVSTS AC, error return normal return	Return hardware device status word (refer to Paragraph 4.10.1).
CALLI AC, 55	DEVPPN	MOVE AC, [SIXBIT/DEV/] DEVPPN AC, error return normal return AC: = XWD proj-prog. number on a normal return	Return the project programmer number associated with a device (refer to Paragraph 6.2.9.12).
CALLI AC, 56	SEEK†††	AC is software channel number SEEK AC, return	Perform a SEEK to current selected block for software channel AC (refer to Paragraph 6.2.9.3).
CALLI AC, 57	RTTRP	MOVEI AC, RTBLK RTTRP AC, error return normal return	Connect real-time devices to PI system (refer to Paragraph 3.8.1).
CALLI AC, 60	LOCK	MOVE AC, [XWD high seg code, low seg code] LOCK AC, error return normal return	Lock job in core (refer to Paragraph 3.2.2).
CALLI AC, 61	JOBSTS	MOVEI AC, channel no. or MOVNI AC, job JOBSTS AC, error return normal return	Return status information about device TTY and/or controlled job (refer to Paragraph 5.9.4.4).
CALLI AC, 62	LOCATE	MOVEI AC, location LOCATE AC, error return normal return	Change the job's logical station (refer to Paragraph 3.4.3).
CALLI AC, 63	WHERE	MOVEI AC, channel no. or MOVE AC, [SIXBIT/dev/] WHERE AC, error return normal return	Return the physical station of the device (refer to Paragraph 4.10.5).

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI[†] Mnemonic	CALL	Function
CALLI AC, 64	DEVNAM	MOVEI AC, channel no. or MOVE AC, [SIXBIT/dev/] DEVNAM AC, error return normal return	Return physical name of device obtained through generic INIT/OPEN or logical device assignment (refer to Paragraph 4.10.6).
CALLI AC, 65	CTLJOB	MOVE AC, job number CTLJOB AC, error return normal return	Return job number of controlling job (refer to Paragraph 5.9.4.5).
CALLI AC, 66	GOBSTR	MOVE AC, [XWD N, LOC] GOBSTR AC, error return normal return LOC: job number LOC+1: XWD proj no, prog no LOC+2: SIXBIT/NAME/or-1 LOC+3: 0 LOC+4: Status bits	Return next file structure name in an arbitrary job's search list (refer to Paragraph 6.2.9.9).
CALLI AC, 67	ACTIVATE		Reserved for the future.
CALLI AC, 70	DEACTIVATE		(Reserved for the future.)
CALLI AC, 71	HPQ	MOVE AC, high-priority queue no. HPQ AC, error return normal return	Place job in high priority scheduler's run queue (refer to Paragraph 3.8.5).
CALLI AC, 72	HIBER	MOVSI AC, enable bits HRRI AC, sleep time HIBER AC, error return normal return	Allow job to become dormant until the specified event occurs (refer to Paragraph 3.1.4.2).
CALLI AC, 73	WAKE	MOVE AC, job no. WAKE AC, error return normal return	Allow job to activate the specified dormant job (refer to Paragraph 3.1.4.3).
CALLI AC, 74	CHGPPN[††]	MOVE AC, new proj. prog. no. CHGPPN AC, error return normal return	Change project-programmer number. Gives an error return if executed by a job already logged-in.
CALLI AC, 75	SETUUO	MOVE AC, [XWD function, argument] SETUUO AC, error return normal return	Set system and job parameters (refer to Paragraph 3.4.2).
CALLI AC, 76	DEVGEN		Reserved for the future.

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI[†] Mnemonic	CALL	Function
CALLI AC, 77	OTHUSR	OTHUSR AC, non-skip return skip return AC: = proj. prog. no.	Determine if another job is logged in with same project-programmer number (refer to Paragraph 3.6.2.4).
CALLI AC, 100	CHKACC	MOVE AC, [EXP LOC] CHKACC AC, error return normal return LOC: XWD action, protection LOC+1: directory proj-prog no. LOC+2: user proj-prog no.	Check user's access to the file specified (refer to Paragraph 6.2.9.6).
CALLI AC, 101	DEVSIZ	MOVE AC, [EXP LOC] DEVSIZ AC, error return normal return LOC: EXP STATUS LOC+1: SIXBIT/dev/	Determine buffer size for the specified device (refer to Paragraph 4.10.4).
CALLI AC, 102	DAEMON	MOVE AC, [XWD + length, adr of arg. list] DAEMON AC, error return normal return	Request DAEMON to perform a specified task (refer to Paragraph 3.7).
CALLI AC, 103	JOBPEK[††]	MOVE AC, adr of arg block JOBPEK AC, error return normal return	Read or write another job's core. Gives the error return if executed by a non-system-privileged program.
CALLI AC, 104	ATTACH[††]	MOVE AC, [XWD line no., job no.] ATTACH AC, error return normal return	Attach the job to the specified TTY line number. Gives the error return if executed by a non-system-privileged program.
CALLI AC, 105	DAEFIN[††]	MOVE AC, [XWD + length, adr of arg. list] DAEFIN AC, error return normal return	Indicate that the request to the DAEMON program has been completed. Gives the error return if executed by a non-system-privileged program.
CALLI AC, 106	FRCUUO[††]	MOVE AC, [XWD + length, adr of arg. list] FRCUUO AC, error return normal return	Force a command for a job. Gives the error return if executed by a non-system-privileged program.

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, 107	DEVLNM	MOVE AC, [SIXBIT/dev/] or MOVEI AC, channel no. MOVE AC+1, [SIXBIT/logical name/] DEVLNM AC, error return normal return	Set a logical name for this specified device (refer to Paragraph 4.8.4).
CALLI AC, 110	PATH.	MOVE AC, [XWD + length, adr. of argument list] PATH. AC, error return normal return ADR: N or SIXBIT/NAME/ ADR+1: Scan switch ADR+2: PPN ADR+3: SFD name ADR+4: SFD name ⋮	Read or modify the default directory path or read the current path of a file OPEN on a channel. Refer to Paragraph 6.2.9.1.
CALLI AC, 111	METER.	MOVE AC, [XWD N, LOC] METER. AC, error return normal return LOC: function code LOC+1: argument depends LOC+2: on function code used. ⋮ LOC+N-1:	Provide performance analysis and metering of dynamic system variables. Refer to Paragraph 3.9.
CALLI AC, 112	MTCHR.	MOVEI AC, channel no. or MOVE AC, [SIXBIT/dev/] MTCHR. AC, error return normal return	Return characteristics of the magnetic tape. Refer to Paragraph 5.5.3.2.
CALLI AC, 113	JBSET.††	MOVE AC, [2,,BLOCK] JBSET. AC, error return normal return BLOCK: 0,, job number BLOCK+1: function,, value	Execute the specified function of SETUUO for a particular job.
CALLI AC, 114	POKE.	MOVE AC, [3,,BLOCK] POKE. AC, error return normal return BLOCK: location BLOCK+1: old value BLOCK+2: new value	Alter the specified location in the Monitor. Refer to Paragraph 3.6.3.3.

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

CALLI	CALLI† Mnemonic	CALL	Function
CALLI AC, 115	TRMNO.	MOVE AC, job number TRMNO. AC, error return normal return	Return number of the terminal currently controlling the specified job. Refer to Paragraph 5.10.5.
CALLI AC, 116	TRMOP.	MOVE AC, [XWD N, ADR] TRMOP. AC, error return normal return ADR: function code ADR+1: terminals's universal index . Following arguments depend : on function used.	Perform miscellaneous terminal functions. Refer to Paragraph 5.10.6.
CALLI AC, 117	RESDV.	MOVE AC, channel no. RESDV. AC, error return normal return	Reset the specified channel. Refer to Paragraph 4.8.2.

CALLI AC, 120	UNLOK.	MOVSI AC,1
		MOVSI AC,0
		HRRI AC,1
		HRRI AC,0
		UNLOK. AC,
		error return
		normal return
			Unlock a locked job in core. Refer to Paragraph 3.2.2.4.

CALLI AC, 121	DISK.	MOVE AC, [XWD function, ADR] DISK. AC, error return normal return	Set or read a disk or file system parameter (e.g., set the disk priority for a channel or the job). Refer to Paragraph 6.2.9.14.
CALLI AC, 122	DVRST.††	MOVE AC, [SIXBIT/dev/] or MOVEI AC, channel no. DVRST. AC, error return normal return	Restrict the specified device to a privileged job.
CALLI AC, 123	DVURS.††	MOVE AC, [SIXBIT/dev/] or MOVEI AC, channel no. DVURS. AC, error return normal return	Remove the restricted status of the specified device.

†The CALLI mnemonics are defined in a separate MACRO assembler table, which is scanned whenever an undefined OP CODE is found. If the symbol is found in the CALLI table, it is defined as though it had appeared in an appropriate OPDEF statement, that is

RETURN : EXIT

If EXIT is undefined, it will be assembled as though the program contained the statement

OPDEF EXIT [CALLI 12]

This facility is available in MACRO V.43 and later.

Table 2-2 (Cont)
CALL and CALLI Monitor Operations

††This CALLI is a system-privileged UUO available only to users logged in under [1,2] or to programs running with the JACCT bit set. Complete documentation for system-privileged UUOs appears in the Specifications section of the DECsystem-10 Software Notebooks.
†††All CALLI's above CALLI 55 do not have a corresponding CALL with a SIXBIT argument. This is to save monitor table space.

The customer is allowed to add his own CALL and CALLI calls to the monitor. A negative CALLI effective address (-2 or less) should be used to specify such customer-added operations.

2.2.2.2 Suppression of Logical Device Names - Some system programs, e.g., LOGOUT, require I/O to specific physical devices regardless of the logical name assignments. Therefore, for any CALLI, if bit 19 (UU.PHS) in the effective address of the CALLI is not equal to bit 18, only physical names will be used; logical device assignments will be ignored. This suppression of logical device names will be ignored. This suppression of logical device names is helpful, for example, when using the results of the DEVNAM UUO where the physical name corresponding to a logical name is returned.

2.2.2.3 Restriction on Monitor UUOs in Reentrant User Programs - A number of restrictions on UUOs that involve a high segment prevent naive or malicious users from interferring with other users while sharing segments and minimize monitor overhead in handling two-segment programs. The basic rules are as follows:

a. All UUOs can be executed from the low or high segment although some of their arguments cannot be in or refer to the high segment.

b. No buffers, buffer headers, or dump-mode command lists may exist in the high segment for reading from or writing to any I/O device.

c. No I/O is processed into or out of the high segment except via the SAVE and SSAVE commands.

d. No STATUS, CALL or CALLI UUO allows a store in the high segment.

e. The effective address of the LOOKUP, ENTER, INPUT, OUTPUT, and RENAME UUOs cannot be in the high segment. If any rule is violated, an address check error message is given.

f. As a convenience in writing user programs, the monitor makes a special check so that the INIT UUO can be executed from the high segment, although the calling sequence is in the high segment. The monitor also allows the effective address of the CALL UUO, which contains the SIXBIT monitor function name, and the effective address of the OPEN UUO, which contains the status bits, device name, and buffer header addresses, in the high segment. The address of TTCALL 1, and TTCALL 3, may be in the high segment for convenience in typing messages.

2.2.3 Operation Codes 100-127 (Unimplemented Op Codes)

Op code 100(UJEN)	Dismiss real-time interrupt from user mode (refer to Paragraph 3.8.4).
Op codes 101-107 114-117 123	Monitor prints ?ILL INST. AT USER n and stops the job.
Op codes 110-113 120-122 124-127	These op codes are valid on the KI10. If used on the KA10, the monitor prints ?KI10 ONLY INST. AT USER n and stops the job.

2.2.4 Illegal Operation Codes

The eight I/O instructions (e.g., DATAI) and JRST instructions with bit 9 or 10 = 1 (e.g., HALT, JEN) are interpreted by the monitor as illegal instructions (refer to the System Reference Manual in the Software Notebooks). The job is stopped and a question mark is printed immediately. A carriage return-line feed is then output, followed by an error message. For example, a DATAI instruction would produce the following:

```
?
? ILL INST AT USER addr
```

2.2.5 Naming Conventions for Monitor Symbols

The names of the monitor's data base symbols contain dots or percent signs so that they can be made user-mode symbols without conflicting with previously-coded user programs. Data symbols can be divided into five classes:

1) numbers
2) masks
3) UUO names
4) GETTAB arguments
5) error codes.

Symbols defining numbers begin with a dot, followed by a two-letter prefix indicating the type of number, and end with a three-character abbreviation representing the specific number. Numbers are 18-bit quantities and include core addresses and function codes. The following are examples of names of various numbers:

.JBxxx	Job Data Area
.GTxxx	GETTAB table numbers
.RBxxx	Extended arguments for LOOKUP, ENTER, RENAME

Names for masks start with a two-letter prefix indicating the individual word, followed by a dot, and end with three characters representing the specific mask. Masks are 36-bit quantities and include bits and fields. The following are examples of names of masks:

JP.xxx	Privilege word bits
JW.xxx	WATCH word bits
PC.xxx	PC word bits

Names for UUOs implemented after the 5.03 release of the monitor are five or less characters followed by a dot. For example,

PATH.	UUO to modify directory path
TRMOP.	UUO to perform terminal functions.

Individual words within a GETTAB table start with a percent sign, followed by two characters representing the generic name of the table, and end with three characters identifying the specific word. For example,

%NSCMX	CORMAX word in the nonswapping data table.
%CNSTS	States word in the configuration table.

Names of bytes and bits within a GETTAB word begin with two characters representing the word, followed by a percent sign, and end with three characters designating the specific byte.

ST % DSK	Byte representing disk system; contained in the states word.
ST % SWP	Byte indicating swapping system; contained in the states word.

Error codes returned on a UUO error have names with the following pattern: two characters indicating the UUO, three characters designating the failure type, and a terminating percent sign.

DMILF%	DAEMON error; illegal function.
RTDIU%	RTTRP error; device in use.
LKNLP%	LOCK error; no locking privileges.

Many of the values useful in user programming are encoded in the parameter file C.MAC for the convenience of writing and modifying programs.

CHAPTER 3
NON-I/O UUOS

3.1 EXECUTION CONTROL

3.1.1 Starting

A user program may start another program only by using the RUN or GETSEG UUOs (refer to Paragraphs 3.3.1 and 3.3.2). A user at a terminal may start a program with the monitor commands RUN, START, CSTART, CONT, CCONT, DDT, and REENTER (refer to DECsystem-10 Operating System Commands). The starting address either appears as an argument of the command or is stored in the user's job data area (refer to Chapter 1).

3.1.1.1 SETDDT AC, or CALLI AC, 2 - This UUO causes the contents of the AC to replace the DDT starting address, which is stored in the protected job data area location .JBDDT. The starting address is used by the monitor command, DDT.

3.1.2 Stopping

Any of the following procedures can stop a running program:

a. One ↑C from the user's terminal if the user program is in a TTY input wait; otherwise, two ↑Cs from the user's terminal (refer to DECsystem-10 Operating System Commands);

b. A monitor detected error;

c. Program execution of HALT, CALL [SIXBIT/EXIT/], or CALL [SIXBIT/LOGOUT/].

3.1.2.1 Illegal Instructions (700-777, JRST 10, JRST 14) and Unimplemented OP Codes (101-127)- Illegal instructions trap to the monitor, stop the job, and print:

?ILL INST. AT USER adr   or   ?KI ONLY INST. AT USER adr

Refer to Paragraph 2.2.3 for an explanation of op codes 101-127. Note that the program cannot be continued by typing the CONT or CCONT commands.

3.1.2.2 HALT or JRST 4 - The HALT instruction is an exception to the illegal instructions; it traps to the monitor, stops the job, and prints:

?HALT AT USER adr

where n is the location of the HALT instruction. If the HALT instruction is in location 41 and the program executed a user UUO (operation codes 001-037), the address in the error message is that of the user UUO instead of address 41.

However, the CONT and CCONT commands are still valid, and, if typed, will continue the program at the effective address of the HALT instruction. After a user program HALT instruction followed by a START, DDT, CSTART, or REENTER command, .JBOPC contains the address of the HALT. To proceed at the address specified by the effective address, it is necessary for the user or his program to recompute the effective address of the HALT instruction and to use this address to start (refer to .JBOPC description, Table 1-1 in Paragraph 1.2.1). HALT is not the instruction used to terminate a program (refer to Paragraph 3.1.2.3). HALT is useful for indicating impossible error conditions.

3.1.2.3 EXIT AC, or CALLI AC, 12 - When the value of AC is zero, all I/O devices (including real-time devices) are RELEASed (refer to Paragraph 4.8.1); the job is unlocked from core; the user mode write protect bit (UWP) for the high segment is set; the APR traps are reset to 0; the PC flags are cleared; and the job is stopped. If timesharing was stopped (refer to Paragraph 3.8.3), it is resumed. In other words, after releasing all I/O devices that close out all files, a RESET is done (refer to Paragraph 4.1.2). The carriage-return/line-feed is performed, and

EXIT

is printed on the user's terminal, which is left in monitor mode. The CONT and CCONT commands cannot continue the program.

When the value of AC is nonzero, the job is stopped, but devices are not RELEASed and a RESET is not done. Instead of printing EXIT, only a carriage-return and line-feed is performed, and a period is printed on the user's terminal. The CONT and CCONT commands may be used to continue the program. In other words, this form of EXIT does not affect the state of the job except to stop it and return the terminal to monitor mode. Programs using EXIT 1, (MONRT.) as a substitute for EXIT (to eliminate the typing of EXIT) should RELEASE all devices first.

3.1.3 Trapping

3.1.3.1 APRENB AC, or CALLI AC, 16 - APR trapping allows a user to handle any and all traps that occur while his job is running on the central processor, including illegal memory references, non-existent memory references, pushdown list overflow, arithmetic overflow, floating-point overflow, and clock flag. To enable for trapping, a APRENB AC, or CALLI AC, 16 is executed, where the AC contains the central processor flags to be tested on interrupts, as defined below:

Name	AC Bit		Trap On
AP.REN	18	400000	Repetitive enable
AP.POV	19	200000	Pushdown overflow
AP.ILM	22	20000	Memory protection violation
AP.NXM	23	10000	Nonexistent memory flag
AP.PAR	24	4000	Parity error
AP.CLK	26	1000	Clock flag
AP.FOV	29	100	Floating-point overflow
AP.AOV	32	10	Arithmetic overflow

When one of the specified conditions occurs while the central processor is in user mode, the state of the central processor is CONditioned Into (CONI) location, .JBCNI, and the PC is stored in location .JBTPC in the job data area (refer to Table 1-1 in Paragraph 1.2.1). Then control is transferred to the user trap-answering routine specified by the contents of the right half of .JBAPR, after the arithmetic and floating-point overflow flags are cleared. (However, the job is stopped if the PC is equal to the first or second instruction in the user's trap routine.) The user program must set up location .JBAPR before executing the APRENB UUO. To return control to his interrupted program, the user's trap-answering routine must execute a JRSTF @ .JBTPC which clears the bits that have been processed and restores the state of the processor.

The APRENB UUO normally enables traps for only one occurrence of any selected condition and must be re-issued after each condition of a trap. To disable this feature, set bit 18 to a 1 when executing the UUO. However, even with bit 18 = 1, clock interrupts must be re-enabled after each trap.

If the user program does not enable traps, the monitor sets the PDP-10 processor to ignore arithmetic and floating-point overflow, but enables interrupts for the other error conditions in the list above. If the user program produces such an error condition, the monitor stops the user job and prints one of the following appropriate messages:

```
?PC OUT OF BOUNDS AT USER PC addr
?ILL MEM REF AT USER PC addr
?NON-EX MEM AT USER PC addr
?PDL OV AT USER PC addr
?MEM PAR ERROR AT USER PC addr
```

The CONT and CCONT commands will not succeed after such an error.

3.1.3.2 Error Intercepting - When certain conditions occur in the program, the monitor intercepts the condition and examines location .JBINT in the job data area. Depending on the contents of this location, control is either retained by the user program or is given to the monitor for action. If this location is zero, the job is stopped and the user and possibly the operator are notified by appropriate messages, if any. If location .JBINT is non-zero, the contents is interpreted as the address of a block with the following format:

```
LOC:    XWD N, INTLOC
LOC+1:  XWD BITS, CLASS
LOC+2:  0
LOC+3:  0
```

where N is the number of words in the block (N >3).

INTLOC is the location at which the program is to be restarted.

BITS is a set of bits interpreted as follows:

If bit 0 = 1, an error message, if any, is not to be typed on the user's terminal or, in some cases, the operator's terminal.

If bit 0 = 0, an error message, if any, will be typed on the user's terminal and possibly the operator's terminal.

CLASS is a set of bits interpreted as follows:

For each type of error, CLASS has a specific bit. For a given error, the job will be interrupted if the appropriate bit is 1 and the content of LOC+2 is zero. The job will be stopped if either the appropriate bit is 0 or the appropriate bit is 1 and the content of LOC+2 is not zero. By requiring LOC+2 to be zero, the possibility of a loop occurring is prevented.

The monitor examines the CLASS bits and the contents of LOC+2 to determine if the job is to be stopped or interrupted on the particular error. If the job is interrupted, the following information is then stored in LOC+2 and LOC+3:

LOC+2	The last user PC word.
LOC+3	RH = the channel number.
	LH = the error bit as defined in CLASS (see below).

The job is then restarted at location INTLOC.

The CLASS bits are defined as follows:

Device Errors

Bit 35[1] (ER.IDV) represents device errors that can be corrected by human intervention. The appropriate message returned to the user is

DEVICE xxx OPR zz ACTION REQUESTED

where xxx is the device name, and zz is the number of the station at which the operator is located. The operator receives the message

%PROBLEM ON DEVICE xxx FOR JOB n

[1] This bit depends on FTOPRERR which is normally off in the DECsystem-1040.

where xxx is the device name, and n is the number of the job that is stopped. When the operator has corrected the error, he starts the job with the JCONT command and the message

CONT BY OPER

appears on the user's terminal to signify that the error has been corrected.

↑C Intercept

Bit 34[1] (ER.ICC) indicates a ↑C intercept. This intercept allows the user's program to process a ↑C itself instead of allowing the job to automatically return to monitor level. If this bit is 1, the job does not return to monitor level on two ↑Cs (or on one ↑C if the job is in TTY input wait), but instead traps to the user's interrupt routine. There are no messages associated with this bit. When enabled for ↑C, the program should normally exit immediately by releasing any special resources and issuing an EXIT UUO (MONRT. or CALLI 1, 12).. If the user types .CONT, the job continues.

```
        TITLE   CONCIN -- SAMPLE FOR CONTROL-C INTERCEPT

;THIS ROUTINE SHOWS HOW TO ENABLE FOR A CONTROL-C INTERCEPT
;AND HANDLE IT CORRECTLY.  THE IDEA IS TO GET THE USER TO
;MONITOR LEVEL AS QUICKLY AS POSSIBLE.

        LOC     134             ;SET POINTER IN .JBINT
        EXP     INTBLK          ;  TO THE INTERRUPT BLOCK
        RELOC

INTBLK: XWD     4,INTLOC        ;4 WORDS LONG,,PLACE TO START
        XWD     0,2             ;NO MESSAGE CONTROL,,TYPE 2 (↑C)
        Z                       ;GETS LAST USER PC
        Z                       ;LH GETS INTERRUPT TYPE

;THE INTERRUPT ROUTINE STARTS HERE

INTLOC: MOVEM   1,TEMP1         ;SAVE AC 1
        HLRZ    1,INTBLK+3      ;GET REASON FOR INTERRUPT
        CAIE    1,2             ;SEE IF CONTROL-C
        HALT    .               ;ERROR IF NOT
                        ;RELEASE ANY SPECIAL RESOURCES HERE
                        ;  BUT BE CAREFUL THAT THIS DOES NOT
                        ;  TAKE VERY LONG OR CAUSE A LOOP.
        EXIT    1,              ;RETURN TO MONITOR
        MOVE    1,INTBLK+2      ;GET RETURN PC
        EXCH    1,TEMP1         ;RESTORE AC
        PUSH    P,INTBLK+2      ;SAVE RETURN ADDRESS
        SETZM   INTBLK+2        ;CLEAR INTERUPT TO ALLOW ANOTHER ONE
        POPJ    P,              ;RETURN TO PROGRAM WHERE STOPPED

TEMP1:  Z                       ;TEMPORARY
```

[1] This bit depends on FTCCIN which is normally off in the DECsystem-1040

The following example illustrates user ↑C processing by a program which will not let users reach monitor level by means of a ↑C.

```
        LOC     134             ;SET UP .JBINT TO POINT TO
        EXP     INTBLK          ;  THE INTERRUPT BLOCK
        RELOC

INTBLK: XWD     3,INTLOC        ;3 WORDS LONG,,PLACE TO START
        XWD     Ø,2             ;NO MESSAGE CONTROL,,TYPE 2 (↑C)
        Z                       ;GETS LAST USER PC
        Z                       ;LH GETS INTERRUPT TYPE

;THE INTERRUPT ROUTINE

INTLOC: SKIPL   RENFLA          ;OK TO FAKE A REENTER?
        JRST    .+3             ;NO, CURRENT ROUTINE CANNOT BE
                                ;INTERRUPTED

        SETZM   INTBLK+2        ;YES, RE-ENABLE INTERRUPT AND GO
        JRST    REENRT          ;  TO INTERRUPT ROUTINE

        SETOM   RENSWH          ;SET FLAG TO SAY "REENTER AS SOON AS
                                ;  YOU CAN"
        PUSH    P,INTBLK+2      ;GET LAST PC. PUSH/POP
        SETZM   INTBLK+2        ;RE-ENABLE INTERRUPT
        POPJ    P,              ;GO BACK TO INTERRUPTED ROUTINE
                                ;   NOTE THAT IF A CONTROL-C IS
                                ;   TYPED AFTER THE SETZM, THE
                                ;   INTERRUPTS NEST.
```

Off-line Disk Unit

Bit 33 (ER.OFL) indicates a disk unit has dropped off-line. The operator is given the message

UNIT xxx WENT OFF-LINE (FILE UNSAFE)
PLEASE POWER DOWN AND THEN TURN IT ON AGAIN

immediately and then once every minute. The user receives the message

DSK IS OFF-LINE. WAITING FOR OPERATOR
ACTION. TYPE ↑C TO GET A HUNG MESSAGE
(IN 15 SECONDS). DONT TYPE ANYTHING TO WAIT
FOR THE OPERATOR TO FIX THE DEVICE.

If the user has a system resource, he receives the additional message:

THE SYSTEM WILL DO NO USEFUL WORK UNTIL
THE DRIVE IS FIXED OR YOU TYPE ↑C

Full File Structure

Bit 32 (ER.FUL) indicates that a file structure has filled up with data (i.e., there are no free blocks). There are no messages associated with this bit.

Exhausted Disk Quota

Bit 31 (ER.QEX) indicates that the user's disk quota has been exhausted. The user receives the message

[EXCEEDING QUOTA file structure name]

Exceeded Time Limit

Bit 30[1] (ER.TLX) indicates that the user's run time limit (as set by a previous SET TIME command) has been exceeded. This bit is used only by non-batch jobs. The user receives the message

?TIME LIMIT EXCEEDED

3.1.4 Suspending

3.1.4.1 SLEEP AC, or CALLI AC, 31 - This UUO temporarily stops the job and continues it automatically after the elapsed real-time (in seconds) indicated by the contents of the AC. There is an implied maximum of approximately 68 sec (82 sec in 50-Hz countries) or 1 min. A program that requires a longer SLEEP or HIBER time should use the HIBER UUO with no clock request and then call DAEMON, via the .CLOCK function (refer to Paragraph 3.7.2), to wake it.

3.1.4.2 HIBER AC, or CALLI AC, 72[2] - The HIBERNATE UUO allows a job to become dormant until a specified event occurs. The possible events that can wake a hibernating job are: 1) input activity from the user's TTY or any TTY INITed by this job (both line mode and character mode), 2) PTY activity for any PTY currently INITed by this job, 3) the time-out of a specified amount of sleep time, or 4) the issuance of a WAKE UUO directed at this job either by some other job with wake-up rights or by this job at interrupt level.

The HIBERNATE UUO must contain in the left half of AC the wake-condition enable bits and in the right half the number of ms for which the job is to sleep before it is awakened.

[1] This bit depends on FTTLIM which is normally off in the DECsystem-1040.

[2] This UUO depends on FTHIBWAK which is normally off in the DECsystem-1040.

The call is as follows:

```
MOVSI AC, enable bits      ;get HIBERNATE conditions
HRRI AC, sleep time        ;number of ms to sleep
HIBER AC,                  ;or CALLI AC, 72
error return
normal return
```

The HIBERNATE UUO enable condition codes are as follows:

Bits	Meaning
18-35	Number of ms sleep time. It is rounded up to an even multiple of jiffies (maximum being 2^{12} jiffies). Zero means no clock request (i.e., infinite sleep).
15-17	WAKE UUO protection code: Bit 17 (HB.RWT) = 1, project codes must match. Bit 16 (HB.RWP) = 1, programmer codes must match. Bit 15 (HB.RWJ) = 1, only this job can wake itself.
13-14	Wake on TTY input activity: Bit 14 (HB.RTC) = 1, wake on character ready. Bit 13 (HB.RTL) = 1, wake on line of input ready.
12	(HB.RPT) Wake on PTY activity since last HIBERNATE.
0	(HB.SWP) Causes job to be swapped out immediately.

An error return is given if the UUO is not implemented. The SLEEP UUO should be used in this case. A normal return is given after an enabled condition occurs.

Jobs either logged-in as [1,2] or running with the JACCT bit on can wake any hibernating job regardless of the protection code. This allows privileged programs, which are the only jobs that can wake certain system jobs, to be written.

A RESET UUO always clears the protection code and wake-enable bits for the job. Therefore, until the first HIBERNATE UUO is called, there is no protection against wake-up commands from other jobs. To guarantee that no other job wakes the job, a WAKE UUO followed by a HIBERNATE UUO with the desired protection code should be executed. The WAKE UUO ensures that the first HIBERNATE UUO always returns immediately, leaving the job with the correct protection code.

3.1.4.3 WAKE AC, or CALLI AC, 73[1] - The WAKE UUO allows one job to activate a dormant job when some event occurs. This feature can be used with Batch so that when a job wants a core dump taken, it can wake up a dump program. Also, real-time process control jobs can cause other process control jobs to run in response to a specific alarm condition. The WAKE UUO can be called for a RTTRP job running at interrupt level, thereby allowing a real-time job to wake its background portion

---

[1]This UUO depends on FTHIBWAK which is normally off in the DECsystem-1040.

quickly in order to respond to some real-time condition. (Refer to Paragraph 3.8.1.2 for the restrictions on accumulators when using the RTTRP UUO at interrupt level.)

The call is as follows:

```
MOVE AC, JOBNUM          ;number of job to be awakened
WAKE AC,                 ;or CALLI AC, 73
error return
normal return
```

An error return is given if the proper wake privileges are not specified. There is a wake bit associated with each job. If any of the enabled conditions specified in the last HIBERNATE UUO occurs, then the wake bit is set. The next time a HIBERNATE UUO is executed, the wake bit is cleared and the HIBERNATE UUO returns immediately. The wake bit eliminates the problem of a job going to sleep and missing any wake conditions.

On a normal return, the job has been awakened and has started at the location of the normal return of the HIBER UUO that caused it to become dormant.

3.2 CORE CONTROL

For various reasons, privileged jobs may desire to be locked in core so that they are never to be considered for swapping or shuffling. Some examples of these jobs are as follows:

Real-time jobs	These jobs require immediate access to the processor in response to an interrupt from an I/O device.
Display jobs	The display must be refreshed from a display buffer in the user's core area in order to keep the display picture flicker-free.
Batch	Batch throughput may be enhanced by locking the Batch job controller in core.
Performance analysis	Jobs monitoring the activities of the system need to be locked in core so that they can be invoked quickly with low overhead in order to record activities of the monitor.

3.2.1 Definitions

In swapping and non-swapping systems, unlocked jobs can occupy only the physical core not occupied by locked jobs. Therefore, locked jobs and timesharing jobs contend with one another for physical core memory. In order to control this contention, the system manager is provided with a number of system parameters as described below.

<u>Total User Core</u> is the physical core that can be used for locked and unlocked jobs. This value is equal to total physical core minus the monitor size.

<u>CORMIN</u> is the guaranteed amount of contiguous core that a single unlocked job can have. This value is a constant system parameter and is defined by the system manager at monitor generation time using

MONGEN. It can be changed at monitor startup time using the ONCE ONLY dialogue. This value can range from 0 to Total User Core.

CORMAX is the largest contiguous size that an unlocked job can be. It is a time-varying system parameter that is reduced from its initial setting as jobs are locked in core. In order to satisfy the guaranteed size of CORMIN, the monitor never allows a job to be locked in core if this action would result in CORMAX becoming less than CORMIN. The initial setting of CORMAX is defined at monitor generation time using MONGEN and can be changed at monitor startup time using the ONCE ONLY dialogue. CORMAX can range from CORMIN to Total User Core. A guaranteed amount of core available for locked jobs can be made by setting the initial value of CORMAX to less than Total User Core.

3.2.2 LOCK AC, or CALLI AC, 60[1]

This UUO provides a mechanism for locking jobs in user memory. The user may specify if the high segment, low segment, or both segments are to be locked, and whether the core is to be physically contiguous. Note that on KA10-based systems, core is always allocated contiguously, and that the job may be moved to an extremity of user core before it is locked.

A job may be locked in core if all of the following are true:

a. The job has the LOCK privilege (set from the accounting file ACCT.SYS by LOGIN).

b. The job, when locked, would not prevent another job from expanding to the guaranteed limit, CORMIN.

c. The job, when locked, would not prevent an existing job from running. Note that unlocked jobs can exceed CORMIN.

d. The job when mapped, if specifying exec mapping, would not exceed the maximum amount of exec virtual address space available for locking (KI10 only).

The call is:

```
MOVE AC, [XWD high seg. code, low seg. code]
LOCK AC                                      ;or CALLI AC, 60
error return                                 ;AC contains an error code
normal return
```

The segment codes are a series of bits which specify the way in which the high segment (LH code) and the low segment (RH code) are to be locked. The order and position of the bits in the left half correspond to the order and position of the bits in the right half; that is, to obtain the bit number for the high segment, subtract 18 from the corresponding bit for the low segment. The bits are shown below.

[1]This UUO depends on FTLOCK which is normally off in the DECsystem-1040.

Bit 17 (high segment) Bit 35 (low segment)	If 1, lock the segment in the manner indicated by the following bits. If 0, do not lock the segment; the following bits are ignored.
Bit 16 (high segment) Bit 34 (low segment)	If 0, map contiguously in the exec virtual memory (always implied on the KA10). This causes the segment to be added to the exec virtual address space so that it can be executed in exec mode. For example, this is required when exec mode real-time trapping (RTTRP) is used. On the KI10, the amount of exec virtual address space used by locked jobs is a limited resource with a defined maximum per processor. If mapping the segment would cause the maximum to be exceeded, the LKNEM% error return is given. The maximum amount available can be obtained from the CPU variable GETTAB table for each processor (GETTAB word %CVEVM). The current amount used can also be obtained from the table (%CVEVU). If 1, do not map in exec virtual memory.
Bit 15 (high segment) Bit 33 (low segment)	If 0, lock in contiguous physical memory locations (always implied on the KA10). This causes the segment to be moved and remapped, if necessary, so that its physical core is contiguous. On the KA10 system, the segment is also moved to one end of user core in order to minimize fragmentation of memory. If 1, do not attempt physical contiguity.

If the user requests a segment to be locked in contiguous physical memory, the monitor attempts to lock the segment as low in physical memory as possible. When the segment is locked below 112K, physical and virtual contiguity are equivalent, and thus in this case, virtual contiguity does not require the exec virtual memory resource to achieve contiguity.

On a KA10-based system, physical memory is always allocated contiguously and user segments are directly addressable in exec mode, and therefore, bit codes 1,3,5 and 7 are synonomous.

The setting of bits 33 and 34 (bits 15 and 16) is compatible with the implementation of the LOCK UUO on a KA10-based system. That is, code 1 is the most restrictive, so that a program coded for the KA10 system that implicitly uses these properties will also run on the KI10 system. Applications that do not require all properties can add the appropriate bits to the LOCK UUO's calling sequence.

On a normal return, the job is locked in core. If there is a high segment, the LH of AC contains its absolute address in units of pages (one page is 512 words). The value can be converted to a word address by shifting it left nine bits. If there is no high segment, the LH of AC contains zero. The RH of AC contains the absolute address of the low segment, shifted right nine bits.

On an error return, the job is not locked in core and AC either is unchanged or contains an error code. The AC is unchanged when the LOCK UUO is executed in monitors previous to the implementation of the UUO. An error code indicates the condition that prevented the job from being locked. The error codes are as follows:

Error Code	Name	Explanation
0	LKNIS%	The UUO is not included in this system because it has not been defined with MONGEN or because the appropriate feature test switch is off.
1	LKNLP%	The job does not have locking privileges, or RTTRP privileges, if required.
2	LKNCA%	If the job were locked in core, it would not be possible to run the largest existing non-locked job. (Applies only to swapping systems.)
3	LKNCM%	If the job were locked in core, it would not be possible to meet the guaranteed largest size for an unlocked job, that is, CORMAX would be less than CORMIN.
4	LKNEM%	The mode of locking requested exec virtual memory mapping but the allowable amount of exec mapping has been exhausted.

NOTE

The CORE UUO may be given for the high segment of a locked job only if it is removing the high segment from the addressing space. When the segment is locked in core, the CORE UUO and the CORE command with a non-zero argument cannot be satisfied and, therefore, always give an error return. The program should determine the amount of core needed for the execution and request this amount before executing the LOCK UUO.

Although memory fragmentation is minimized byboth the LOCK UUO and the shuffler, the locking algorithm always allows job locking, even though severe fragmentation may take place, as long as

1) all existing jobs can continue to run, and
2) at least CORMIN is available as a contiguous space (see Figure 3-1E).

Therefore, it is important that system managers use caution when granting locking privileges. The following are guidelines for minimizing fragmentation when using the LOCK UUO.

3.2.2.1 KA10 Systems - The guidelines for KA10 systems are:

a. There is no memory fragmentation if two jobs or less are locked in core.
b. There is no fragmentation if the locked jobs do not relinquish their locked status (i.e., no job terminates that has issued a LOCK UUO). In general, jobs with locking privileges should be production jobs.

c. If a job issuing a LOCK UUO is to be debugged and production jobs with locking privileges are to be run, the job to be debugged should be initiated and locked in core first, since it will be locked at the top of core. Then, the production jobs should be initiated since they will all be locked at the bottom of core. This procedure reserves the space at the top of core for the job being debugged and guarantees that there is no fragmentation as it locks and unlocks.

d. With a suitable setting of CORMIN and the initial setting of CORMAX in relation to Total User Core, the system manager can establish a policy which guarantees

   1) a maximum size for any unlocked job (CORMIN),

   2) a minimum amount of total lockable core for all jobs (Total User Core - CORMAX), and

   3) the amount of core which locked and unlocked jobs can contend for on a first-come-first-serve basis (Total User Core - initial CORMAX + CORMIN).

3.2.2.2 Core Allocation Resource - Because routines that lock jobs in core use the swapping and core allocation routines, they are considered a sharable resource. This resource is the semipermanent core allocation resource (mnemonic=CA). When a job issues a LOCK UUO and the system is currently engaged in executing a LOCK UUO for another job, the job enters the queue associated with the core allocation resource. Because a job may share a queue with other jobs and because swapping and shuffling may be required to position the job to where it is to be locked, the actual execution time needed to complete the process of locking a job might be on the order of seconds.

When it has been established that a job can be locked, the low segment number and the high segment number (if any) are stored as flags to activate the locking routines when the swapper and shuffler are idle. The ideal position for the locked job is also stored as a goal for the locking routines. In KA10 swapping systems, the ideal position is always achieved to guarantee minimum fragmentation. In nonswapping systems, minimum fragmentation is achieved only if the ideal position does not contain an active segment (see Figure 3-1).

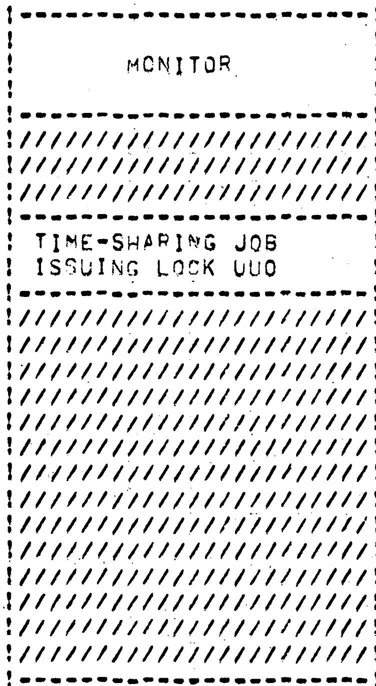
In swapping systems, after the job is locked in core, the locking routine determines the size of the new largest contiguous region available to unlocked jobs. This value will be greater than or equal to CORMIN. If this region is less than the old value of CORMAX, then CORMAX is set equal to the size of the new reduced region. Otherwise, CORMAX remains set to its old value.

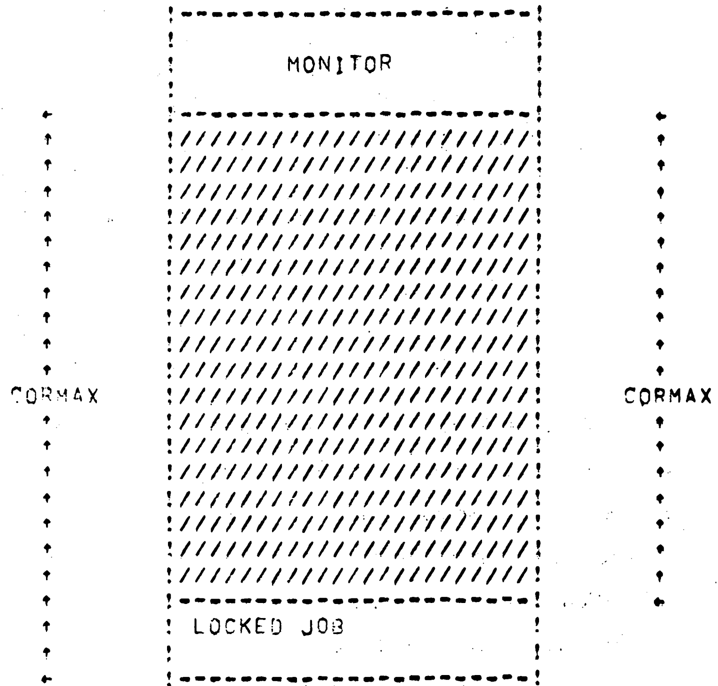
3.2.2.3 UNLOK. AC, or CALLI AC, 120[1] - This UUO provides a mechanism for a job to unlock itself without doing a RESET UUO. The user can specify if one or both segments are to be unlocked. The call is:

```
        MOVSI AC, 1         ;if high segment is to be unlocked
        MOVSI AC, 0         ;if no high segment, or if high segment
                            ;is not to be unlocked
        HRRI AC, 1          ;if low segment is to be unlocked.
        HRRI AC, 0          ;if low segment is not to be unlocked.
        UNLOK. AC,          ;or CALLI AC, 120
        error return
        normal return
```

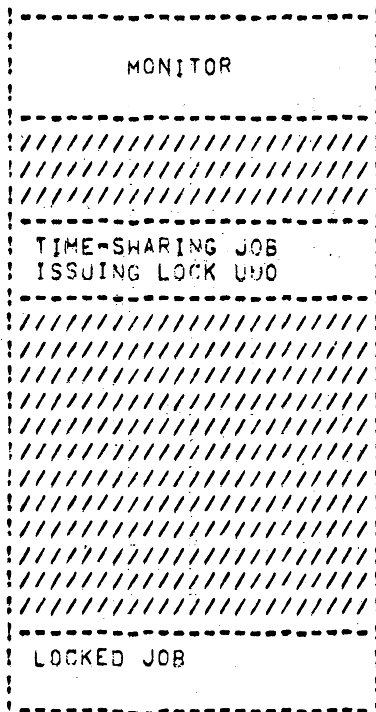
[1] This UUO depends on FTLOCK which is normally off in the DECsystem-1040.

A)





B)



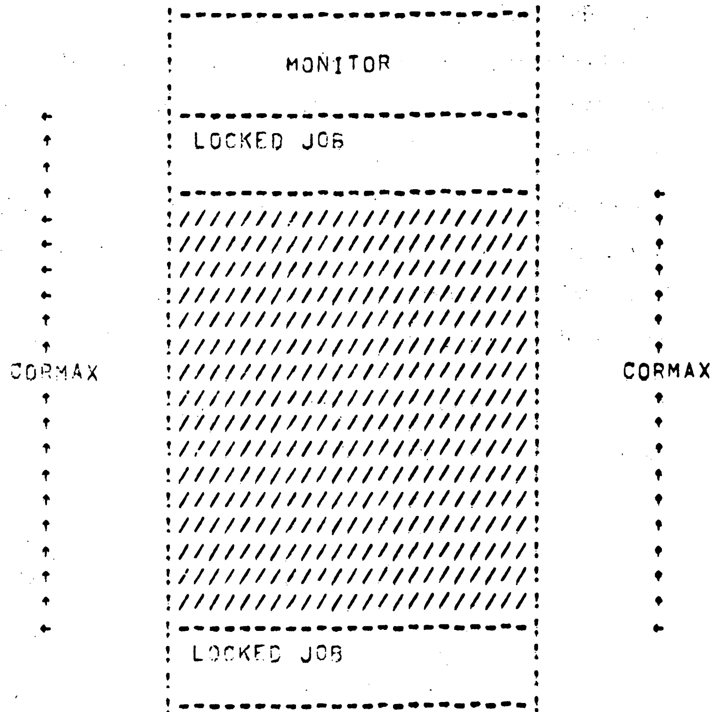


Figure 3-1 Locking Jobs In Core on KA10 Systems

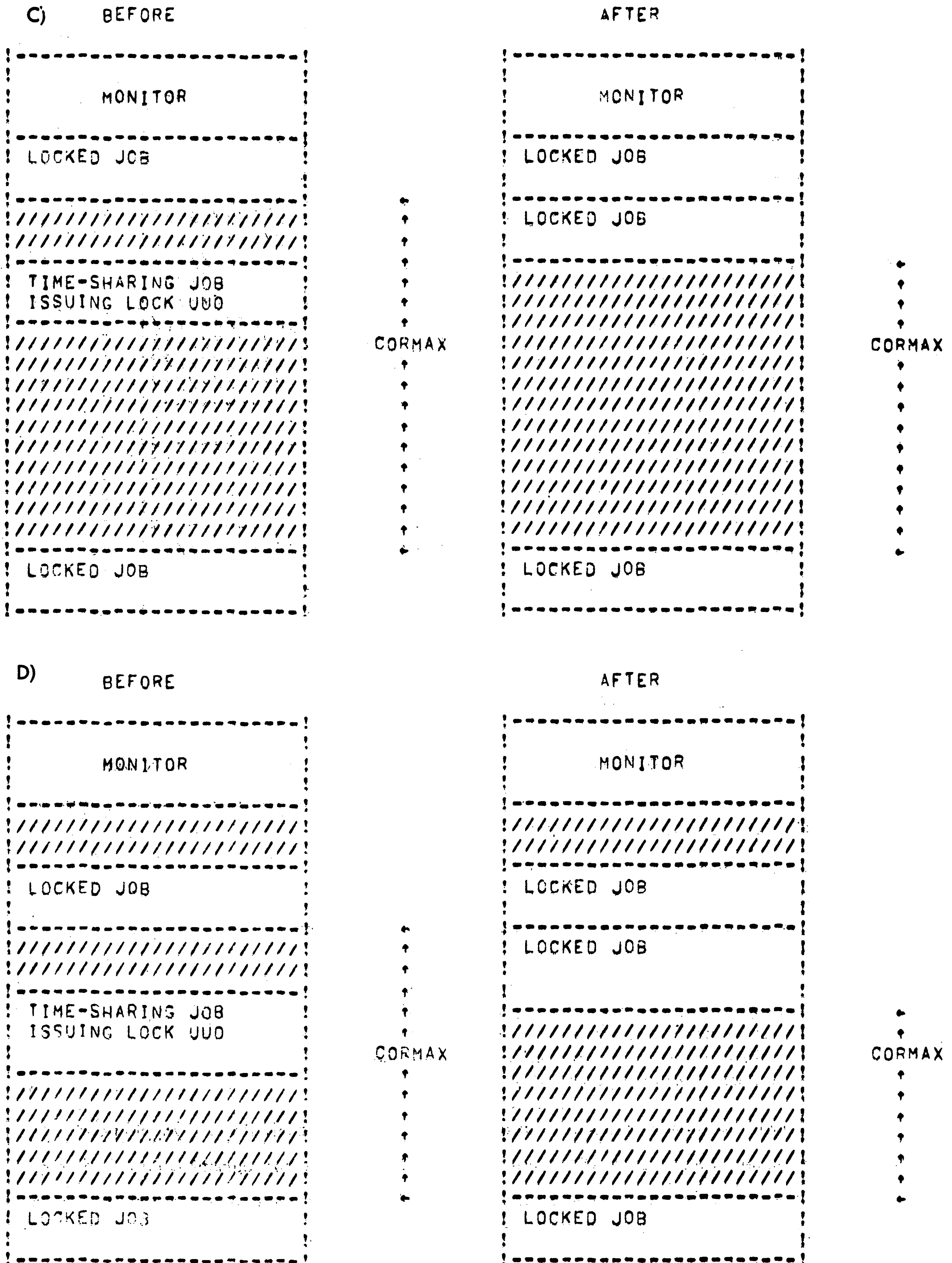


Figure 3-1 Locking Jobs In Core on KA10 Systems (Cont)

E) Unlikely Fragmentation Case

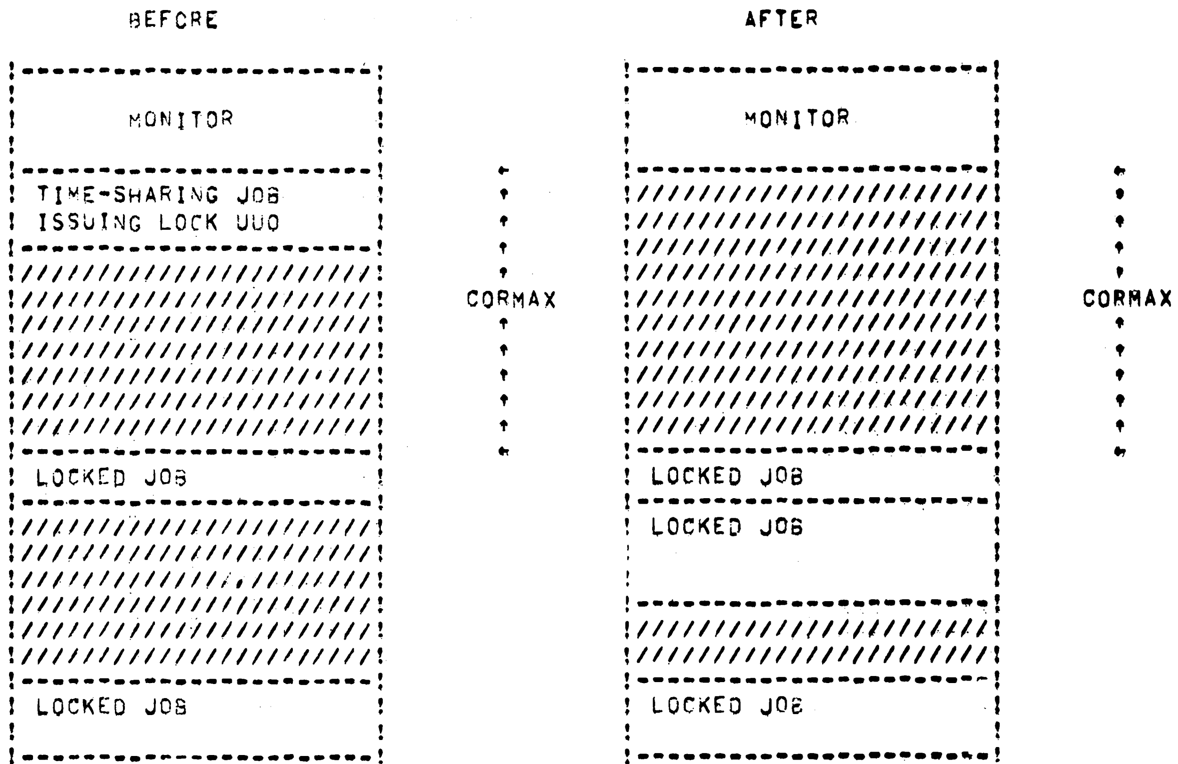


Figure 3-1 Locking Jobs In Core on KA10 Systems (Cont)

An error return is given if the UUO is not implemented. If this is the case, a job can relinquish its locked status when either the user program executes an EXIT or RESET UUO, or the monitor performs an implicit RESET for the user. Implicit RESETS occur when

a. The user program issues a RUN UUO, or

b. The user types any of the following monitor commands: R, RUN, GET, SAVE, SSAVE, CORE 0, and any system program-invoking command.

NOTE

If several jobs are sharing a locked high segment, the high segment is unlocked only when the SN%LOK bit is turned off for all jobs sharing the segment (i.e., when all jobs which executed the LOCK UUO have performed the unlock function) (refer to GETTAB table 14).

On a normal return, the segment (or job) is unlocked and becomes a candidate for swapping and shuffling. Any meter points (METER.UUO) are deactivated and, if the low segment is unlocked, any real-time devices are RESET. CORMAX is increased to reflect the new size of the largest contiguous region available to unlocked jobs. However, CORMAX is never set to a greater value than its initial setting.

3.2.3 CORE AC, or CALLI AC, 11

This UUO provides a user program with the ability to expand and contract its core size as its memory requirements change. To allocate core in either or both segments, the left half of AC is used to specify the highest user address to be assigned to the high segment and the right half is used to specify the highest user address in the low segment. The monitor will assign the smallest amount of core which will satisfy the request. If the left half of AC contains 0, the high segment core assignment is not changed. If the left half of AC is non-zero and is either less than 400000 or the length of the low segment, whichever, is greater, the high segment is eliminated. If this is executed from the high segment, an illegal memory error message is printed when the monitor attempts to return control to the illegal address.

A RH of 0 leaves the low segment core assignment unaffected. The monitor clears new core before assigning it to the user; therefore, privacy of information is ensured.

The error return is given if:

1) The LH is greater than or equal to 400000 and the system does not have a two-segment capability.
2) The LH is greater than or equal to 400000 and the user has been meddling without write access privileges (refer to Paragraph 6.2.3).
3) The LH and the RH are both zero.

In swapping systems, this programmed operator returns the maximum number of 1K core blocks (all of core minus the monitor, unless an installation chooses to restrict the amount of core) available to the user. By restricting the amount of core available to users, the number of jobs in core simultaneously is increased. In nonswapping systems, the number of free and dormant 1K blocks is returned; therefore, the CORE UUO and the CORE command return the same information.

For compatibility, the KI10 also returns the number of 1K blocks available even though core is allocated in 512-word pages. The value returned is truncated to the nearest multiple of 1K (e.g., if 21 pages are available, the value returned in 10K). If it is necessary to obtain the exact amount of core available in units of pages, the user can examine the monitor location CORMAX (in GETTAB table 12) with the GETTAB UUO (refer to Paragraph 3.6.3.4). CORMAX is the maximum number of <u>words</u> available to the user and thus can be converted to either pages or K.

The call is:

```
MOVE AC [XWD HIGH ADR or 0, LOW ADDR or 0]
CORE AC,                                    ;or CALLI AC, 11
error return
normal return
```

The CORE UUO re-assigns the low segment (if RH is non-zero) and then re-assigns the high segment (if LH is non-zero). If the sum of the new low segment and the old high segment exceeds the maximum amount of core allowed to a user, the error return is given, the core assignment is unchanged, and the maximum core available to the user for high and low segments (in 1K blocks) is returned in the AC. In a nonswapping system, the number of free and dormant 1K blocks is returned.

If the sum of the new low segment and the new high segment exceeds the maximum amount of core allowed to a user, the error return is given, the new low segment is assigned, the old high segment remains, and the maximum core available to the user in 1K blocks is returned in the AC. Therefore, to increase the low segment and decrease the high segment at the same time, two separate CORE UUOs should be used to reduce the chances of exceeding the maximum size allowed to a user job. If the new low segment extends beyond 377777, the high segment shifts up into the virtual addressing space instead of being overlaid. If a long low segment is shortened to 377777 or less, the high segment shifts from the virtual addressing space to 400000 instead of growing longer or remaining where it was. If the high segment is a program, it does not execute properly after a shift unless it is a self-relocating program in which all transfer instructions are indexed.

If the high segment is eliminated by a CORE UUO, a subsequent CORE UUO, in which the LH is greater than 400000, will create a new, nonsharable segment rather than re-establishing the old high segment. This segment becomes sharable after it has been:

a. Given an extension .SHR.
b. Written onto the storage device.
c. Closed so that a directory entry is made.
d. Initialized from the storage device by GET, R, or RUN commands or RUN or GETSEG UUOs.

The loader and the SAVE and GET commands use the above sequence to create and initialize new sharable segments.

A user program which expands core should compare its highest desired address with its highest legal address obtained from the Job Data Area location .JBREL (refer to Chapter 1). If the desired address is greater than the highest legal address, the program should execute a CORE UUO for the new desired address (not for the highest old legal address plus 512 or 1024). The monitor then updates .JBREL by the number of words in its basic core allocation unit (i.e., 1024 words on the KA10 processor or 512 words on the KI10 processor). Subsequent compares of the desired address and the highest legal address do not cause a CORE UUO until the next increase of core is required. If used this way, a CORE UUO will execute on both the KA10 and KI10 processors and will require less monitor CPU time because the number of CORE UUOs needed will be minimized.

The following example illustrates the method for obtaining core only when needed.

```
;SUBROUTINE TO GET CORE ONLY WHEN NEEDED
;CALL:   MOVE    T1, HIGHEST DESIRED ADDRESS
;        PUSHJ   P, CHKCOR
;        RETURN HERE UNLESS NO MORE CORE

CHKCOR: CAMLE   T1,.JBREL##         ;GREATER THAN HIGHEST LEGAL ADDRESS?
        POPJ    P,                  ;NO, PRESENT CORE BIG ENOUGH.
        CORE    T1,                 ;YES, GET NEXT INCREMENT OF CORE.
          JRST   ERROR              ;NOT AVAILABLE.
        POPJ    P,                  ;NEXT INCREMENT ASSIGNED.
```

3.2.4 SETUWP AC, or CALLI AC, 36

This UUO allows a user program to set or clear the hardware user-mode write protect bit and to obtain the previous setting. It must be used if a user program is to modify the high segment.
The call is:

```
SETUWP AC,              ;OR CALLI AC, 36
error return
normal return
```

If the system has a two-register capability, the normal return will be given unless the user has been meddling without write privileges, in which case an error return will be given. A normal return is given whether or not the program has a high segment, because the reentrant software is designed to allow users to write programs for two-register machines, which will run under one-register machines. Compatibility of source and relocatable binary files is, therefore, maintained between one-register and two-register machines.

If the system has a one-register capability, the error return (bit 35 of AC=0) is given. This error return allows the user program to find out whether or not the system has a two-segment capability. The user program specifies the setting of the user-mode write protect bit in bit 35 of AC (write protect = 1, write privileges = 0). The previous setting of the user-mode write protect bit is returned in bit 35 of AC, so that any user subroutine can preserve the previous setting before changing it. Therefore, nested user subroutines, which either set or clear the bit, can be written, provided the subroutines save the previous value of the bit and restore it on returning to its caller.

3.3 SEGMENT CONTROL

3.3.1 RUN AC, or CALLI AC, 35

This UUO has been implemented so that programs can transfer control to one another. Both the low and high segments of the user's addressing space are replaced with the program being called.

The call is:

```
MOVSI AC, starting address increment
HRRI AC, adr of six-word argument block
RUN AC,                                       or CALLI AC, 35
error return (unless HALT in LH)
[normal return is not here, but to start-
ing address plus increment of new program]
```

The arguments contained in the six-word block are:

```
E:  SIXBIT/logical device name/
    SIXBIT/filename/                  ;for either or both high and low files
    SIXBIT/ext. for low file/         ;if LH = 0, .LOW is assumed if high
                                      segment exists, .SAV is assumed if
                                      high segment does not exist.
    0
    XWD proj. no., prog. no.          ;if = 0, use current user's proj, prog
    XWD 0, optional core              ;RH = new highest user address to be
           assignment                 assigned to low segment.
                                      LH is ignored rather than setting high
                                      segment.
```

A user program usually will specify only the first two words and set the others to 0. The RUN UUO destroys the contents of all of the user's ACs and releases all the user's I/O channels; therefore, arguments or devices cannot be passed to the next program.

The RUN UUO to certain system programs (e.g., LOGIN, LOGOUT) automatically sets the appropriate privileged bits (JACCT and JLOG). These bits are not set (or are turned off if they were set) for programs that are not privileged programs from device SYS or for programs whose starting address offset is greater than 1. Assigning a device as SYS does not cause these bits to be set.

The RUN UUO clears all of core. However, programs should not count on this action, and must still initialize core to the desired value to allow programs to be restarted by a ↑C, START sequence without having to do I/O.

Programs on the system library should be called by using device SYS with a zero project-programmer number instead of device DSK with the project-programmer number [1, 4]. The extension should also be 0 so that the calling user program does not need to know if the called system program is reentrant or not.

The LH of AC is added to and stored in the starting address (.JBSA) of the new program before control is transferred to it. The command ↑C followed by the START command restarts the program at the location specified by the RUN UUO, so that the user can start the current system program over again.

The user is considered to be meddling with the program (refer to Paragraph 3.3.5) if the LH of AC is not 0 or 1 unless the program being run is execute-only for this job. In this case, the offset is treated as 0.

Programs accept commands from a terminal or a file, depending on how they were started, due to control by the program calling the RUN UUO. The following convention is used with all of DEC's standard system programs: 0 in LH of AC means type an asterisk and accept commands from the terminal. A 1 means accept commands from a command file, if it exists; if not, type an asterisk and accept commands from the terminal. The convention for naming system program command files is that

the filename be of the form

###III.TMP

where III are the first three (or fewer if three do not exist) characters of the name of the program doing the LOOKUP, and ### is the decimal character expansion (with leading zeroes) of the binary job number. The job number is included to allow a user to run two or more jobs under the same project-programmer number. For example,

009PIP.TMP
039MAC.TMP

Decimal numbers are used so that a user listing his directory can see the same number as the PJOB command types. These command files are temporary and may, therefore, be deleted by the KJOB program (refer to KJOB command and Appendix C in DECsystem-10 Operating System Commands).

At times it is necessary to remember the arguments that a user typed in to invoke a program (i.e, the arguments on a GET or RUN command). For example, the COBOL program needs these arguments in order to GETSEG the next overlay from the same place. In all monitors, when the program is first started, this information can be obtained from the following accumulators:

AC0 (.SGNAM) contains the filename.
AC7 (.SGPPN) contains the directory name.
AC11 (.SGDEV) contains the device name.
AC17 (.SGLOW) contains the extension of the low segment.

Note that the starting address should be changed by the program so that a ↑C, START sequence will not destroy the remembered arguments in the ACs. This information should not be used when desiring to save the current segment name (GETTAB should be used in this case), but rather when obtaining the call arguments before calling the next segment.

The RUN UUO can give an error return with an error code in AC if any errors are detected; thus, the user program may attempt to recover from the error and/or give the user a more informative message on how to proceed. Some user programs do not go to the bother of including error recovery code.

The monitor detects this and does not give an error return if the LH of the error return location is a HALT instruction. If this is the case, the monitor simply prints its standard error message for that type of error and returns the user's terminal to monitor mode. This optional error recovery procedure also allows a user program to analyze the error code received and then execute a second RUN UUO with a HALT if the error code indicates an error for which the monitor message is sufficiently informative or one from which the user program cannot recover.

The error codes are an extension of the LOOKUP, ENTER, and RENAME UUO error codes and are defined in the S.MAC monitor file. Refer to Appendix E for an explanation of the error codes.

The monitor does not attempt an error return to a user program after the high or low segment containing the RUN UUO has been overlaid. The UUO should be placed in the low segment in case the error is discovered after the high segment has been released.

To successfully program the RUN UUO for all size systems and for all system programs with a size that is not known at the time the RUN UUO is coded, it is necessary to understand the sequence of operations the RUN UUO initiates. Assume that the job executing the RUN UUO has both a low and a high segment. (It can be executed from either segment; however, fewer errors can be returned to the user if it is executed from the high segment.)

The sequence of operations for the RUN UUO is as follows:

1. Does a high segment already exist with desired name?
   If yes, go to 30.
   INIT and LOOKUP filename .SHR. If not found, go to 10.
   Read high file into top of low segment by extending it. (Here the old segment and new high segment and old high segment together may not exceed the maximum user core legally available to this job at the time of the UUO nor may it cause the total amount of virtual core assigned to all users to exceed the size of the swapping space.)

   REMAP the top of low segment replacing old high segment in logical addressing space.
   If high segment is sharable (.SHR) store its name so others can share it.
   Always go to 40 or return to user if GETSEG UUO.

10. LOOKUP filename .HGH. If not found, go to 41 or error return to user if GETSEG UUO.
    Read high file into top of low segment by extending it. (The old low segment and new high segment and old high segment together may not exceed the maximum user core legally available to this job at the time of the UUO nor may it cause the total amount of virtual core assigned to all users to exceed the size of the swapping space.)
    Check for I/O errors. If any, error return to user unless HALT in LH of return.
    Go to 41.

30. Remove old high segment, if any, from logical addressing space.
    Place the sharable segment in user's logical addressing space. Go to 40 or return to user if GETSEG UUO.

35. Remove old high segment, if any, from logical addressing space.
    (Go to 41).

40. Copy vestigial job data area into job data area.
    Does the new high segment have a low file
    (LH of .JBCOR >137)?
    If not, go to 45.

41. LOOKUP filename .SAV or .LOW or user specified extension. Error if not found. Return to user if there is no HALT in LH of error return, provided that if the CALL is from the high segment, it is still the original high segment and has not been removed from the user's addressing space. Otherwise, the monitor prints one of the following error messages:

?NOT A SAVE FILE
?filename .SAV NOT FOUND
?TRANSMISSION ERROR
?LOOKUP FAILURE n
?nK OF CORE NEEDED
?NO START ADR

and stops the job.
Reassign low segment core according to size of file or user specified core argument, whichever is larger. Previous low segment is overlaid. Read low file into beginning of low segment. Check for I/O errors. If there is an error print error message and do not return to user. If there are no errors, perform START.

45. Reassign low segment core according to larger of user's core argument or argument when file saved (RH of .JBCOR).

NOTE

To be guaranteed of handling the largest number of errors, the cautious user should remove his high segment from high logical addressing space (use CORE UUO with a one in LH of AC). The error handling code should be put in the low segment along with the RUN UUO and the size of the low segment reduced to 1K. A better idea would be to have the error handling code written once and put in a seldom used (probably nonsharable) high segment, which could be gotten in high segment using GETSEG UUO (see below) when an error return occurs to low segment on a RUN UUO.

3.3.2 GETSEG AC, or CALLI AC, 40

This UUO has been implemented so that a high segment can be initialized from a file or shared segment without affecting the low segment. It is used for shared data segments, shared program overlays, and run-time routines such as FORTRAN or COBOL object time systems. This programmed operator works exactly like the RUN UUO with the following exceptions:

a. No attempt is made to read a low file.

b. The accumulators are not preserved. The only change made to JOBDAT is to set the left half of .JBHRL to 0 (a SAVE command then saves all of the high segment) and the right half to the highest legal user address.

c. If an error occurs, control is returned to the location of the error return, unless the left half of the location contains a HALT instruction.

d. On a normal return, the control is returned to two locations following the UUO, whether it is called from the low or high segment. It should be called from the low segment unless the normal return coincides with the starting address of the new high segment.

e. User channels 1 through 17 are not released so the GETSEG UUO can be used for program overlays, such as the COBOL compiler. Channel 0 is released because it is used by the UUO.

f. .JBSA and .JBREN are zeroed if they point to a high segment that is being removed. This produces the message:

?NO START ADDRESS

if a START or REENTER command is given.

Refer to steps 1 through 30 of the RUN UUO description (Paragraph 3.3.1) for details of GETSEG UUO operation.

3.3.3 REMAP AC, or CALLI AC, 37

This UUO takes the top part of a low segment and remaps it into the high segment. The previous high segment (if any) will be removed from the user's addressing space. The new low segment will be the previous low segment minus the amount remapped.

The call is:

```
MOVEI AC, desired highest adr in low segment
REMAP AC,                                        ;or CALLI AC, 37
error return
normal return
```

The monitor rounds up the address to the nearest core allocation unit of either 1024_{10} (2000_8) words on KA10-based systems or 512_{10} (1000_8) words on KI10-based systems. If the argument exceeds the length of the low segment, remapping will not take place, the high segment will remain unchanged in the user's addressing space, and the error return will be taken. The error return will also be taken if the system does not have a two-register capability. The content of AC is unchanged. The content of .JBREL (refer to Paragraph 1.2.1) is set to the new highest legal user address in the low segment. The LH of .JBHRL is set to 0 (a SAVE command then saves all of the high segment) and the RH is set to the highest legal user address in the high segment (401777 or greater or 0). The hardware relocation will be changed, and the user-mode write protect bit will be set.

This UUO is used by the LOADER to load reentrant programs, which make use of all of physical core. Otherwise, the LOADER might exceed core in assigning additional core and moving the data from the low to the high segment with a BLT instruction. The GET command also uses this UUO to perform I/O into the low segment instead of the high segment.

3.3.4 Testing for Sharable High Segments

Occasionally, it is desirable for a program to determine whether its high segment is sharable. If the high segment is sharable, the program may decide not to modify itself. The following code tests the high segment whether or not 1) the system has a high segment capability or 2) the job has a high segment.

```
HRROI   T, .GTSGN           ;see if high segment is sharable
GETTAB  T,                  ;look at monitor .GTSGN table
JRST    .+2                 ;table or UUO not present
TLNN    T, (SN%SHR)         ;is sharable bit on ?
JRST    NOTSHR              ;no, go ahead and modify here
                            ;if high segment is sharable.
```

3.3.5 Modifying Shared Segments and Meddling

A high segment is usually write-protected, but it is possible for a user program to turn off the user write-protect bit or to increase or decrease a shared segment's core assignment by using the SETUWP or CORE UUO. These UUOs are legal from the high or low segment if the sharable segment has not been "meddled" with, unless the user has write privileges for the file that initialized the high segment. Even the malicious user can have the privilege of running such a program, although he does not have the access rights to modify the file used to initialize the sharable segment.

Meddling is defined as any of the following, even if the user has privileges to write the file which initialized the sharable segment.

a. START or CSTART commands with an argument.
b. DEPOSIT command in the low or high segment.
c. RUN UUO with anything other than a 0 or 1 in LH of AC as a starting address increment.
d. GETSEG UUO.

It is not considered meddling to perform any of the above commands or UUOs with a nonsharable program. It is never considered meddling to type ↑C followed by START (without an argument), CONT, CCONT, CSTART (without an argument), REENTER, DDT, SAVE, or E command.

When a sharable program is meddled with, the monitor sets the meddle bit for the user. An error return is given when the clearing of the user write-protect bit is attempted with the SETUWP UUO or when the reassignment of core for the high segment (except to remove it completely) is attempted with the CORE UUO. An attempt to modify the high segment with the DEPOSIT command causes the message

OUT OF BOUNDS

to be printed. If the user write-protect bit was not set when the user meddled, it will be set to protect the high segment in case it is being shared. The command and the two UUOs are allowed in spite of meddling, if the user has the access privileges to write the file which initialized the high segment.

A privileged programmer is able to supersede a sharable program, which is in the process of being shared by a number of users. When a successful CLOSE, OUTPUT, or RENAME UUO is executed for a file with the same directory name and filename (previous name if the RENAME UUO is used) as the segment being shared, the name of the segment is set to 0. New users do not share the older version, but they do share the newer version. This requires the monitor to read the newly created file only once to initialize it. The monitor deletes the older version when all users are finished sharing it.

Users with access privileges are able to write programs that access sharable data segments via the GETSEG UUO (which is meddling) and then turn off the user write-protect bit using SETUWP UUO. With DECtape, write privileges exist if it is assigned to the job (cannot be a system tape) or is not assigned to any job and is not a system tape.

When control can be transferred only to a small number of entry points (two), which the shared program is prepared to handle, then the shared program can do anything it has the privileges to do, although the person running the program does not have these privileges.

The ASSIGN (and the DEASSIGN, DISMOUNT/REMOV, FINISH, KJOB commands if the device was previously assigned by console) command clears all shared segment names currently in use, which were initialized for the device, if the device is removable (DTA, MTA). Otherwise, new users could continue to share the old segment indefinitely, even if a new version were mounted on the device. Therefore, it is possible to update the library during regular timesharing, if the programmer has access privileges.

3.4 PROGRAM AND PROFILE IDENTIFICATION

3.4.1 SETNAM AC, or CALLI AC, 43

This UUO is used by the LOADER. The content of AC contains a left-justified SIXBIT program name, which is stored in a monitor job table. The information in the table is used by the SYSTAT program (refer to Table 3-1 in Paragraph 3.6.3.3). This UUO clears the "SYS:" program bit JB.LSY (used by Batch), clears the execute-only bit, and outputs a SET WATCH VERSION number (refer to DECsystem-10 Operating System Commands).

3.4.2 SETUUO AC, or CALLI AC, 75[1]

This UUO is used to set various system or job parameters. To set system parameters, the user must be logged in under [1, 2] or the job must be running with the JACCT bit set. Refer to the Specifications section of the DECsystem-10 Software Notebooks for a complete description of the privileged functions.

The contents of AC contain a function code in the left half and an argument in the right half. The call is:

```
        MOVE AC, [XWD function, argument]
        SETUUO AC,                              ;or CALLI AC, 75
        error return
        normal return
```

[1] This UUO depends on FTSET which is normally off in the DECsystem-1040. If FTSET is on, individual functions depend on the other feature test switches as noted in the text.

The functions and arguments are as follows:

Function	Name	Argument
0	.STCMX	CORMAX. Privileged function.
1	.STCMN	CORMIN. Privileged function.
2	.STDAY	DAYTIME. Privileged function (FTSEDAT).
3	.STSCH	SCHED. Privileged function.
4	.STCDR	CDR (input name counter for this job). Not a privileged function. If AC is non-zero, the content is the same as the next input name. If AC is 0, the current counter is returned in AC (FTSPL).
5	.STSPL	SPOOL for this job. Not a privileged function unless the user is unspooling devices. Bits are bits 31-35 of .GTSPL (FTSPL). Bit 35 JS.PLP line printer spooling Bit 34 JS.PPL plotter spooling Bit 33 JS.PPT paper tape punch spooling Bit 32 JS.PCP card punch spooling Bit 31 JS.PCR card reader spooling
6	.STWTC	WATCH for this job. Not a privileged function. Bits are bits 1-6 of .GTWCH (FTWATCH). Bit 1 JW.WDY watch time of day Bit 2 JW.WRN watch run time Bit 3 JW.WWT watch wait time Bit 4 JW.WDR watch disk reads Bit 5 JW.WDW watch disk writes Bit 6 JW.WVR watch version numbers.
7	.STDAT	DATE. Privileged function (FTSEDAT).
10	.STOPR	OPR. Privileged function.
11	.STKSY	KSYS. Privileged function (FT5UUO).
12	.STCLM	CORE limit. Privileged function (FTTLIM).
13	.STTLM	TIME limit for this job. Privileged function (FTTLIM).
14	.STCPU	CPU specification for this job. The following bits select the CPU on which the job is allowed to run. Bit 35 SP.CR0 CPU0 Bit 34 SP.CR1 CPU1 Bit 33 SP.CR2 CPU2 Bit 32 SP.CR3 CPU3 Bit 31 SP.CR4 CPU4 Bit 30 SP.CR5 CPU5
15	.STCRN	CPU runnability. Privileged function.
16	.STLMX	LOGMAX. Privileged function.

Function	Name	Argument
17	.STBMX	BATMAX. Privileged function.
20	.STBMN	BATMIN. Privileged function.
21	.STDFL	DSKFUL for this job. Not a privileged function. An argument of 0 (.DFPSE) causes a pause and an argument of 1 (.DFERR) causes an error when the disk is full or the user's quota is exceeded. The current setting can be determined by issuing an argument other than 0 or 1. The value returned is either 0 or 1 depending on whether PAUSE or ERROR is set. The initial setting is ERROR.

The error return is given if 1) the UUO is not implemented, 2) the user does not have the correct privileges for the function specified, or 3) the argument specified is invalid.

On a normal return, AC remains unchanged.

3.4.3 LOCATE AC, or CALLI AC, 62[1]

This UUO is used to change the logical station associated with the user's job. The call is:

```
MOVEI AC, station number
LOCATE AC,                        ;or CALLI AC, 62
error return
normal return
```

The station number requested is contained in AC as follows:

- -1 changes the job's location to the physical station of the job's controlling terminal.
- 0 changes the job's location to the central station.
- n changes the job's location to remote station n.

The normal return is taken if the UUO is implemented, the station is defined, and the station is in contact. Subsequent generic device specifications are at the new station. The error return is taken if the UUO is not implemented or the specified station is illegal or not in contact.

3.5 INTER-PROGRAM COMMUNICATION

3.5.1 TMPCOR AC, or CALLI AC, 44[2]

This UUO allows a job to leave several short files in core from the running of one user program or system program to the next. These files are referenced by a three-character filename and are unique

[1] This UUO depends on FTREM which is normally off in the DECsystem-1040.

[2] This UUO depends on FTTMP which is normally off in the DECsystem-1040.

to each job. All files are deleted when the job is killed. This system of temporary storage improves response time and reduces the number of disk operations. If this UUO fails, the file specification DSK:nnnNAM.TMP, where nnn is the job number and NAM is the three-character filename, should be used for temporary disk storage.

Each temporary file appears to the user as one dump mode buffer. The actual size of the file, the number of temporary files a user can have, and the total core a user can use for temporary storage are parameters determined at MONGEN time. All temporary files reside in a fixed area, but the space is dynamically allocated among different jobs and several different files for any given job.

The call is:

```
        MOVE AC, [XWD CODE, BLOCK]
        TMPCOR AC,                      ;or CALLI AC, 44
        error return
        normal return
        .
        .
        .
BLOCK:  XWD NAME, 0                     ;NAME is filename
        IOWD BUFLEN, BUFFER             ;user buffer area
                                        ;(zero for no buffer)
```

The AC must be set by the user program prior to execution of the UUO and is changed by the UUO on return to a value that depends on the particular function performed. Functions of the TMPCOR UUO are presented in the following paragraphs.

3.5.1.1 CODE = 0 (.TCRFS), Obtain Free Space - This is the only form of the UUO that does not use a two-word parameter block and, therefore, the contents of AC are ordinarily set to 0. A normal return is given (unless the UUO is not implemented), and the number of the free words available to the user is returned in AC.

3.5.1.2 CODE = 1 (.TCRRF), Read File - If the specified file is not found, the number of free words available for temporary files is returned in AC and the error return is taken. If the specified file is found, the length of the file in words (that is, the length in BUFLEN when writing the file rounded up to the next highest multiple of four) is returned in AC, and as much of the file as possible is copied into the user's buffer. The user may check for truncation of the file by comparing the contents of AC with BUFLEN.

3.5.1.3 CODE = 2 (.TCRDF), Read and Delete File - This function is similar to CODE = 1, except that if the specified file is found, it is deleted and its space is reclaimed.

3.5.1.4 CODE = 3 (.TCRWF), Write File - If a file exists with the specified name, it is deleted and its space reclaimed. The requested size of the file is the value in BUFLEN rounded up to the next highest multiple of four. If there is enough space

a. The file is written.

b. The number of remaining blocks is returned in AC.

c. The normal return is taken.

If there is not enough space to completely write the file

a. The file is not written.

b. The number of free words available to the user is returned in AC.

c. The error return is taken.

3.5.1.5 CODE = 4 (.TCRRD), Read Directory - The number of different files in the temporary file area of the job is returned in AC. An entry is made for each file in the user's buffer area until either there is no more space or all files have been listed. The error return is never taken. The user may check for truncation of the entries by comparing the contents of AC with BUFLEN. The format of a directory entry is as follows:

XWD NAME, SIZE

where NAME is the filename and SIZE is the file length in words.

3.5.1.6 CODE = 5 (.TCRDD), Read and Clear Directory - This function is similar to CODE = 4, except that any files in the temporary storage area of the job are deleted and their space is reclaimed. This UUO is used by the LOGOUT program.

3.6 ENVIRONMENTAL INFORMATION

3.6.1 Timing Information

The 5.05 and later monitors use two time and two date standards. The time accounting is performed by two clocks. The APR clock, driven by the power source frequency (60 Hz in North America, 50 Hz in most other countries), is accurate over long periods of time. For this reason, it is used to keep the time of day, e.g., for the TIMER UUO. It can also be used for runtime accounting measurement (i.e., keeping track of the processor time each job uses). However, there will be some loss of accuracy since the time intervals in which a job runs are often less than the period of the APR clock.

The DK10 clock, a 100000 Hz clock, is accurate over short periods of time. It is used to perform runtime accounting, and thereby achieves greater accuracy than the APR clock.

The traditional DECsystem-10 date (returned with the DATE UUO) is a 15-bit integer. This integer is incremented by 1 each day, by 31 each month (regardless of the actual number of days in the month), and by 12*31 each year (also regardless of the actual number of days in the year). This date format is easy to resolve into year-month-day; however, the difference between two dates in this format is not necessarily the actual number of days between them.

A universal date-time standard (GETTAB table 11, item 53) is also used in which the left half of the word is the date and the right half is the time. The date is uniformly incremented each day (at midnight, Greenwich Mean Time) with 1 being November 18, 1858. This date is consistent with the Smithsonian Astronomical Date Standard and other computer systems. The time is a fraction of a day. Thus, the 36-bit quantity is in units of days with a binary point between the left and right halves. The resolution is approximately 1/3 of a second; that is, the least significant bit (bit 35) represents approximately 1/3 of a second. Since the time is Greenwich Mean Time (GMT), all installations have a date-time reference which is independent of location and local time conventions.

For convenience, the monitor maintains a set of GETTAB values which gives the local date and time in terms of year, month, day, hours, minutes, and seconds (GETTAB table 11, items 56-63).

3.6.1.1 DATE AC, or CALLI AC, 14 - A 15-bit binary integer computed by the formula

date=((year-1964)x12+(month-1))x31+day-1

represents the date.
This integer representation is returned right justified in AC.

3.6.1.2 TIMER AC,or CALLI AC, 22 - This UUO returns the time of day, in clock ticks (jiffies), right justified in AC. A jiffy is 1/60 of a second (16.6 milliseconds) for 60-cycle power and 1/50 of a second (20 milliseconds) for 50-cycle power. The MSTIME UUO should normally be used so that the time is not a function of the cycle.

3.6.1.3 MSTIME AC, or CALLI AC, 23 - This UUO returns the time of day, in milliseconds, right justified in AC.

3.6.2 Job Status Information

3.6.2.1 RUNTIM AC, or CALLI AC, 27 - The accumulated running time (in milliseconds) of the job number specified in AC is returned right justified in AC. If the job number in AC is zero, the running time of the currently running job is returned. If the job number in AC does not exist, zero is returned.

3.6.2.2 PJOB AC, or CALLI AC, 30 - This UUO returns the job number right justified in AC.

3.6.2.3 GETPPN AC, or CALLI AC, 24 - This UUO returns in AC the project-programmer pair of the job. The project number is a binary number in the left half of AC, and the programmer number is a binary number in the right half of AC. If the program has the JACCT bit set, a skip return is given if the old project-programmer number is also logged in on another job.

3.6.2.4 OTHUSR AC, or CALLI AC, 77 - This UUO is used to determine if another job is logged in with the same project-programmer number as the job executing the UUO. The non-SKIP return is given if

1) the UUO is not implemented, in which case the AC remains unchanged, or
2) the UUO is implemented and no other jobs are logged in with the same project-programmer number, in which case the AC contains the project-programmer number of the job executing the UUO.

The SKIP return is given if the UUO is implemented and other jobs are logged in with the same project-programmer number. The AC contains the project-programmer number of the job executing the UUO. This UUO is used by KJOB.

3.6.3 Monitor Examination

3.6.3.1 PEEK AC, or CALLI AC, 33 - This UUO allows a user program to examine any location in the monitor. It is used by SYSTAT, FILDDT, and DATDMP and could be used for on-line monitor debugging. The PEEK UUO requires bit 16 (JP.SPA - examine all of core) and/or bit 17 (JP.SPM - examine the monitor) to be set in the privilege word . GTPRV.
The call is:

```
        MOVEI AC, exec address          ;TAKEN MODULO SIZE OF MONITOR
        PEEK AC,                        ;OR CALLI AC, 33
```

This call returns with the contents of the monitor location in AC.

3.6.3.2 SPY AC, or CALLI AC, 42 - This UUO is used for efficient examination of the monitor during timesharing. Any number of K of physical core (not limited to the size of the monitor) is

placed into the user's logical high segment. This amount cannot be saved with the monitor SAVE command (only the low segment is saved), cannot be increased or decreased by the CORE UUO (error return taken), or cannot have the user-mode write-protect bit cleared (error return taken). The call is:

```
        MOVEI AC, highest physical core location desired
        SPY AC,                          ;or CALLI AC, 42
        error return
        normal return
```

Any program that is written to use the SPY UUO should try the PEEK UUO if it receives an error return. The SPY UUO requires bit 16 (JP.SPA - examine all of core) and/or bit 17 (JP.SPM - examine the monitor) to be set in the privilege word .GTPRV.

3.6.3.3 POKE. AC, or CALLI AC, 114[1] - This UUO is used by a privileged user to alter one location in the monitor at a time. The POKE. UUO requires bit 4 (JP.POK) to be set in the privilege word .GTPRV.

The call is:

```
        MOVE AC, [3,,ADR]
        POKE. AC,                        ;or CALLI AC, 114
        error return
        normal return

ADR:    monitor location
        old value
        new value
```

The error return is given if:

The user is not privileged; AC contains 0.

The value specified in ADR+1 as the old value is not the same as the actual value contained in the monitor location; AC contains 1.

The address specified is not a valid monitor address; AC contains 2.

3.6.3.4 GETTAB AC, or CALLI AC, 41 - This UUO provides a mechanism which will not vary from monitor to monitor for user programs to examine the contents of certain monitor locations. The call is:

```
        MOVE AC, [XWD index, table number]
        GETTAB AC,                       ;or CALLI AC, 41
        error return
        normal return
```

[1]This UUO depends on FTPOKE which is normally off in the DECsystem-1040.

The left half of AC contains a job number or some other index to a table. Some job numbers may refer to high segments of programs by using arguments greater than the highest job number for the current monitor. A LH of -1 indicates the current job number. A LH of -2 references the job's high segment. An error return is given if there is no high segment or if the hardware and software are non-reentrant. The right half of AC contains a table number from the list of monitor data tables and parameters in Table 3-1. The entries in these tables are globals in the monitor subroutine COMMON. The actual values of the core addresses of these locations are subject to change and can be found in the LOADER storage map for the monitor. The complete description of these globals is found in the listing of COMMON.

The customer is allowed to add his own GETTAB tables to the monitor. A negative right half should be used to specify such customer-added tables.

An error return leaves the AC unchanged and is given if the job number or index number in the left half of AC is too high, the table number in the right half of AC is too high, or the user does not have the privilege of accessing the specified table.

A normal return supplies the contents of the requested table in AC,or a zero if the table is not defined in the current monitor.

The SYSTATprogram makes frequent use of this UUO.

NOTE

Many GETTAB tables have information in the undescribed bits. This information is likely to change and should be ignored. Although the field may currently be zero, there is no reason to believe that it will always be zero.

Table 3-1
GETTAB Tables

Table Numbers (RH of AC)	Table Names	Explanation
00	.GTSTS	Job status word; index by job or segment number.
01	.GTADR	Job relocation and protection; index by job or segment number
02	.GTPPN	Project and programmer numbers; index by job or segment number
03	.GTPRG	User program name; index by job or segment number.
04	.GTTIM	Total run time used in units of jiffies; index by job number. The value of a jiffy can be obtained from bit 6 of the STATES word (item 17 in the .GTCNF table).
05	.GTKCT	Kilo-Core ticks of job; index by job number.
06	.GTPRV	Privilege bits of job; index by job number, refer to Paragraph 3.6.3.4.1.
07	.GTSWP	Swapping parameters of job; index by job or segment number.
10	.GTTTY	Terminal-to-job translation; index by job number.
11	.GTCNF	Configuration table; index by item number, refer to Paragraph 3.6.3.4.2.
12	.GTNSW	Nonswapping data; index by item number, refer to Paragraph 3.6.3.4.3.
13	.GTSDT	Swapping data; index by item number, refer to Paragraph 3.6.3.4.4.
14	.GTSGN	High segment table; index by job number. Bit 0 = 0, then bits 18-35 are index of high segment (if bits 18-35 = 0, then there is no high segment). Bit 0 = 1, then bits 18-35 are number of K to spy on. Bit 1 (SN%SHR) = 1 if job has a high segment that is sharable. Bit 5 (SN%LOK) = 1 if job has a high segment that is locked.
15	.GTODP	Once-only disk parameters; index by item number, refer to Paragraph 3.6.3.4.5.
16	.GTLDV	5-series monitor disk parameters; index by item number, refer to Paragraph 3.6.3.4.6.

Table 3-1 (Cont)
GETTAB Tables

Table Numbers (RH of AC)	Table Names	Explanation
17	.GTRCT	Disk blocks read by job; used by DSK command: a. Bits 0-11 = incremental blocks b. Bits 12-35 = total blocks since start of job. Index by job number. Job 0 indicates the number of blocks swapped in.
20	.GTWCT	Disk blocks written by job: a. Bits 0-11 = incremental blocks. b. Bits 12-35 = total blocks since start of job. Index by job number. Job 0 indicates the number of blocks swapped out.
21	.GTDBS	Reserved for future.
22	.GTTDB	Reserved for future.
23	.GTSLF	Table of GETTAB addresses (GETTAB immediate); index by GETTAB table number, refer to Paragraph 3.6.3.4.7.
24	.GTDEV	Device or file structure name of sharable high segment. Index by high segment number.
25	.GTWSN	Two-character SIXBIT names for job queues; index by item numbers, refer to Paragraph 3.6.3.4.8.
26	.GTLOC	Job's logical station; index by job number.
27	.GTCOR	Physical core allocation. One bit per one K of core if system does not include LOCK UUO. Two bits per entry if system includes LOCK UUO. A non-zero entry indicates core in use.
30	.GTCOM	Table of SIXBIT names of monitor commands.
31	.GTNM1	First half of name of user in SIXBIT; index by job number.
32	.GTNM2	Last half of name of user in SIXBIT; index by job number.
33	.GTCNO	Job's charge number; index by job number.
34	.GTTMP	Job's TMPCOR pointers; index by job number.
35	.GTWCH	Job's WATCH bits; index by job number, refer to Paragraph 3.6.3.4.9.
36	.GTSPL	Job's spooling control bits; index by job number, refer to Paragraph 3.6.3.4.10.
37	.GTRTD	Job's real-time status word; index by job number.

Table 3-1 (Cont)
GETTAB Tables

Table Numbers (RH of AC)	Table Names	Explanation
40	.GTLIM	Job's time limit in jiffies and Batch status; index by job number. a. Bits 1-9 (JB.LCR) = job's core limit. b. Bit 10 = 1 (JB.LBT) if a Batch job. c. Bit 11 = 1 (JB.LSY) if program comes from SYS. Set on R command or equivalent. Cleared on R command (or equivalent) or SETNAM UUO. d. Bits 12-35 (JB.LTM) = job's time limit.
41	.GTQQQ	Timesharing scheduler's queue headers.
42	.GTQJB	Timesharing scheduler's queue that job is in; index by job number.
43	.GTCM2	Table of SET command names.
44	.GTCRS	Status of hardware taken on a crash. 0: CR.SAP = CONI APR, 1: CR.SPI = CONI PI, 2: CR.SSW = DATAI APR The remainder of the table contains the status of the various devices.
45	.GTISC	Swapper's input scan list of queues.
46	.GTOSC	Swapper's output scan list of queues.
47	.GTSSC	Scheduler's scan list of queues.
50	.GTRSP	Response counter table. Time in jiffies when user started to wait for his job to run. This time is cleared when the job is first given to the processor by the scheduler.
51	.GTSYS	System variables which are independent of CPU. Word 0 (%SYERR) = system wide hardware error count. Word 1 (%SYCCO) = number of times COMCNT was off. Word 2 (%SYDEL) = number of error-logging disabled errors. Word 3 (%SYSPC) = LH is a 3-letter code of the last STOPCD. RH is the address +1 of the last STOPCD executed. Word 4 (%SYNDS) = number of debug STOPCDs. Word 5 (%SYNJS) = number of job STOPCDs. Word 6 (%SYNCP) = total number of commands processed by the system since it was started.
52	.GTWHY	Operator why comments in ASCIZ.
53	.GTTRQ	Total time job was in run queues whether it was running or not.

Table 3-1 (Cont)
GETTAB Tables

Table Numbers (RH of AC)	Table Names	Explanation
54	.GTSPS	Job status word of second processor. Bit 29 (SP.SC0) = SET CPU command can be used. Bit 35 (SP.CR0) = SET CPU UUO can be used. Bits for other processors can be obtained by shifting left 1 bit per processor.
55	.GTC0C	CPU0 CDB constants; index by item number, refer to Paragraph 3.6.3.4.11.
56	.GTC0V	CPU0 CDB variables; index by item number, refer to Paragraph 3.6.3.4.12.
57	.GTC1C	CPU1 CDB constants; index by item number; see .GTC0C.
60	.GTC1V	CPU1 CDB variables; index by item number; see .GTC0V.
61	.GTC2C	CPU2 CDB constants; index by item number; see .GTC0C.
62	.GTC2V	CPU2 CDB variables; index by item number; see .GTC0V.
63	.GTC3C	CPU3 CDB constants; index by item number; see .GTC0C.
64	.GTC3V	CPU3 CDB variables; index by item number; see .GTC0V.
65	.GTC4C	CPU4 CDB constants; index by item number; see .GTC0C.
66	.GTC4V	CPU4 CDB variables; index by item number; see .GTC0V.
67	.GTC5C	CPU5 CDB constants; index by item number, see .GTC0C.
70	.GTC5V	CPU5 CDB variables; index by item number; see .GTC0V.
71	.GTFET	Current setting of all features defined in F.MAC, index by item number, refer to Paragraph 3.6.3.4.14.
72	.GTEDN	Table of ersatz device names (e.g., NEW, LIB) with their corresponding project-programmer numbers. The search lists of these devices can be obtained from the PATH. UUO.

3.6.3.4.1 Entries in Table 6- .GTPRV (Privilege Table)

Each job has a one-word entry to indicate job privileges. The privilege bits are as follows:

Bit	Mnemonic	Meaning
1-2	JP.DPR	Highest disk priority for this job.
3	JP.MET	Job is allowed to execute the METER.UUO.
4	JP.POK	Job is allowed to execute the POKE.UUO.
5	JP.CCC	Job is allowed to change its CPU specification via a command or UUO.
6-9	JP.HPQ	Highest high-priority queue available to this job.
10	JP.NSP	Job is allowed to unspool devices.
13	JP.RTT	Job is allowed to execute the RTTRP UUO.
14	JP.LCK	Job is allowed to execute the LOCK UUO.
15	JP.TRP	Job is allowed to execute the TRPSET UUO.
16	JP.SPA	Job is allowed to PEEK and SPY on all of core.
17	JP.SPM	Job is allowed to PEEK and SPY on the monitor.

3.6.3.4.2 Entries in Table 11 - .GTCNF (Configuration Table)

Item	Location	Use
0	%CNFG0	Name of system in ASCIZ.
-		
4	%CNFG4	
5	%CNDT0	Date of system in ASCIZ.
6	%CNDT1	
7	%CNTAP	Name of system device (SIXBIT).
10	%CNTIM	Time of day in jiffies.
11	%CNDAT	Today's date (15-bit format).
12	%CNSIZ	Highest location in monitor +1.
13	%CNOPR	Name of OPR TTY (SIXBIT).
14	%CNDEV	LH is start of DDB (device-data-block) chain.
15	%CNSJN	LH=-# of high segments, RH=+# of jobs (counting NULL job).
16	%CNTWR	Non-zero if system has two-register hardware and software.

Item	Location	Use
17	%CNSTS	Location describing feature switches of this system in LH, and current state in RH. Assembled according to MONGEN dialog and S.MAC: Bit 0=1 if disk system (ST%DSK) Bit 1=1 if swap system (ST%SWP) Bit 2=1 if LOGIN system (ST%LOG) Bit 3=1 if full duplex software (ST%FTT) Bit 4=1 if privilege feature (ST%PRV) Bit 5=1 if assembled for choice of reentrant or non-reentrant software at monitor load time (ST%TWR) Bit 6=1 if clock is 50 cycle instead of 60 cycle (ST%CYC) Bit 7-9 type of disk system (ST%TDS): if 0, 4-series disk system. if 1, 5-series disk system. if 2, spooled disk. Bit 10=1 if independent programmer numbers between project (INDPPN is non-zero) (ST%IND) Bit 11=1 if image mode on terminal (8-bit SCNSER) (ST%IMG) Bit 12=1 if dual processor system (ST%DUL) Bit 13=1 if multiple RIBs supported (ST%MRB) Bit 14=1 if high precision time accounting (ST%HPT) Bit 15 = 1 if overhead excluded from time accounting (ST%EMO) Bit 16=1 if real-time clock (ST%RTC) Bit 17=1 if built to handle FOROTS (ST%MBF) Set by the privileged operator command, SET SCHED: Bit 27=1 means no operator (ST%NOP) Bit 28=1 means unspooling devices (ST%NSP) Bit 29=1 means assigning devices (ST%ASS) Bit 33=1 means only Batch jobs may LOGIN (except from CTY or OPR) (ST%BON) Bit 34=1 means no remote LOGINs (ST%NRL) Bit 35=1 means no more LOGINs except from CTY or OPR (ST%NLG)
20	%CNSER	Serial number of PDP-10 processor. Set by MONGEN dialog.
21	%CNNSM	Number of nanoseconds per memory cycle for memory system. If the GETTAB fails, the number of nanoseconds per memory cycle is ↑D1000. Used by SYSTAT to compute shuffling time.

Item	Location	Use
22	%CNPTY	PTY parameters for Batch. LH = the number of the first invisible terminal (which is one greater than the number of the CTY) RH = the number of PTY's in the system configuration.
23	%CNFRE	AOBJN word to use bit map in monitor for allocating 4-word core blocks.
24	%CNLOC	LH=0, RH=address in monitor for free 4-word core block areas. (This is never changed while monitor runs).
25	%CNSTB	Link to STB chain for remote Batch.
26	%CNOPL	Address of the line data block (LDB) of the operator's terminal.
27	%CNTTF	Pointer to TTY free chunks.
30	%CNTTC	LH=number of TTY chunks. RH=address of first TTY chunk.
31	%CNTTN	Number of free TTY chunks.
32	%CNLNS	Pointer to current TTY as seen by the command decoder.
33	%CNLNP	Pointer to examine TTY line table, including remote terminals. LH= -total number of TTY lines. RH=beginning of line table.
34	%CNVER	Version of monitor. (Stored in location 137 of monitor as a save file when monitor is not running.) Bits 0-17 reserved for customer. Bits 18-23 monitor level (e.g., 5) Bits 24-29 monitor release (e.g., 3). Bits 30-35 used for internal development. If the GETTAB fails, the monitor is a version previous to 5.03
35	%CNDSC	Pointer to data set control table. LH = -length of table. RH = beginning of control table.
36	%CNDLS	Last received interrupt from the DC10.
37	%CNCCI	Last received interrupt from the 680I.
40	%CNSGT	Last dormant segment which was deleted to free a segment number.
41	%CNPOK	Address of last location changed in monitor by the POKE.UUO.

Item	Location	Use
42	%CNPUC	LH = the number of the job which successfully executed the POKE.UUO last. RH = the number of successful POKE.UUOs executed.
43	%CNWHY	The reason for the last reload (SIXBIT unabbreviated operator answer). Refer to ONCE in the DECsystem-10 Software Notebooks.
44	%CNTIC	The number of clock ticks per second. This is the time-of-day clock. The number is obtained by conducting a simple experiment at monitor load time. A different clock can be used for incremental run time accounting (refer to %CNRTC below).
45	%CNPDB	The pointer to the process data block (PDB) pointer tables.
46	%CNRTC	The run time clock rate (jiffies per second). That is, the rate of the clock used to measure the run time of the job and the system statistics (null, lost, and overhead time). This is the precision of the measurement, not the units of measurement.
47	%CNCHN	The pointer to the list of channel (DF10) data blocks. LH = the address of the 1st channel data block. RH = unused.
50	%CNLMX	LOGMAX. The maximum number of jobs allowed to LOGIN.
51	%CNBMX	BATMAX. The maximum number of Batch jobs allowed to LOGIN.
52	%CNBMN	BATMIN. The guaranteed number of Batch jobs (i.e., the number of jobs reserved for Batch).
53	%CNDTM	The universal date-time standard (refer to Paragraph 3.6.1).
54	%CNLNM	LOGNUM. The number of jobs currently logged-in.
55	%CNBNM	BATNUM. The number of Batch jobs currently logged-in.
56	%CNYER	LOCYER. The year.
57	%CNMON	LOCMON. The month (Jan = 1, Feb = 2, etc.).
60	%CNDAY	LOCDAY. The local day of the month (1, 2, 3,...).
61	%CNHOR	LOCHOR. The local hour in 24-hr format.
62	%CNMIN	LOCMIN. Minutes(0, 1, . . . , 59).
63	%CNSEC	LOCSEC. Seconds (0, 1, . . ., 59).

Item	Location	Use
64	%CNGMT	Offset for the universal date-time standard in order to convert it to local time from Greenwich Mean Time (not yet implemented).
65	%CNDBG	Debugging status word. Bit 0=1 System debugging (ST%DBG). Bit 1=1 Reload on debug stop code (ST%RDC). Bit 2=1 Reload on job stop code (ST%RJE).
66	%CNFRU	Amount of free core currently in use by the monitor.

3.6.3.4.3 Entries in Table 12 - .GTNSW (Nonswapping Data)

With the 5.05 and later monitors, no new entries will be added to the .GTNSW table because many of the parameters in this table are dependent upon the processor used and therefore are different for each processor in a multiprocessor system. GETTAB tables 51-70 exist for new parameters as well as the .GTNSW parameters.

Item	Location	Use
0 - 7		Obsolete, unspecified data.
10	%NSCMX	CORMAX. Size in words of largest legal user job +1 (Low seg+high seg).
11	%NSCLS	Byte pointer to last free block.
12	%NSCTL	Total free+dormant+idle K physical core left (virtual core).
13	%NSSHW	Job number shuffler has stopped.
14	%NSHLF	Absolute address of job above lowest hole, 0 if no job.
15	%NSUPT	Time system has been up in jiffies.
16	%NSSHF	Total number of words shuffled by system.
17	%NSSTU	Number of job using SYS if not a disk.
20	%NSHJB	Highest job number currently assigned.
21	%NSCLW	Total number of words cleared by system.
22	%NSLST	Total number of clock ticks when null job ran and other jobs wanted to but could not because: a. Swapped out or on way in or out. b. Monitor waiting for I/O to stop so it can shuffle or swap. c. Job being swapped out because of expanding core.

Item	Location	Use
23	%NSMMS	Size of physical memory in words.
24	%NSTPE	Total number of user parity errors (memory) since system was loaded.
25	%NSSPE	Total number of spurious (refer to Paragraph 7.7) parity errors (memory).
26	%NSMPC	Total number of multiple parity errors (memory).
27	%NSMPA	The absolute location of the last user mode memory parity error.
30	%NSMPW	The contents of the last user mode memory parity error.
31	%NSMPP	The user PC of the last user mode memory parity error.
32	%NSEPO	Total number of PDL OVR's at UUO level in exec mode which were not recovered.
33	%NSEPR	Number of PDL OVR's at UUO level which were recovered by assigning extended list.
34	%NSNXM	Highest legal value of CORMAX.
35	%NSKTM	Count-down timer for SET KSYS UUO.
36	%NSCMN	Amount of core guaranteed to be available after locking jobs in core (CORMIN).
37	%NSABC	Count of number of address breaks handled.
40	%NSABA	Contents of data switches on last address break.
41	%NSLJR	Last job that ran if different from the current job.
42	%NSACR	Accumulated CPU response. Total number of jiffies that all users waited for their jobs to initially run after either a command was issued which ran a job (program) or terminal input was given that removed the job from a TTY input wait state.
43	%NSNCR	Number of CPU responses for all users waiting for jobs to run (refer to %NSACR above). Dividing the value of %NSACR by the value of %NSNCR gives the average response time since system startup.
44	%NSSCR	Accumulated squares of the CPU response times obtained from %NSACR.

3.6.3.4.4 Entries in Table 13 - .GTSDT (Swapping Data)

Item	Location	Use
0	%SWBGH	Number of K in biggest hole in core.
1	%SWFIN	-Job number of job being swapped out, +Job number of job being swapped in.
2	%SWFRC	Job being forced to swap out.
3	%SWFIT	Job waiting to be fit into core.
4	%SWVRT	Amount of virtual core left in system in K (initially set to number of K of swapping space).
5	%SWERC	LH=number of swap read or write errors, RH=error bits (bits 18-21 same as status bits) + number of K discarded.
6	%SWPIN	-1 if job swapped in (monitors which swap process data blocks (PDBs) only).

3.6.3.4.5 Entries in Table 15 - .GTODP (Once-Only Disk Parameters)

Item	Location	Use
0	%ODSWP	Unused, contains zero in 5-series monitors.
1	%ODK4S	K of disk words set aside for swapping on all units in active swapping list.
2	%ODPRT	In-core protect time multiplies size of job in K-1.
3	%ODPRA	In-core protect time added to above result after multiply.

3.6.3.4.6 Entries in Table 16 - .GTLVD (5-series Monitor Disk Parameters)

Item	Location	Use
0	%LDMFD	Project-programmer number for UFDs only [1,1].
1	%LDSYS	Project-programmer number for device SYS [1,4]. In 4-series monitors [1,1].
2	%LDFFA	Project-programmer number for FAILSAFE [1,2].
3	%LDHLP	Project-programmer number for SYSTAT and HELP [2,5].
4	%LDQUE	Project-programmer number for spooling programs [3,3].
5	%LDSPB	a. LH=address of first PPB block. b. RH=address of next PPB block to be scanned.

Item	Location	Use
6	%LDSTR	a. LH=address of first file structure data block. b. RH=relative address of next file structure data block, i.e., the address within the data block which points to the actual address of the next data block.
7	%LDUNI	a. LH=address of data block of first unit in system. b. RH=relative address of data block of next unit in system.
10	%LDSWP	a. LH=address of first unit for swapping in system. b. RH=relative address of next unit for swapping in system.
11	%LDCRN	Number of 4-word access blocks for disk systems allocated at ONCE - only time.
12	%LDSTP	Standard file protection code (057), can be changed by installation. In 4-series monitors (055).
13	%LDUFP	Standard UFD protection code (775), can be changed by installation. In 4-series monitors (055).
14	%LDMBN	Number of monitor buffers allocated at once-only time (2). In 4-series monitors, 1.
15	%LDQUS	SIXBIT name of file structure containing 3,3.UFD for spooling and OMOUNT queues. In 4-series monitors, DSK.
16	%LDCRP	UFD used for storing system crashes. In 4-series monitors, [10,1].
17	%LDSFD	Maximum number of nested SFD's which the monitor allows to be created.
20	%LDSPP	Protection of spooled output files (bits 0-7).
21	%LDSYP	Standard protection for files in SYS: (155) except for files with an extension of .SYS.
22	%LDSSP	Standard protection for files in SYS: with an extension of .SYS (157).
23	%LDMNU	Maximum negative argument to USETI which reads extended RIBs.
24	%LDMXT	Maximum number of blocks transferred with one I/O operation (one IOWD). Normally 100000 but can be defined at MONGEN to be smaller so that a job doing high priority disk I/O will be locked out for a shorter period of time (since it can be locked out for as long as the channel is busy).
25	%LDNEW	Project-programmer number for experimental SYS [1,5].

Item	Location	Use
26	%LDOLD	Project-programmer number for library of superseded system programs [1,3].
27	%LDUMD	Project-programmer number for user mode diagnostics [6,6].
30	%LDNDB	Default number of disk buffers in a buffer ring.

3.6.3.4.7 Entries in Table 23 - .GTSLF (GETTAB Immediate)

This table is useful for a program that uses the SPY UUO for efficiency and needs the core address of the monitor tables. Absolute location 151 in the monitor contains the address of the beginning of this table.

The format of each entry is as follows:

LH=Bits 0-8 = maximum item number in table.
Bit 9 = data may be process data.
Bit 10 = data may be segment data.
Bits 14-17 = a monitor AC.

RH=executive-mode address of table (item 0).

Examples:

```
XWD ITEM + JBTMXL, JOBSTS
XWD ITEM + TTPMXL, TTYTAB
```

3.6.3.4.8 Entries in Table 25 - .GTWSN (Two-character SIXBIT names for job queues)

Word 0

Bits 0-11 = contain the two SIXBIT character mnemonic of job state code 0.
Bits 12-23 = contain the two SIXBIT character mnemonic of job state code 1.
Bits 24-35 = contain the two SIXBIT character mnemonic of job state code 2.

Word 1

Bits 0-11 = contain the mnemonics of job state code 3.
Bits 12-23 = contain the mnemonics of job state code 4.
Bits 24-35 = contain the mnemonics of job state code 5.
etc.

The job state codes for a disk system are as follows:

RN - one of the run queues.
WS - I/O wait satisfied.
TS - TTY I/O wait satisfied.
DS - disk I/O wait satisfied.
AU - disk alter UFD wait.

MQ	-	disk monitor buffer wait.
DA	-	disk storage allocation wait.
CB	-	disk core block scan wait.
D1	-	DECtape control wait.
D2	-	second DECtape control wait.
DC	-	data control wait.
MT	-	magnetic tape control wait.
CA	-	core allocation wait (to be locked).
IO	-	I/O wait.
TI	-	TTY I/O wait.
DI	-	disk I/O wait.
SL	-	sleep wait.
NU	-	null state.
ST	-	stop (↑C) state.
JD	-	DAEMON wait.

These state codes are printed by SYSTAT. Note that SYSTAT displays other codes based on analysis, such as the following:

TO	-	TTY output.
↑C	-	job stopped.
↑W	-	command wait.
OW	-	operator wait.
HB	-	hibernate.

3.6.3.4.9 Entries in Table 35 - .GTWCH (WATCH Table)

Each job has a one-word entry to indicate the WATCH bits. The bits for each word are as follows:

Bit	Mnemonic	Meaning
1	JW.WDY	Watch time of day.
2	JW.WRN	Watch run time.
3	JW.WWT	Watch wait time.
4	JW.WDR	Watch disk reads.
5	JW.WDW	Watch disk writes.
6	JW.WVR	Watch versions.

3.6.3.4.10 Entries in Table 36 - .GTSPL (Spooling Table)

Each job has a one-word entry to indicate the spooling control bits. These bits are as follows:

Bit	Mnemonic	Meaning
35	JS.PLP	Line printer spooling.
34	JS.PPL	Plotter spooling.
33	JS.PPT	Paper tape punch spooling.
32	JS.PCP	Card punch spooling.
31	JS.PCR	Card reader spooling.
24-26	JS.PRI	Disk priority.

3.6.3.4.11 Entries in Table 55 - .GTC0C (CPU0 CDB constants table)

The items in this table correspond to the items in the constants table for each processor.

CPU1	Table 57 - .GTC1C
CPU2	Table 61 - .GTC2C
CPU3	Table 63 - .GTC3C
CPU4	Table 65 - .GTC4C
CPU5	Table 67 - .GTC5C

Item	Location	Use
0	%CCPTR	LH=pointer to next CDB, or 0 if this is the last CDB. RH=unused.
1	%CCSER	APR serial number.
2	%CCOKP	If less than or equal to zero, CPU is running ok. If greater than zero, CPU has stopped running correctly. Contents of word is the number of jiffies CPU has been stopped.
3	%CCTOS	Trap offset for KA10 interrupt locations (0 or 100).
4	%CCLOG	Logical CPU name in SIXBIT (CPUn).
5	%CCPHY	Physical CPU name in SIXBIT (CPAn, CPIn, or CP6n).
6	%CCTYP	Type of processor (LH for customers, RH for DEC) 1 (.CC166) = PDP-6 2 (.CCKAX) = KA10 3 (.CCKIX) = KI10
7	%CCMPT	Relative GETTAB pointer to memory parity bad address subtable. Refer to Paragraph 3.6.3.4.13. Bits 0-8: maximum relative entry in subtable Bits 18-35: relative address of first word in subtable in CPU variable GETTAB (.GTC0V). If word is 0, the subtable has been conditionally assembled out of the monitor.
10	%CCRTC	Real time clock (DK10) DDB. If word is 0, there is no real time clock on this CPU.
11	%CCRTD	Real time clock DDB if high precision time accounting. If 0, there is no high precision time accounting on this CPU.

Item	Location	Use
12	%CCPAR	Relative GETTAB pointer to memory parity subtable. Refer to Paragraph 3.6.3.4.13. Bits 0-8 — maximum relative entry in subtable. Bits 18-35 — relative address of first word in subtable in CPU variable GETTAB. (.GTC0V). If word is 0, the subtable has been conditionally assembled out of the monitor.
13	%CCRSP	Relative GETTAB pointer to response subtable. Refer to Paragraph 3.6.3.4.13. Bits 0-8 — Maximum relative entry in subtable. Bits 18-35 — relative address of first word in subtable in CPU variable GETTAB (.GTC0V).

3.6.3.4.12 Entries in Table 56 - .GTC0V (CPU0 CDB Variable Table)

The items in this table correspond to the items in the variables table for each processor.

CPU1 Table 60 - .GTC1V
CPU2 Table 62 - .GTC2V
CPU3 Table 64 - .GTC3V
CPU4 Table 66 - .GTC4V
CPU5 Table 70 - .GTC5V

Item	Location	Use
5	%CVUPT	Uptime in jiffies for this CPU.
12	%CVLST	Lost time in jiffies for this CPU.
14	%CVTPE	Total memory parity error words detected during all CPU sweeps on this CPU while processor was in exec or user mode. If the system halts, this location has already been updated.
15	%CVSPE	Total spurious memory parity errors detected on this CPU (i.e., errors which did not reoccur when the CPU swept through core). Can occur on a read-pause-write which rewrites memory or on a channel-detected parity not found on the sweep (refer to %CVPCS in parity subtable.)
16	%CVMPC	Multiple memory parity errors for this CPU. That is, the number of time the operator pushed CONTINUE after a serious memory parity halt. LH = 1 if serious error on this bad parity (must halt). LH is cleared on CONTINUE or STARTUP.

Item	Location	Use
17	%CVMPA	Memory parity address for this CPU. That is, first bad physical memory address found when the monitor swept through core after processor or channel detected first parity error.
20	%CVMPW	Memory parity word for this CPU. That is, contents of first bad word found by monitor when it swept through core after the processor or channel detected first bad parity.
21	%CVMPP	Memory parity PC for this CPU. That is, PC of last memory parity (not counting sweep through core).
27	%CVABC	Address break count on this CPU.
30	%CVABA	Address break address on this CPU.
31	%CVLJR	Last job run on this CPU including the null job.
32-34		Obsolete. Refer to items 20-23 in the Response Subtable.
35	%CVSTS	Stop timesharing on this CPU. Contains job number which performed the TRPSET UUO.
36	%CVRUN	Operator-controlled scheduling for this CPU (OPSER: SET RUN command). Bit 0 (CV%RUN) = 1 do not run jobs on this CPU.
37	%CVNUL	Null time in jiffies for this CPU.
40	%CVEDI	LH = exec PC so that offending instruction can be corrected. RH = number of exec "don't care" interrupts (i.e., user enabled APR interrupts which monitor causes (AOV, FOV).
41	%CVJOB	Current job running on this CPU (0 is null job).
42	%CVOHT	Overhead time in jiffies for this CPU. Includes clock queue processing, short command processing, swapping and scheduling decisions, and software context switching. Does not include UUO execution or I/O interrupt time, since these times are not overhead.
43	%CVEVM	(KI10 only) Maximum amount of exec virtual address space to be used for mapping user segments on a LOCK UUO.
44	%CVEVU	(KI10 only) Current amount of exec virtual address space being used for mapping user segments on a LOCK UUO.
45	%CVLLC	On a dual processor system, the count of the number of times a CPU has looped in the CPU interlock while waiting for it to be relinquished by the second CPU.

Item	Location	Use
46	%CVTUC	Total number of UUOs executed on this CPU from exec and user mode.
47	%CVTJC	Total number of job context switches from one job to a different job, including the null job, on this CPU.

3.6.3.4.13 GETTAB Subtables - Via the GETTAB mechanism, GETTAB subtables make monitor-collected data available to user programs and, at the same time, allow the installation to decide if it wants to use more monitor table space without invalidating any user programs. These subtables are included in all systems except the DECsystem-1040. However, they may be excluded by changing the appropriate conditional assembly switches with MONGEN. It is anticipated that only installations that need the core space for other uses will decide to exclude the subtables.

To reference a subtable, the user program first does a GETTAB UUO to obtain the pointer to the subtable (refer to Paragraph 3.6.3.4.11). Then the program does a second GETTAB to get the desired item in the subtable. If the pointer is zero, the desired subtable is not included in the system.

The following example illustrates the method for obtaining the accumulated response times for CPU N for all users that waited for their jobs to initially run after TTY input was given.

```
%CCRSP==XWD 13,55       ;WORD AND TABLE NUMBER FOR RESPONSE SUBTABLE
%CVRAI==3               ;SUBTABLE INDEX FOR ACCUMULATED TTY INPUT UUO
                        ;   RESPONSE.
.GTC0V==56              ;GETTAB TABLE FOR CPU0 VARIABLES
    MOVEI T1,N          ;CPU NUMBER (0,1,...,5)
    LSH T1,N            ;CONSTANTS TABLE GETTAB INDEX MOVES UP BY TWOS.
    MOVE T2,[%CCRSP]    ;RELATIVE GETTAB POINTER WORD FOR RESPONSE
                        ;   SUBTABLE FOR CPU0.
    ADD T2,T1           ;FORM GETTAB ARGUMENT FOR CPU N.
    GETTAB T2,          ;GET RELATIVE POINTER TO RESPONSE SUBTABLE.
      JRST NONE         ;NOT THERE (MONITOR IS ONE BEFORE 5.05)
    JUMPE T2,NONE       ;IF 0, SUBTABLE NOT INCLUDED IN THIS
                        ;   LOAD OF THE MONITOR.
    ADDI T2,%CVRAI      ;FORM DESIRED INDEX IN SUBTABLE WITH
                        ;   RESPECT TO VARIABLE GETTAB.
    HRL T2,T2           ;RELATIVE ADDRESS OF SUBTABLE WITH
                        ;   RESPECT TO VARIABLE TABLE.
    HRRI T2,.GTC0V(T1)  ;FORM PROPER GETTAB FOR CPU VARIABLES.
    GETTAB T2,          ;GET RESPONSE TIME.
      JRST NONE         ;NOT THERE. THIS SHOULD NOT HAPPEN SINCE
                        ;   ZERO TEST ON RELATIVE POINTER FAILED.
    HERE WITH RESPONSE IN T2
```

Response Subtable

The response subtable is pointed to by %CCRSP in the constants table for each processor. This subtable is under the conditional assembly switch FTRSP. Refer to Paragraph 3.6.3.4.3 for additional response information.

Item	Location	Use
0	%CVRSO	Accumulated TTY output UUO responses. That is, the total number of jiffies users have spent waiting for their jobs to do a TTY output UUO (on CPU0) after either a command was issued which ran a job or terminal input was given that removed the job from a TTY input wait state.
1	%CVRNO	Number of TTY output UUO responses for this CPU.
2	%CVRHO	The high-order sum of the squares of TTY output UUO responses. Used for computing standard deviation.
3	%CVRLO	The low-order sum of the squares of TTY output UUO responses.
4	%CVRSI	Accumulated TTY input UUO responses for this CPU. That is, the total number of jiffies users have spent waiting for their jobs to do a TTY input UUO (on CPU0) after either a command was issued which ran a job or terminal input was given that removed the job from a TTY input wait state.
5	%CVRNI	Number of TTY input UUO responses for this CPU.
6	%CVRHI	The high-order sum of the squares of TTY input UUO responses. Used for computing standard deviation.
7	%CVRLI	The low-order sum of the squares of TTY input UUO responses.
10	%CVRSR	Accumulated CPU quantum requeue responses. That is, total number of jiffies users spent waiting for their jobs to exceed the CPU quantum on this CPU after either a command was issued which ran a job or terminal input was given that removed the job from a TTY input wait state.
11	%CVRNR	Number of CPU quantum requeue responses for this CPU.
12	%CVRHR	The high-order sum of the squares of CPU quantum requeue response. Used for computing standard deviation.
13	%CVRLR	The low-order sum of the squares of CPU quantum requeue response.
14	%CVRSX	Accumulated response terminated by the first occurrence of one of the above 3 events (TTY output, TTY input, or CPU quantum requeue).
15	%CVRNX	Number of such responses in %CVRSX.
16	%CVRHX	The high-order sum of the squares of responses in %CVRSX. Used for computing standard deviation.

Item	Location	Use
17	%CVRLX	The low-order sum of the squares of responses in %CVRSX.
20	%CVRSC	Accumulated CPU responses on this CPU. Total number of jiffies that users waited for their jobs to run after either a command was issued which ran a job or terminal input was given that removed the job from a TTY input state.
21	%CVRNC	Number of CPU responses for all users waiting for their jobs to run. Dividing this value into the value of %CVRSC gives the average response time since the system was started.
22	%CVRHC	The high-order sum of the squares of CPU responses on this CPU.
23	% CVRLC	The low-order sum of the squares of CPU responses on this CPU.

Parity Subtable

The parity table is pointed to by %CCPAR in the constants table for each processor. This subtable is under the conditional assembly switch FTMEMPAR. Refer to Paragraphs 3.6.3.4.3 and 7.7 for additional parity information.

Item	Location	Use
0	%CVPLA	Highest bad memory parity address on last sweep of memory. Used to tell operator the range of bad addresses.
1	%CVPMR	Relative address (not virtual address) in the high or low segment of the last memory parity error.
2	%CVPTS	Number of parity errors on the last sweep of core. Set to 0 at beginning of the sweep.
3	%CVPSC	Number of parity sweeps by the monitor.
4	%CVPUE	Number of user-enabled parity errors. Refer to Paragraph 3.1.3.1.
5	%CVPAA	The AND of bad addresses on the last memory parity sweep.
6	%CVPAC	The AND of bad contents on the last memory parity sweep.
7	%CVPOA	The OR of bad addresses on the last memory parity sweep.
10	%CVPOC	The OR of bad contents on the last memory parity sweep.
11	%CVPCS	Number of spurious parity errors. (The APR sweep found no bad parity but the channel had requested the sweep rather than the processor). This indicates a channel memory port problem.

Bad Address Subtable

The bad address table is pointed to by %CCMPT in the constants table for each processor. This subtable is under the conditional assembly switch FTMEMPAR and contains the bad addresses on the last memory parity sweep. It is not cleared and the number of valid entries is kept in %CVPTS in the parity subtable.

3.6.3.4.14 Entries in Table 71 - .GTFET (Feature Table)

This table provides the user with a mechanism for determining the current settings of all features defined in F.MAC

Item	Location	Use
0	%FTUUO	UUOs Bit 26 = 1 if control-C intercept (F%CCIN). Bit 27 = 1 if JOBSTS and CTLJOB UUOs are implemented (F%PTYU). Bit 28 = 1 if PEEK UUO implemented (F%PEEK). Bit 29 = 1 if POKE. UUO implemented (F%POKE). Bit 30 = 1 if JOB continue (F%JCON). Bit 31 = 1 if spooling supported (F%SPL). Bit 32 = 1 if job privileges supported (F%PRV). Bit 33 = 1 if DAEMON supported (F%DAEM). Bit 34 = 1 if GETTAB exists (F%GETT). Bit 35 = 1 if 2-register relocation (F%2REL).
1	%FTRTS	Real time and scheduling features Bit 27 = 1 if swapper (F%SWAP). Bit 28 = 1 if shuffler (F%SHFL). Bit 29 = 1 if DK10 service (F%RTC). Bit 30 = 1 if LOCK UUO implemented (F%LOCK). Bit 31 = 1 if TRPSET UUO implemented (F%TRPS). Bit 32 = 1 if real-time traps implemented (F%RTTR). Bit 33 = 1 if SLEEP UUO implemented (F%SLEE). Bit 34 = 1 if HIBER and WAKE UUOs supported (F%HIBW). Bit 35 = 1 if high priority queues supported (F%HPQ)
2	%FTCOM	Commands Bit 23 = 1 if COMPIL commands (F%CCL). Bit 24 = 1 if COMPIL-class (F%CCLX). Bit 25 = 1 if QUEUE (F%QCOM). Bit 26 = 1 if SET UUO and command (F%SET). Bit 27 = 1 if VERSION (F%VERS). Bit 28 = 1 if Batch control file commands (F%BCOM). Bit 29 = 1 if SET DAYTIME and SET DATE (F%SEDA). Bit 30 = 1 if WATCH (F%WATC).

Item	Location	Use
		Bit 31 = 1 if FINISH and CLOSE (F%FINI). Bit 32 = 1 if REASSIGN (F%REAS). Bit 33 = 1 if E and D (F%EXAM). Bit 34 = 1 if SEND (F%TALK). Bit 35 = 1 if ATTACH (F%ATTA).
3	%FTACC	Accounting information Bit 31 = 1 if time and core limits (F%TLIM). Bit 32 = 1 if charge number (F%CNO). Bit 33 = 1 if user name (F%UNAM). Bit 34 = 1 if kilo-core-ticks accumulation (F%KCT). Bit 35 = 1 if run-time computation (F%TIME).
4	%FTERR	Error control and internal options Bit 26 = 1 if swapping process data block (F%PDBS). Bit 27 = 1 if KI10 features at startup time (F%KI10). Bit 28 = 1 if METER. UUO supported (F%METR). Bit 29 = 1 if execute-only files (F%EXON). Bit 30 = 1 if illegal instruction message checks for for KI10 instructions (F%KII). Bit 31 = 1 if code to load BOOTS from disk (F%BOOT). Bit 32 = 1 if more than one swapping device (F%2SWP). Bit 33 = 1 if DAEMON error logging (F%EL). Bit 34 = 1 if multi-processor code loaded (F%MS). Bit 35 = 1 if memory parity error recovery (F%MEMP).
5	%FTDEB	Debugging features Bit 28 = 1 if response time measurement (F%RSP). Bit 29 = 1 if why reload code (F%WHY). Bit 30 = 1 if patch space left in tables (F%PATT). Bit 31 = 1 if back-tracking information left in COMMON (F%TRAC). Bit 32 = 1 if monitor halts on error (F%HALT). Bit 33 = 1 if redundancy checking for internal errors (F%RCHK). Bit 34 = 1 if monitor write protected (F%MONP). Bit 35 = 1 if monitor check summing (F%CHEC).
6	%FTSTR	File structure parameters Bit 21 = 1 if NUL device (F%NUL). Bit 22 = 1 if LIB/SYS/NEW (F%LIB). Bit 23 = 1 if disk priority transfers (F%DPRI). Bit 24 = 1 if append to last block (F%APLB). Bit 25 = 1 if append implies read (F%AIR). Bit 26 = 1 if generic device search (F%GSRC). Bit 27 = 1 if rename across directories (F%DRDR). Bit 28 = 1 if SEEK UUO (F%DSEK). Bit 29 = 1 if super USETI/USETO (F%DSUP). Bit 30 = 1 if disk quotas (F%DQTA).

Item	Location	Use
		Bit 31 = 1 if multiple file structures (F%STR). Bit 32 = 1 if 5-series UUOs (F%5UUO). Bit 33 = 1 if physical-only I/O (F%PHYO). Bit 34 = 1 if sub-file directories (F%SFD). Bit 35 = 1 if STRUUO functions (F%MOUN).
7	%FTDSK	Internal disk parameters Bit 21 = 1 if DEBUG CB interlock (F%CBDB) Bit 22 = 1 if LOGIN system (F%LOGI). Bit 23 = 1 if disk system (F%DISK). Bit 24 = 1 if race-condition prevention in FILFND (F%FFRE). Bit 25 = 1 if swap read error recovery (F%SWPE). Bit 26 = 1 if bad block marking (F%DBBK). Bit 27 = 1 if UFD compressor (F%DUFC). Bit 28 = 1 if disk error simulation (F%DETS). Bit 29 = 1 if extended RIBs supported (F%DMRB). Bit 30 = 1 if smaller allocation for disk core blocks (F%DSMC). Bit 31 = 1 if allocation optimization (F%DALC). Bit 32 = 1 if disk usage statistics (F%DSTT). Bit 33 = 1 if hung disk recovery (F%DHNG). Bit 34 = 1 if disk off-line recovery (F%DBAD). Bit 35 = 1 if latency optimization (F%DOPT).
10	%FTSCN	Scanner options Bit 27 = 1 if TTY BLANK command (F%TBLK). Bit 28 = 1 if page and display knowledge (F%TPAG). Bit 29 = 1 if automatic dialer supported (F%DTAL). Bit 30 = 1 if special line control (F%SCLC). Bit 31 = 1 if hardware (DC10 or DC68) scanner (F%SCNR). Bit 32 = 1 if modem control (F%MODM). Bit 33 = 1 if single scanner 630 (F%630H). Bit 34 = 1 if U.K. modem supported (F%GPO2). Bit 35 = 1 if real half-duplex terminals (F%HDPX).
11	%FTPER	I/O Parameters Bit 27 = 1 if CDP trouble intercept (F%CPTR). Bit 28 = 1 if CDR trouble intercept (F%CRTR). Bit 29 = 1 if CTY1 supported (F%CTY1). Bit 30 = 1 if remote station supported (F%REM). Bit 31 = 1 if LPT error recovery (F%LPTR). Bit 32 = 1 if device errors to operator (F%OPRE). Bit 33 = 1 if CDR super-image mode (F%CDRS). Bit 34 = 1 if MTA density and buffer size (F%MTSE). Bit 35 = 1 if TMPCOR area (F%TMP).

3.6.4 Configuration Information

3.6.4.1 SWITCH AC, or CALLI AC, 20 - This UUO returns the contents of the central processor data switches in AC. Caution must be exercised in using the data switches because they are not an allocated resource and are always available to all users.

3.6.4.2 LIGHTS AC, or CALLI AC, -1 - This UUO displays the contents of AC in the console lights.

3.7 DAEMON AC, OR CALLI AC, 102[1]

This UUO requests the DAEMON program to perform a specified function for the user program. The call is:

```
        MOVE AC, [XWD length (n+1), ADR]
        DAEMON AC,                           ;or CALLI AC, 102
        error return
        normal return
ADR:    function
          arg1
          arg2
            :
          argn
```

The length of the argument list can be zero if the number of arguments is fixed. The first word of the argument list is the code for the requested function. Non-privileged functions of the DAEMON UUO are presented in the following paragraphs. Refer to the Specifications section of the <u>DECsystem-10 Software Notebooks</u> for a description of the privileged functions.

3.7.1 .DCORE Function

This function causes DAEMON to write a dump file of the job's core area. The call is:

```
ADR:    1                   ;.DCORE function
        SIXBIT/dev/
        SIXBIT/file/
        SIXBIT/ext/
        0
        XWD ppn
        SIXBIT/SFD1/
            :
        SIXBIT/SFDN/
```

[1] This UUO depends on FTDAEM which is normally off in the DECsystem-1040.

If an argument is omitted, the default is the same as in the DCORE command (refer to DECsystem-10 Operating System Commands).

3.7.2 .CLOCK Function

This function causes DAEMON to enter a request in the clock queue in order to wake the job after the specified number of seconds has elapsed. The UUO returns as soon as the request is entered. The HIBER UUO with no clock request (refer to Paragraph 3.1.4.2) should then be used to place the job in the sleep queue.

The call is:

```
          MOVEI AC, BLOCK
          DAEMON AC,
            JRST ERROR
          SETZ AC,
          HIBER AC,
            JRST ERROR

ERROR: ...                    ;simulate the DAEMON UUO
                              ; with the SLEEP UUO.
BLOCK: 2                      ;.CLOCK function
       +seconds               ;number of seconds to sleep.
```

If the job already has a request in the clock queue, the new request supersedes the current request. Thus, jobs desiring to be awakened several times should issue one request for the soonest wake time.

There is no maximum on the amount of time a job can sleep and therefore, this UUO is useful when a sleep time of more than 63 seconds is desired (the SLEEP and HIBER UUOs have an implied maximum of 63 seconds). A request specifying 0 seconds clears the job's entry in the clock queue and immediately wakes the job. Note that the resolution of the timer may be several seconds slow if the system is heavily loaded.

3.7.3 Returns

The error return is given if the UUO is not implemented, DAEMON is not running, or DAEMON cannot complete the requested function. If the UUO is not implemented or DAEMON is not running, AC remains unchanged. If DAEMON cannot complete the request, AC contains one of the following error codes:

1	DMILF%	Illegal function.
2	DMACK%	Address check. The argument block is outside of user core or in the job data area.
3	DMWNA%	Wrong number of arguments.
4	DMSNH%	Impossible UUO failure (should never happen).

(Cont on next page)

5	DMCWF%	Cannot write file. An OPEN or INIT failed.
6	DMNPV%	No privileges. An attempt was made to write in the accounting files without having the proper privileges.
7	DMFFB%	FACT format is bad.
10	DMPTH%	Invalid path specification.

The normal return is taken if the requested function is successfully completed.

3.8 REAL-TIME PROGRAMMING

3.8.1 RTTRP AC, or CALLI AC, 57[1]

The real-time trapping UUO is used by timesharing users to dynamically connect real-time devices to the priority interrupt system, to respond to these devices at interrupt level, to remove the devices from the interrupt system, and to change the PI level on which the devices are associated. The RTTRP UUO can be called from UUO level or from interrupt level. This is a privileged UUO that requires the job to have real-time privileges (granted by LOGIN) and to be locked in core (accomplished by LOCK UUO). These real-time privileges are assigned by the system manager and obtained by the monitor from ACCT. SYS. The privilege bits required are:

1) JP.LCK (Bit 14) - allows the job to be locked in core.
2) JP.RTT (Bit 13) - allows the RTTRP UUO to be executed.

WARNING

> Improper use of features of the RTTRP UUO can cause the system to fail in a number of ways. Since design goals of this UUO were to give the user as much flexibility as possible, some system integrity had to be sacrificed. The most common errors are protected against since user programs run in user mode with all ACs saved. It is recommended that debugging of real-time programs not be done when system integrity is important. However, once these programs are debugged, they can run simultaneously with batch and timesharing programs.

Real-time jobs control devices one of two ways: block mode or single mode. In block mode, an entire block of data is read before the user's interrupt program is run. In single mode, the user's interrupt program is run every time the device interrupts. Furthermore, there are two types of block mode: fast block mode and normal block mode. These differ in response times. The response time to read a block of data in fast block mode is 6.5 μs per word and in normal block mode, 14.6 μs per word. (This is the CPU time to complete each data transfer.) In all modes, the response time measured from the receipt of the real-time device interrupt to the start of the user control program is 100 μs.

---

[1] This UUO depends on FTRTTRP which is normally off in the DECsystem-1040.

The RTTRP UUO allows a real-time job to either put a BLKI or BLKO instruction directly on a PI level (block mode) or add a device to the front of the monitor PI channel CONSO skip chain (single mode). Since the BLKI and BLKO are executed in exec mode, a KI10-based system requires that the job be mapped in exec virtual memory, in addition to being locked (refer to the LOCK UUO). When an interrupt occurs from the real-time device in single mode or at the end of a block of data in block mode, the monitor saves the current state of the machine, sets the new user virtual memory and APR flags, and traps to the user's interrupt routine. The user services his device and then returns control to the monitor to restore the previous state of the machine and to dismiss the interrupt.

In fast block mode the monitor places the BLKI/BLKO instruction directly in the PI trap location followed by a JSR to the context switcher. This action requires that the PI channel be dedicated to the real-time job during any transfers. In normal block mode the monitor places the BLKI/BLKO instruction directly after the real-time device's CONSO instruction in the CONSO skip chain (refer to Chapter 7).

Any number of real-time devices using either single mode or normal block mode can be placed on any available PI channel. The average extra overhead for each real-time device on the same channel is 5.5 μs per interrupt.

The call is:

```
        MOVEI AC, RTBLK         ;AC contains address of data block.
        RTTRP AC,               ;or CALLI AC, 57, put device on PI level.
        error return            ;AC contains an error code.
        normal return           ;PI is set up properly.
```

The data block depends on the mode used. In single mode the data block is:

```
RTBLK:  XWD PICHL, TRPADR       ;PI channel (1-6) and trap address.
        EXP APRTRP              ;APR trap address.
        CONSO DEV, BITS         ;CONSO chain instruction.
        0                       ;no BLKI/BLKO instruction.
```

The data block in fast block mode is:

```
RTBLK:  XWD PICHL, TRPADR       ;PI and trap address when BLKO done.
        EXP APRTRP              ;APR trap address.
        BLKO DEV, BLKADR        ;BLKI or BLKO instruction.
        0                       ;BLKADR points to the IOWD of
                                ;block to be sent.
```

The data block in normal block mode is:

```
RTBLK:  XWD PICHL, TRPADR       ;channel and trap address.
        EXP APRTRP              ;APR trap address.
        CONSO DEV, @BITMSK      ;control bit mask from user area.
        BLKI DEV, BLKADR        ;BLKI instruction.
```

On multiprocessor systems, the real-time trap UUO applies only to the processor specified by the job's CPU specification (refer to the SET CPU command or the SET UUO). If the specification indicates more than one processor, the specification is changed to indicate CPU0. Note that the PI channel (PICHL) and processor traps (APRTRP) are only for the indicated CPU.

3.8.1.1 Data Block Mnemonics - The following mnemonics are used in describing the data block associated with the RTTRP UUO.

PICHL - PICHL is the PI level on which the device is to be placed. Levels 1-6 are legal depending on the system configuration. If PICHL = 0, the device is removed from all levels. When a device is placed on a PI level, normally all other occurrences of the device on any PI level are removed. If the user desires the same device on more than one PI level simultaneously (i.e., a data level and an error level), he can issue the RTTRP UUO with PICHL negative. This indicates to the system that any other occurrence of this device (on any PI level) is not to be removed. Note that this addition to a PI level counts as a real-time device, occupying one of the possible real-time device slots.

TRPADR - TRPADR is the location trapped to by the real-time interrupt (JRST TRPADR). Before the trap occurs, all ACs are saved by the monitor and can be overwritten without concern for their contents.

APRTRP - APRTRP is the trap location for all APR traps. When an APR trap occurs, the monitor simulates a JSR APRTRP. The user gains control from an APR trap on the same PI level that his real-time device is on. The monitor always traps to the user program on illegal memory references, non-existent memory references, and push-down overflows. This allows the user to properly turn off his real-time device if needed. The monitor also traps on the conditions specified by the APRENB UUO (see Paragraph 3.1.3.1). No APR errors are detected if the interrupt routine is on a PI level higher than or equal to the APR interrupt level.

DEV - DEV is a real-time device code.

BITS - BITS is the bit mask of all interrupt bits of the real-time device and must not contain any other bits. If the user desires control of this bit mask from his user area, he may specify one level of indirection in the CONSO instruction (no indexing), i.e., CONSO DEV, @ MASK where MASK is the location in the user area of the bit mask. MASK must not have any bits set in the indirect or index fields.

BLKADR - BLKADR is the address in the user's area of the BLKI/BLKO pointer word. Before returning to the user, the monitor adds the proper relocation factor to the right half of the pointer word. Data can only be read into the low segment above the protected job data area, i.e., above location 114.

Since the pointer word is in the user's area, the user can set up a new pointer word when the word count goes to 0 at interrupt level. This allows fast switching between buffers. When the user desires to set up his own pointer word, the right half of the word must be set as an exec virtual instead of a user virtual address. The job's relocation value is returned from both the LOCK UUO and the first RTTRP UUO executed for setting the BLKI/BLKO instruction. If this pointer word does not contain a legal address, a portion of the system might be overwritten. A check should be made to determine if the negative word count in the left half of the pointer word is too large. If the word count extends beyond the user's own area, the device may cause a non-existent memory interrupt, or may overwrite a timesharing job. If all of the above precautions are taken, this method of setting up the pointer word is much faster and more flexible than issuing the RTTRP UUO at interrupt level.

3.8.1.2 Interrupt Level Use of RTTRP - The format of the RTTRP UUO at interrupt level is similar to the format at user level except for two restrictions:

1) AC 16 and AC 17 cannot be used in the UUO call (i.e., CALLI 16, 57 is illegal at interrupt level).
2) All ACs are overwritten when the UUO is executed at interrupt level. Therefore, the user must save any desired ACs before issuing the RTTRP UUO. This restriction is used to save time at interrupt level.

CAUTION

If an interrupt level routine executes a RTTRP UUO that affects the device currently being serviced, no additional UUOs of any kind (including RTTRP) can be executed during the remainder of the interrupt. At this point, any subsequent UUO dismisses the interrupt.

3.8.1.3 RTTRP Returns - On a normal return, the job is given user IOT privileges. These privileges allow the user to execute all restricted instructions including the necessary I/O instructions to control his device.

The IOT privilege must be used with caution because improper use of the I/O instructions could halt the system (i.e., HALT on the KA10; CONO APR, 0; DATAO APR, 0; CONO PI, 0 on the KA10 and KI10; and CONO PAG, 0 or DATAO PAG, 0 on the KI10). Note that a user can obtain just the user IOT privilege by issuing the RTTRP UUO with PICHL = 0.

An error return is not given to the user until RTTRP scans the entire data block to find as many errors as possible. On return, AC may contain the following error codes.

Name	Code	Value	Meaning
RTJNP%	Bit 24=1	4000	Job not privileged.
RTCPU%	Bit 25=1	2000	Not permitted on CPU1. (This is a temporary error condition reflecting the fact that the initial release of the 5.05 monitor will not support the RTTRP UUO on CPU1).
RTDIU%	Bit 26=1	1000	Device already in use by another job.
RTIAU%	Bit 27=1	400	Illegal AC used during RTTRP UUO at interrupt level.
RTJNL%	Bit 28=1	200	Job not locked in core.
RTSLE%	Bit 29=1	100	System limit for real-time devices exceeded.
RTILF%	Bit 30=1	40	Illegal format of CONSO, BLKO, or BLKI instruction.
RTPWI%	Bit 31=1	20	BLKADR or pointer word illegal.
RTEAB%	Bit 32=1	10	Error address out of bounds.
RTTAB%	Bit 33=1	4	Trap address out of bounds.
RTPNB%	Bit 34=1	2	PI channel not currently available for BLKI/BLKO's.
RTPNA%	Bit 35=1	1	PI channel not available (restricted use by system).

3.8.1.4 Restrictions -

1) Devices may be chained onto any PI channel that is not used for BLKI/BLKO instructions by the system or by other real-time users using fast block mode. This includes the APR channel. Normally PI levels 1 and 2 are reserved by the system for magnetic tapes and DECtapes. PI level 7 is always reserved for the system.

2) Each device must be chained onto a PI level before the user program issues the CONO DEV, PIA to set the device onto the interrupt level. Failure to observe this rule or failure to set the device on the same PI level that was specified in the RTTRP UUO could hang the system.

3) If the CONSO bit mask is set up and one of the corresponding flags in a device is on, but the device has not been physically put on its proper PI level, a trap may occur to the user's interrupt service routine. This occurs because there is a CONSO skip chain for each PI level, and if another device interrupts whose CONSO instruction is further down the chain than that of the real-time device, the CONSO associated with the real-time device is executed. If one of the hardware device flags is set and the corresponding bit in the CONSO bit mask is set, the CONSO skips and a trap occurs to the user program even though the real-time device was not causing the interrupt on that channel. To avoid this situation the user can keep the CONSO bit mask in his user area (refer to Paragraph 3.8.1.1). This procedure allows the user to chain a device onto the interrupt level, keeping the CONSO bit mask zero until the device is actually put on the proper PI level with a CONO instruction. This situation never arises if the device flags are turned off until the CONO DEV, PIA can be executed.

4) The user should guard against putting programs on high priority interrupt levels which execute for long periods of time. These programs could cause real-time programs at lower levels to lose data.

5) The user program must not change any locations in the protected job data area (locations 20-114), because the user is running at interrupt level and full context switching is not performed.

6) If the user is using the BLKI/BLKO feature, he must restore the BLKI/BLKO pointer word before dismissing any end-of-block interrupts. This is accomplished with another RTTRP UUO or by directly modifying the absolute pointer word supplied by the first RTTRP UUO. Failure to reset the pointer word could cause the device to overwrite all of memory.

3.8.1.5 Removing Devices from a PI Channel - When PICHL=0 in the data block (see Paragraph 3.8.1.1), the device specified in the CONSO instruction is removed from the interrupt system. If the user removes a device from a PI chain, he must also remove the device from the PI level (CONO DEV, 0).

A RESET, EXIT, or RUN UUO from the timesharing levels removes all devices from the interrupt levels (see Paragraph 3.2.2.4). These UUOs cause a CONO DEV, 0 to be executed before the device is removed. Monitor commands that issue implicit RESETS also remove real-time devices (e.g., R, RUN, GET, CORE 0, SAVE, SSAVE).

3.8.1.6 Dismissing the Interrupt - The user program must always dismiss the interrupt in order to allow monitor to properly restore the state of the machine. The interrupt may be dismissed with any UUO other than the RTTRP UUO or, on the KA10, any instruction that traps to absolute location 60. The standard method of dismissing the interrupt is with a UJEN instruction (op code 100). This instruction gives the fastest possible dismissal.

3.8.1.7 Examples

```
          ********** EXAMPLE 1 **********
                     SINGLE MODE

          TITLE RTSNGL - PAPER TAPE READ TEST USING CONSO CHAIN

          PIOFF=400                  ;TURN PI SYSTEM OFF
          PION=200                   ;TURN PI SYSTEM ON
          TAPE=400                   ;NO MORE TAPE IN READER IF TAPE=0
          BUSY=20                    ;DEVICE IS BUSY READING
          DONE=10                    ;A CHARACTER HAS BEEN READ
PDATA:    Z                          ;LOCATION WHERE DATA IS READ INTO
```

```
PTRTST: RESET                    ;RESET THE PROGRAM
        MOVE [XWD 1,1]           ;LOCK BOTH HIGH AND LOW SEGMENTS
        LOCK                     ;LOCK THE JOB IN CORE
        JRST FAILED              ;LOCK UUO FAILED
        SETZM PTRCSO             ;MAKE SURE CONSO BITS ARE ZERO
        SETZM DONFLG             ;INITIALIZE DONE FLAG
        MOVEI RTBLK              ;GET ADDRESS OF REAL TIME DATA BLOCK
        RTTRP                    ;PUT REAL TIME DEVICE ON THE PI LEVEL
        JRST FAILED              ;RTTRP UUO FAILED
        MOVEI 1,DONE             ;SET UP CONSO BIT MASK
        HLRZ 2,RTBLK             ;GET PI NUMBER FROM RTBLK
        TRO 2,BUSY               ;SET UP CONO BITS TO START TAPE GOING
        CONO PI,PIOFF            ;GUARD AGAINST ANY INTERRUPTS
        MOVEM 1,PTRCSO           ;STORE CONSO BIT MASK
        CONO PTR,(2)             ;TURN PTR ON
        CONO PI,PION             ;ALLOW INTERRUPTS AGAIN
        MOVEI 5                  ;SET UP TO SLEEP FOR 5 SECONDS
        SLEEP
        SKIPN DONFLG             ;HAVE WE FINISHED READING THE TAPE
        JRST .-3                 ;NO GO BACK TO SLEEP
        EXIT

RTBLK:  XWD 5,TRPADR             ;PI CHANNEL AND TRAP ADDRESS
        EXP APRTRP               ;APR ERROR TRAP ADDRESS
        CONSO PTR,@PTRCSO        ;INDIRECT CONSO BIT MASK = PTRCSO
        Z                        ;NO BLKI/O INSTRUCTION

PTRCSO: Z                        ;CONSO BIT MASK
DONFLG: Z                        ;PI LEVEL TO USER LEVEL COMM.
RTBLK1: Z                        ;DATA BLOCK TO REMOVE PTR
        Z                        ;FROM PI CHANNEL
        CONSO PTR,0
        Z

TRPADR: CONSO PTR,TAPE           ;END OF TAPE?
        JRST TDONE               ;YES, GO STOP JOB
        DATAI PTR,PDATA          ;READ IN DATA WORD
        UJEN                     ;DISMISS THE INTERRUPT

APRTRP: Z                        ;APR ERROR TRAP ADDRESS
TDONE:  MOVEI RTBLK1             ;SET UP TO REMOVE PTR
        CONO PTR,0               ;TAKE DEVICE OFF HARDWARE PI LEVEL
        RTTRP                    ;REMOVE FROM SOFTWARE PI LEVEL
        JFCL                     ;IGNORE ERRORS
        SETOM DONFLG             ;MARK THAT READ IS OVER
        SETZM PTRCSO             ;CLEAR CONSO BIT MASK
        UJEN                     ;DISMISS THE INTERRUPT

FAILED: TTCALL 3,[ASCIZ/RTTRP UUO FAILED!/]
        EXIT

        END PTRTST
```

```
          ********** EXAMPLE 2 **********
                  FAST BLOCK MODE

          TITLE RTFBLK - PAPER TAPE READ TEST IN BLKI MODE

          TAPE=400                  ;NO MORE TAPE IN READER IF TAPE=0
          BUSY=20                   ;DEVICE IS BUSY READING
          DONE=10                   ;A CHARACTER HAS BEEN READ
BLKTST:   RESET                     ;RESET THE PROGRAM
          MOVE [XWD 1,1]            ;LOCK BOTH HIGH AND LOW SEGMENTS
          LOCK                      ;LOCK THE JOB IN CORE
          JRST FAILED               ;LOCK UUO FAILED
          SETZM DONFLG              ;INITIALIZE DONE FLAG
          MOVEI RTBLK               ;GET ADDRESS OF REAL TIME DATA BLOCK
          RTTRP                     ;PUT REAL TIME DEVICE ON THE PI LEVEL
          JRST FAILED               ;RTTTP UUO FAILED
          HLRZ 2,RTBLK              ;GET PI NUMBER FROM RTBLK
          TRO 2,BUSY                ;SET UP CONO BITS TO START TAPE GOING
          CONO PTR,(2)              ;TURN PTR ON
          MOVEI 5                   ;SET UP TO SLEEP FOR 5 SECONDS
          SLEEP
          SKIPN DONFLG              ;HAVE WE FINISHED READING THE TAPE
          JRST .-3                  ;NO GO BACK TO SLEEP
          EXIT

RTBLK:    XWD 6,TRPADR              ;PI CHANNEL AND TRAP ADDRESS
          EXP APRTRP                ;APR ERROR TRAP ADDRESS
          BLKI PTR,POINTR           ;READ A BLOCK AT A TIME
          Z

POINTR:   IOWD 5,TABLE              ;POINTER FOR BLKI INSTRUCTION
OPOINT:   IOWD 5,TABLE              ;ORIGINAL POINTER WORD FOR BLKI
TABLE:    BLOCK 5                   ;TABLE AREA FOR DATA BEING READ
DONFLG:   Z                         ;PI LEVEL TO USER LEVEL COMM.
RTBLK1:   Z                         ;DATA BLOCK TO REMOVE PTR
          Z                         ;FROM PI CHANNEL
          CONSO PTR,0
          Z

TRPADR:   CONSO PTR,TAPE            ;END OF TAPE?
          JRST TDONE                ;YES, GO STOP JOB
          MOVE OPOINT               ;GET ORIGINAL POINTER WORD
          MOVEM POINTR              ;RESTORE BLKI POINTER WORD
          UJEN                      ;DISMISS THE INTERRUPT

APRTRP:   Z                         ;APR ERROR TRAP ADDRESS
TDONE:    MOVEI RTBLK1              ;SET UP TO REMOVE PTR
          CONO PTR,0                ;TAKE DEVICE OFF HARDWARE PI LEVEL
          RTTRP                     ;REMOVE FROM SOFTWARE PI LEVEL
          JFCL                      ;IGNORE ERRORS
          SETOM DONFLG              ;MARK THAT READ IS OVER
          UJEN                      ;DISMISS THE INTERRUPT

FAILED:   TTCALL 3,[ASCIZ/RTTRP UUO FAILED!/]
          EXIT

          END BLKTST
```

```
          ********** EXAMPLE 3 **********
                  NORMAL BLOCK MODE

          TITLE RTNBLK - PAPER TAPE READ TEST IN BLKI MODE

          TAPE=400                ;NO MORE TAPE IN READER IF TAPE=0
          BUSY=20                 ;DEVICE IS BUSY READING
          DONE=10                 ;A CHARACTER HAS BEEN READ

BLKTST:   RESET                   ;IO RESET
          MOVE [XWD 1,1]          ;LOCK BOTH HIGH AND LOW SEGMENTS
          LOCK                    ;LOCK THE JOB IN CORE
          JRST FAILED             ;LOCK UUO FAILED
          MOVEI RTBLK1            ;GET ADDRESS OF REAL TIME BLOCK
          RTTRP                   ;GET USER IOT PRIVILEGE
          JRST FAILED             ;UUO FAILED!
          CONO PTR,0              ;CLEAR ALL PTR FLAGS
          SETZM DONFLG            ;INITIALIZE DONE FLAG
          MOVEI RTBLK             ;GET ADDRESS OF REAL TIME DATA BLOCK
          RTTRP                   ;PUT REAL TIME DEVICE ON THE PI LEVEL
          JRST FAILED             ;RTTRP UUO FAILED
          MOVE POINTR             ;GET RELOCATED POINTER WORD FOR LATER
          MOVEM OPOINT            ;STORE FOR INTERRUPT LEVEL USE
          HLRZ 2,RTBLK            ;GET PI NUMBER FROM RTBLK
          TRO 2,BUSY              ;SET UP CONO BITS TO START TAPE GOING
          CONO PTR,(2)            ;TURN PTR ON
          MOVEI 5                 ;SET UP TO SLEEP FOR 5 SECONDS
          SLEEP
          SKIPN DONFLG            ;HAVE WE FINISHED READING THE TAPE
          JRST .-3                ;NO GO BACK TO SLEEP
          EXIT

RTBLK:    XWD 6,TRPADR            ;PI CHANNEL AND TRAP ADDRESS
          EXP APRTRP              ;APR ERROR TRAP ADDRESS
          CONSO PTR,DONE          ;WAIT ONLY FOR DONE FLAG
          BLKI PTR,POINTR         ;READ A BLOCK AT A TIME

POINTR:   IOWD 5,TABLE            ;POINTER FOR BLKI INSTRUCTION
OPOINT:   Z
TABLE:    BLOCK 5                 ;TABLE AREA FOR DATA BEING READ
DONFLG:   Z                       ;PI LEVEL TO USER LEVEL COMM.
RTBLK1:   Z                       ;DATA BLOCK TO REMOVE PTR
          Z                       ;FROM PI CHANNEL
          CONSO PTR,0
          Z

TRPADR:   CONSO PTR,TAPE          ;END OF TAPE?
          JRST TDONE              ;YES, GO STOP JOB
          MOVE OPOINT             ;GET ORIGINAL POINTER LOCATION
          MOVEM POINTR            ;STORE IN POINTER LOCATION
          UJEN                    ;DISMISS THE INTERRUPT

APRTRP:   Z                       ;APR ERROR TRAP ADDRESS
TDONE:    MOVEI RTBLK1            ;SET UP TO REMOVE PTR
          CONO PTR,0              ;TAKE DEVICE OFF HARDWARE PI LEVEL
          RTTRP                   ;REMOVE FROM SOFTWARE PI LEVEL
          JFCL                    ;IGNORE ERRORS
          SETOM DONFLG            ;MARK THAT READ IS OVER
          UJEN                    ;DISMISS THE INTERRUPT

FAILED:   TTCALL 3,[ASCIZ/RTTRP UUO FAILED!/]
          EXIT

          END BKJTST
```

3.8.2 RTTRP Executive Mode Trapping

In special cases, the real-time user requires a faster response time than that offered by the RTTRP UUO when executed in user mode. To accommodate these cases, the user can specify a special status bit in the RTTRP UUO call, which gives the program control in exec mode (refer to Paragraph 2.1.3). Exec-mode trapping gives response times of less than 10 µs to real-time interrupts. To use this exec-mode trapping, the job must have real-time privileges (granted by LOGIN) and be locked in core (accomplished by the LOCK UUO). On KI10-based systems, the job must also be mapped contiguously in exec virtual memory (refer to the LOCK UUO). The privilege bits required are:

1) JP.TRP (Bit 15)
2) JP.LCK (Bit 14)
3) JP.RTT (Bit 13)

Several restrictions must be placed on user programs in order to achieve this level of response. On receipt of an interrupt, program control is transferred to the user's real-time program without saving ACs and with the processor in exec mode. Therefore, the user program must save and restore all ACs that are used, must not execute any UUOs, and cannot leave exec mode. This means that the programs must be self-relocating (i.e., through the use of an index or base register).

CAUTION

Improper use of the exec mode feature of the RTTRP UUO can cause the system to fail in a number of ways. Unlike the user mode feature of RTTRP, errors are not protected against since the programs run in exec mode with no ACs saved.

To specify RTTRP exec-mode trapping, bit 17 of the second word in the data block (RTBLK) must be set to 1. This implies that no context switching is to be done and that a JSR TRPADR is to be used to enter the user's real-time interrupt routine. The user program must save and restore all ACs and should dismiss the interrupt with a JRSTF @ TRPADR. This instruction must be set up prior to the start of the real-time device as an absolute or unrelocated instruction. This can be done because the LOCK UUO returns the absolute addresses of the low and high segments after the job is locked in a fixed place in memory.

The exec-mode trapping feature can be used with any of the standard forms of the RTTRP UUO: single mode, normal block mode, and fast block mode.

3.8.2.1 Example

```
          TITLE RTEXEC

          PIA=5
          DONE=1Ø
          BUSY=2Ø
          TAPE=4ØØ
          I=1
          AC=2
          OPDEF HIBERNATE [CALLI 72]

RTEXEC:   RESET                         ;RESET THE PROGRAM
          SETZM DONFLG                  ;INITIALIZE THE DONE FLAG
          MOVE AC,[XWD 1,1]
          LOCK AC,                      ;LOCK THE JOB IN CORE
                                        ;ABSOLUTE ADDRESS OF JOB IS RETURNED
                                        ;IN AC
          JRST FAILED                   ;ERROR RETURN
          HRRZS AC                      ;GET ONLY LOW SEGMENT ADDRESS
          LSH AC,9                      ;JUSTIFY ADDRESS
          MOVEM AC,INDEX                ;SAVE BASE ADDRESS FOR USE AT INTERRUPT
                                        ;LEVEL
          ADDM AC,EXCHWD                ;RELOCATE INTERRUPT LEVEL PROGRAM
          ADDM AC,JENWD                 ;RELOCATE RETURN INSTRUCTION
          MOVEI AC,RTBLK                ;CONNECT REAL TIME DEVICE
          RTTRP AC,                     ;TO THE PI SYSTEM
          JRST FAILED                   ;RTTRP UUO FAILED
          CONO PTR,2Ø+PIA               ;START REAL TIME DEVICE READING
SLEEP:    MOVEI AC,↑D1ØØØ               ;SLEEP
          HIBERNATE AC,                 ;FOR 1Ø MILLISECONDS
          JRST FAILED                   ;FAILED
          SKIPN DONFLG                  ;IS THE INTERRUPT LEVEL PROGRAM DONE
          JRST SLEEP                    ;NO, GO BACK TO SLEEP
          EXIT                          ;YES, EXIT

RTBLK:    XWD PIA,TRPADR
          XWD 1 APRTRP                  ;BIT 17 SAYS TRAP IN EXEC MODE
          CONSO  TR,DONE
          Ø

TRPADR:   Ø                             ;JSR TRPADR IS DONE UPON INTERRUPT
EXCHWD:   EXCH I,INDEX                  ;SET UP INDEX REGISTER
          CONSO PTR,TAPE                ;TAPE FINISHED?
          JRST TDONE(I)                 ;YES, STOP THE READER
          DATAI PTR,PDATA(I)            ;NO, READ IN THE NEXT CHARACTER
RETURN:   EXCH I,INDEX(I)               ;RESTORE AC'S USED
JENWD:    JRSTF @TRPADR                 ;DISMISS THE INTERRUPT

APRTRP:   Ø                             ;APR ERRORS WILL TRAP HERE
TDONE:    CONO PTR,Ø                    ;TAKE THE READER OFF LINE
          SETOM DONFLG(I)               ;MARK THAT THE TAPE IS FINISHED
          JRST RETURN(I)                ;GO DISMISS THE INTERRUPT

FAILED:   TTCALL 3,[ASCIZ/UUO FAILURE/]
          EXIT

DONFLG:   Ø                             ;FLAG TO SPECIFY END OF JOB
PDATA:    Ø                             ;DATA WORD
INDEX:    Ø                             ;BASE INDEX REGISTER

          END RTEXEC
```

3.8.3 TRPSET AC, or CALLI AC, 25[1]

The TRPSET feature may be used to guarantee some of the fast response requirements of real-time users. In order to achieve fast response to interrupts, this feature temporarily suspends the running of other jobs during its use. This limits the class of problems to be solved to cases where the user wants to transfer data in short bursts at predefined times. Therefore, because the data transfers are short, the time during which timesharing is stopped is also short, and the pause probably will not be noticed by the timesharing users.

The TRPSET UUO allows the user program to gain control of the interrupt locations. If the user does not have the TRPSET privileges (JP.TRP, bit 15), an error return to the next location after the CALLI is always given, and the user remains in user mode. Timesharing is turned back on. If the user has the TRPSET privileges, the central processor is placed in user I/O mode. If AC contains zero, time-sharing is turned on if it was turned off. If the LH of AC is within the range 40 through 57 of the central processor, all other jobs are stopped from being scheduled and the specified executive PI location (40-57) is patched to trap directly to the user. In this case, the monitor moves the contents of the relative location specified in the right half of AC, converts the user virtual address to the equivalent exec virtual address, and stores the address in the specified executive PI location. On a KI10-based system, this requires that the user segment accessed during the interrupt be locked and mapped contiguously in the exec virtual memory (refer to the LOCK UUO). If the segment does not meet these requirements, the error return is given.

On a multiprocessor system, the TRPSET UUO applies to the processor specified by the job's CPU specification (refer to the SET CPU command or the SET UUO). If the specification indicates only CPU1, an error return is given if the job is not locked in core. When the specification indicates more than one processor, the specification is changed to indicate CPU0 (the master processor).

Thus, the user can set up a priority interrupt trap into his relocated core area. On a normal return, AC contains the previous contents of the address specified by LH of AC, so that the user program may restore the original contents of the PI location when the user is through using this UUO. If the LH of AC is not within the range 40 through 57, an error return is given just as if the user did not have the privileges. The basic call is:

```
        MOVE AC,[XWD N, ADR]
        TRPSET AC,
        ERROR RETURN
        NORMAL RETURN
        .
        .
        .
ADR:    JSR TRAP                        ;Instruction to be stored
                                        ;in exec PI location
                                        ;after relocation added to it.
TRAP:   0                               ;Here on interrupt from exec.
```

[1]This UUO depends on FTTRPSET which is normally off in the DECsystem-1040.

The monitor assumes that user ADR contains either a JSR U or BLKI U, where U is a user virtual address; consequently, the monitor adds a relocation to the contents of location U to make it an absolute IOWD (i.e., an exec virtual address). Therefore, a user should reset the contents of U before every TRPSET call.

A RESET UUO returns the user to normal user mode. The following instruction sequence is used to place the real-time device RTD on channel 3.

```
INT46:    BLKI RTD,INBLOK               ;relocation constant
                                        ;for user is added
INT47:    JSR XITINT                    ;to RH when instructions
          .                             ;are placed into 46 and 47.
          .
          .
START:    MOVEI AC,INT46
          HRLI AC,46
          TRPSET AC,
          JRST EXITR                    ;error return
          MOVE AC, [XWD 47, INT47]      ;normal return
          TRPSET AC,
          JRST EXITR                    ;error return
          .                             ;normal return
          .
          .
XITINT:   0                             ;PC saved
          . . .                         ;interrupt dismiss routine
```

To maintain compatibility between a KA10-based system and a KI10-based system, the interrupt routine should be executed in exec mode. However, for convenience, the routine can be executed in user mode in order to avoid relocation to exec virtual memory. This is possible on KA10-based systems if care is taken when dismissing the interrupt (see example below). On KI10-based systems, if there is a possibility that the interrupt may occur during the job's background processing, the interrupt routine must be executed in exec mode (and thus must be locked and exec-mapped with the LOCK UUO). In particular, if the job is executing a UUO at background level, the use of UJEN at interrupt level may cause an error. On KI10-based systems it is recommended that the TRPSET interrupt routines always be coded to run in exec mode (refer to the RTTRP UUO for programming techniques.)

On KA10-based system, the interrupt routine can be coded to run in user mode if the following procedure is observed. If the interrupt occurs while some other part of the user's program is running, the user may dismiss from the interrupt routine with a JEN @ XITINT. However, if the machine is in exec mode, a JEN instruction issued in user mode does not work because of memory relocation. This is solved by a call to UJEN (op code 100). This UUO causes the monitor to dismiss the interrupt from exec mode. In this case, the address field of the UJEN instruction is the user location when the return PC is stored (i.e., UJEN XITINT). The following sequence enables the user program to decide whether it can issue a JEN to save time or dismiss the interrupt with a UUO call.

Example (KA10-based system only):

```
XITINT:   0                                  ;PC with bits in LH

          JRST 1,.+1                         ;essential instruction.
                                             ;returns machine to
                                             ;user mode.
          MOVEM AC, SAVEAC                   ;save accumulator AC
          .                                  ;service interrupt here
          .
          .

          MOVE AC, XITINT                    ;get PC with bits
          SETZM EFLAG

          TLNN AC, 10000                     ;was machine in user
                                             ;mode at entry?
          SETOM EFLAG                        ;no
          MOVE AC, SAVEAC                    ;RESTORE saved AC
          SKIPE EFLAG
          UJEN XITINT                        ;not in user mode at entry
          JEN @ XITINT

SAVEAC:   0

EFLAG:    0
```

On entering the routine from absolute 47 with a JSR to XITINT + REL (where REL. is the relocation constant), the processor enters exec mode. The first executed instruction in the user's routine must, therefore, reset the user mode flag, thereby enabling relocation and protection. The user must proceed with caution when changing channel interrupt chains under timesharing, making certain that the real-time job can co-exist with other timesharing jobs.

3.8.4 UJEN (Op Code 100)

This op code dismisses a user I/O mode interrupt if one is in progress. If the interrupt is from user mode, a JRST 12, instruction dismisses the interrupt. If the interrupt came from executive mode, however, this operator is used to dismiss the interrupt. The monitor restores all accumulators, and executes JEN @ U where user location U contains the program counter as stored by a JSR instruction when the interrupt occurred.

3.8.5 HPQ AC, or CALLI AC, 71[1]

The HPQ UUO is used by privileged users to place their jobs in a high-priority scheduler run queue. These queues are always scanned by the scheduler before the normal run queues, and any runnable job

---

[1] This UUO depends on FTHPQ which is normally off in the DECsystem-1040.

in one of these queues is executed before all other jobs in the system. In addition, these jobs are given preferential access to sharable resources (e.g., shared device controllers). Thus, real-time associated jobs can receive fast response from the timesharing scheduler.

Jobs in high-priority queues are not examined for swap-out until all other queues have been scanned. If a job in a high-priority queue must be swapped, the lowest priority job is transferred first, and the highest priority job last. If the highest priority job is swapped, then that job is the first to be swapped in for immediate execution. Therefore, in addition to being scanned before all other queues for job execution, the high-priority queues are examined after all other queues for swap-out and before all queues for swap-in.

The HPQ UUO requires as an argument the high-priority queue number of the queue to be entered. The lowest high-priority queue is 1, and the highest priority queue is equivalent to the number of queues that the system is built for. The call is as follows:

```
        MOVE AC, HPQNUM             ;get high-priority queue number
        HPQ AC,                     ;or CALLI AC, 71
        error return
        normal return
```

On an error return, AC contains -1 if the user did not have the correct privileges. The privilege bits are 6 through 9 in the privilege word (.GTPRV). These four bits specify a number from 0-17 octal, which is the highest priority queue attainable by the user.

On a normal return, the job is in the desired high-priority queue. A RESET or an EXIT UUO returns the job to the high-priority queue specified in the last SET HPQ command. A queue number of 0 as an argument places the job back to the timesharing level.

3.9 METER.AC, OR CALLI AC, 111[1]

This UUO provides a mechanism for system performance metering by allowing privileged users to dynamically select and collect performance statistics from the monitor. The multifunction UUO controls all aspects of the metering facility in order that the user can collect, present, or reduce data for performance analysis or can tune individual jobs or the entire system. The METER. UUO requires JP.MET (bit 3) to be set in the privilege word .GTPRV.

---

[1] This UUO depends on FTMETR which is normally off in the DECsystem-1040.

The general call is:

```
MOVE AC, [XWD N, ADR]
METER.AC,                    ;or CALLI AC, 111
error return
normal return
```

where

N is the number of arguments in the argument list.
ADR is the beginning of the argument list.

If N is 0, the default number of arguments depends on the particular function used. Arguments in the list can be 1) arguments for the monitor, 2) values returned from the monitor, or 3) a combination of both. The first word of the argument block is the code for the particular function. The detailed descriptions of the various functions of the METER. UUO are presented in the METER. Specification in the Software Notebooks; the following is a list of the functions available.

Function Code	Name	Description
0	.MEFCI	Initialize meter channel
1	.MEFCS	Obtain meter channel status
2	.MEFCR	Release meter channel
3	.MEFPI	Initialize meter points
4	.MEFPS	Obtain meter point status
5	.MEFPR	Release meter points

On an error return, the appropriate error code is returned in AC. Refer to the METER. Specification for the error codes.

On a normal return, AC is perserved.

CHAPTER 4
I/O PROGRAMMING

All user-mode I/O programming is controlled by monitor programmed operators. I/O is directed by

a. Associating a device and a ring of buffers with one of the user's I/O channels (INIT, OPEN).
b. Optionally selecting a file (LOOKUP, ENTER).
c. Passing buffers of data to or from the user program (IN, INPUT, OUT, OUTPUT).

Device specification may be delayed from program-generation time until program-run time because the monitor

a. Allows a logical device name to be associated with a physical device (ASSIGN or MOUNT monitor command).
b. Treats operations that are not pertinent to a given device as no-operation code.

For example: a rewind directed to a line printer does nothing, and file selection operations for devices without a filename directory are always successful.

4.1 I/O ORGANIZATION

4.1.1 Files

A file is an ordered set of data on a peripheral device. The extent of a file on input is determined by an end-of-file condition dependent on the device. For example: a file is terminated by reading an end-of-file gap from magnetic tape, by an end-of-file card from a card reader, or by depressing the end-of-file switch on a card reader (refer to Chapter 5). The extent of a file on output is determined by the amount of information written by the OUT or OUTPUT programmed operators up through and including the next CLOSE or RELEAS operator.

4.1.2 Job I/O Initialization

The monitor programmed operator

CALL [SIXBIT /RESET/] or CALLI 0

should normally be the first instruction in each user program. It immediately stops all I/O transmissions on all devices without waiting for the devices to become inactive. All device allocations made by the INIT and OPEN operators are cleared and, unless the devices have been assigned by the ASSIGN or MOUNT monitor command, the devices are returned to the monitor facilities pool. The content of the left half of .JBSA (program break) is stored in the right half of .JBFF so that the user buffer area is reclaimed if the program restarts. The left half of .JBFF is cleared. Any files that have not been closed are deleted on disk. Any older version with the same filename remains. The user-mode write-protect bit is automatically set if a high segment exists, whether it is sharable or not; therefore, a program cannot inadvertently store into the high segment. Additional functions of the RESET UUO include 1) unlocking the job if it was locked, 2) releasing any real-time devices, 3) resetting any high-priority queues set by the HPQ UUO to the value set by the HPQ command, 4) resuming timesharing if it was stopped as a result of a TRPSET UUO with a non-zero argument, 5) resetting the action of the HIBER and APRENB UUOs, and 6) clearing all PC flags except USRMOD.

4.2 DEVICE SELECTION

For all I/O operations, a specific device must be associated with a software I/O channel. This specification is made by an argument of the INIT or the OPEN programmed operators. The INIT or the OPEN programmed operators may specify a device with a logical name that is associated with a particular physical device by the ASSIGN or MOUNT monitor command. Some system programs, e.g., LOGOUT, require I/O to specific physical devices regardless of what logical names have been assigned. Therefore, on an OPEN UUO, if the sign bit of word 0 of the OPEN block is 1 (UU.PHS), the device name is taken as a physical name only, and logical names are not searched. A given device remains associated with a software I/O channel until released (refer to Paragraph 4.8.1) or until another INIT or OPEN is performed for that channel. Devices are separated into two categories: those with no filename directory (refer to Chapter 5) and those with at least one filename directory (refer to Chapter 6).

Assignable devices (i.e., non-disk and non-spooled devices) in the monitor's pool of available resources are designated as being either unrestricted or restricted. An unrestricted device can be assigned directly by any job via the ASSIGN command or INIT or OPEN UUO. A restricted device can be assigned directly only by a privileged job (i.e., a job logged in under [1,2] or running with the JACCT bit set). However, a non-privileged user can have a restricted device assigned to him indirectly via the MOUNT command. This command allows operator intervention for the selection or denial of a particular device; thus the operator can control the use of assignable devices for the non-privileged user. This is particularly useful when there are multiprogramming batch and interactive jobs competing for the same devices. The restricted status of a device is set or removed by the operator with the OPSER commands :RESTRICT and :UNRESTRICT, which use the privileged UUOs, DVRST. and DVURS. (refer to UUOPRV in the DECsystem-10 Software Notebooks).

4.2.1 Nondirectory Devices

For nondirectory devices (e.g., card reader and punch, line printer, paper-tape reader and punch, and user terminal), selection of the device is sufficient to allow I/O operations over the associated

software channel. All other file specifiers, if given, are ignored. Magnetic tape, a nondirectory device, requires, in addition to the name, that the tape be properly positioned. It is advisable to use the programmed operators that select a file, so that a directory device may be substituted for a nondirectory device at run time.

4.2.2 Directory Device

For directory devices (e.g., DECtape and disk), files are addressable by name. If the device has a single file directory (e.g., DECtape) the device name and filename are sufficient information to determine a file. If the device has multiple file directories (e.g., disk) the name of the file directory must also be specified. These names are specified as arguments to the LOOKUP, ENTER, and RENAME programmed operators.

4.2.3 Device Initialization

The OPEN (operation code 050) and INIT (operation code 041) programmed operators initialize a device and associate it with a software I/O channel number for the job. These UUOs perform almost identical functions; the OPEN UUO is a reentrant form of INIT and is preferred for this reason. In addition to the device name, these programmed operators accept, as arguments, an initial file status and the location of the input and output buffer headers. The calls are:

```
OPEN D,SPEC                 INIT D,STATUS
error return                SIXBIT /dev/
normal return               XWD OBUF, IBUF
      .                     error return
      .                     normal return
SPEC:  EXP STATUS
       SIXBIT /dev/
       XWD OBUF, IBUF
```

The normal return is taken if a device is selected, and if the device is associated with a software I/O channel. The error return is taken if the requested device is in use, if the requested device does not exist, or if the device is restricted and has not been assigned with the MOUNT command.

4.2.3.1 Data Channel - These programmed operators establish a correspondence between the device and a 4-bit channel number, D. Most of the other input/output operators require this channel number as an argument. If a device is already assigned to channel D, it is released (refer to Paragraph 4.8.1).

4.2.3.2 Device Name - The device name, dev, is either a logical or physical device name, with logical names taking precedence over physical names. With multiple stations, the method of device selection depends on the format of the specified SIXBIT device name.

If devn (e.g., LPT7, CDR3) is specified, the monitor attempts to select the device specifically requested.

If devSnn (e.g., CDPS14, PTPS12) is specified, the monitor attempts to select any device of the desired type at the requested station. If a device of the desired type has been previously assigned to this job at the requested station and is not INITed on another channel, it will be selected in preference to an unassigned device.

If dev (e.g., LPT, DTA) is specified, the monitor attempts to select a device of the desired type at the job's logical station. If all devices of this type are in use, the error return is taken. If no device of the desired type exists at the user's logical station, the monitor attempts to select the device at the central station. If the desired type of device has already been assigned to the job at the appropriate station (either the job's logical station or the central station) and is not INITed on another channel, it will be selected instead of an unassigned device.

In non-disk systems, if the specified device is the system device SYS, the job is placed into a system device wait queue and continues to run when SYS becomes available. In disk systems where the system device is one or more file structures, control returns immediately.

The job may pause with the message

?STATION nn NOT IN CONTACT

if the requested station is not in contact with the central station. After station nn has established contact with the central station, the user types CONTINUE for a return to job execution.

4.2.3.3 Initial File Status - The file status, including the data mode, is set to the value of the symbol STATUS. Thereafter, bits are set by the monitor and may be tested and reset by the user via monitor programmed operators. Bits 30-35 of the file status are normally set by an OPEN or INIT UUO. Refer to Table 4-3 in Paragraph 4.6.2 for the file status bits. If the data mode is not legal (refer to Chapters 5 and 6) for the specified device, the job is stopped and the monitor prints

ILL DEVICE DATA MODE FOR DEVICE dev AT USER addr,

where dev is the physical name of the device and addr is the location of the OPEN or INIT operator on the user's terminal. The terminal is left in monitor mode.

4.2.3.4 Data Modes - Data transmissions are either unbuffered or buffered. (Unbuffered mode is sometimes referred to as dump mode.) The mode of transmission is specified by a 4-bit argument to the INIT, OPEN, or SETSTS programmed operator. Tables 4-1 and 4-2 summarize the data modes.

Table 4-1
Buffered Data Modes

Octal Code	Name	Meaning
0	.IOASC	ASCII. Seven bit bytes packed left justified, five characters per word.
1	.IOASL	ASCII line. Same as 0, except that the buffer is terminated by a FORM, VT, LINE-FEED, or ALTMODE character. Differs from ASCII on TTY (half-duplex software) and PTR only.
2-7		Unused.
10	.IOIMG	Image. A device dependent mode. Thirty-six bit bytes. The buffer is filled with data exactly as supplied by the device.
11-12		Unused.
13	.IOIBN	Image binary. Thirty-six bit bytes. This mode is similar to binary mode, except that no automatic formatting or checksumming is done by the monitor.
14	.IOBIN	Binary. Thirty-six bit bytes. This is blocked format consisting of a word count, n (the right half of the first data word of the buffer), followed by n 36-bit data words. Checksum for cards and paper tape.

Table 4-2
Unbuffered Data Modes

Octal Code	Name	Meaning
15	.IOIDP	Image dump. A device dependent dump mode. Thirty-six bit bytes.
16	.IODPR	Dump as records without core buffering. Data is transmitted between any contiguous blocks of core and one or more standard length records on the device for each command word in the command list. Thirty-six bit bytes.
17	.IODMP	Dump one record without core buffering. Data is transmitted between any contiguous block of core and exactly one record of arbitrary length on the device for each command word in the command list. Thirty-six bit bytes.

4.2.3.5 Buffer Header - Symbols OBUF and IBUF, if non-zero specify the location of the first word of the 3-word buffer ring header block for output and input, respectively. Buffered data modes utilize a ring of buffers in the user area and the priority interrupt system to permit the user to overlap computation with his data transmission. Core memory in the user's area serves as an intermediate buffer

between the user's program and the device. The buffer storage mechanism consists of a 3-word buffer ring header block for bookkeeping and a data storage area subdivided into one or more individual buffers linked together to form a ring. During input operations, the monitor fills a buffer, makes that buffer available to the user's program, advances to the next buffer in the ring, and fills that buffer if it is free. The user's program follows the monitor, either emptying the next buffer if it is full or waiting for it to fill.

During output operations, the user's program and the monitor exchange roles; the user program fills the buffers and the monitor empties them. Only the headers that will be used need to be specified. For instance, the output header need not be specified, if only input is to be done. Also, data modes 15, 16, and 17 require no header. If either of the buffer headers or the 3-word block starting at location SPEC lies outside the user's allocated core area†, the job is stopped and the monitor prints

ILLEGAL UUO AT USER addr

(addr is the address of the OPEN or INIT operator) on the user's terminal, leaving the terminal in monitor mode.

The first and third words of the buffer header are set to zero. The left half of the second word is set up with the byte pointer size field in bits 6 through 11 for the selected device-data mode combination.

If the same device (other than disk) is INITed on two or more channels, the monitor retains only the buffer headers mentioned in the last INIT (a 0 specification does not override a previous buffer header specification). Other I/O operations to any of the channels involved act on the buffers mentioned in the last INIT previous to the I/O operations.

4.3 RING BUFFERS

4.3.1 Buffer Structure

The ring buffer (see Figure 4-1) is comprised of a buffer ring header block and buffer rings.

4.3.1.1 Buffer Ring Header Block - The location of the 3-word buffer ring header block is specified by an argument of the INIT and OPEN operators. Information is stored in the header by the monitor in response to the user execution of monitor programmed operators. The user's program finds all the information required to fill and empty buffers in the header. Bit position 0 of the first word of the header is a flag, which, if 1, means that no input or output has occurred for this ring of buffers. The

† Buffer headers may not be in the user's ACs; however, the buffer headers may be in location above .JBPFI (refer to Table 1-1 in Paragraph 1.2.1).

right half of the first word is the address of the second word of the buffer currently used by the user's program. The second word of the header contains a byte pointer to the current byte in the current buffer. The byte size is determined by the data mode. The third word of the header contains a number of bytes remaining in the buffer. A program may not use a single buffer header for both input and output, nor may a single buffer ring header be used for more than one I/O function at a time. Users cannot use the same buffer ring for simultaneous input and output; only one buffer ring is associated with each buffer ring header.

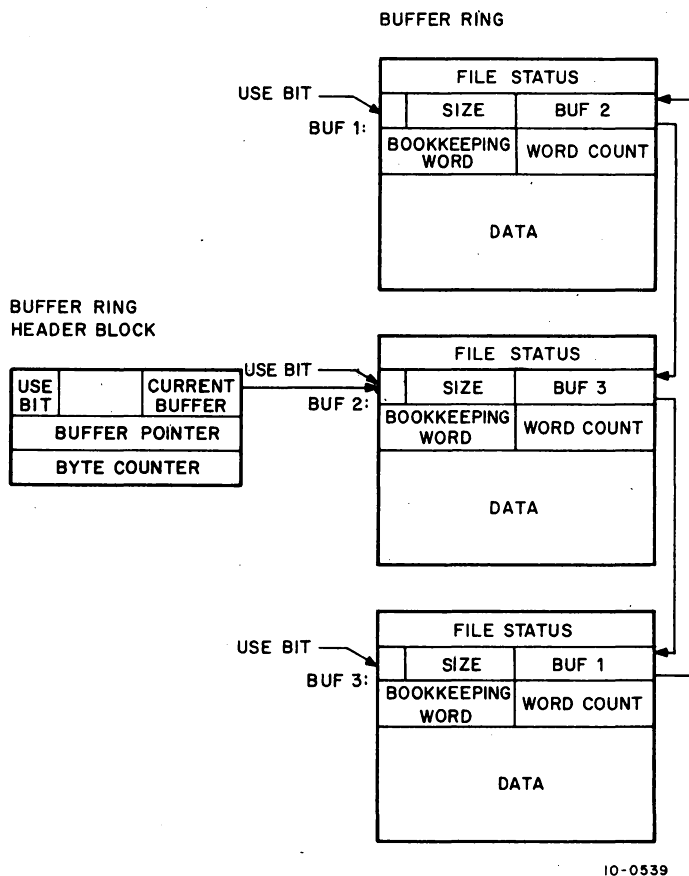


Figure 4-1 User's Ring of Buffers

4.3.1.2 Buffer Ring - The buffer ring is established by the INBUF and OUTBUF operators, or, if none exists when the first IN, INPUT, OUT, or OUTPUT operator is executed, a 2-buffer ring is set up. The effective address of the INBUF and OUTBUF operators specifies the number of buffers in the ring. The location of the buffer ring is specified by the contents of the right half of .JBFF in the user's job data area. The monitor updates .JBFF to point to the first location past the storage area.

All buffers in the ring are identical in structure. The right half of the first word contains the file status when the monitor advances to the next buffer in the ring (see Figure 4-2). Bit 0 of the second word of a buffer, the use bit, is a flag that indicates whether the buffer contains active data. This bit is set to 1 by the monitor when the buffer is full on input or being emptied on output, and set to 0 when the buffer is empty on output or is being filled on input. In other words, if the use bit = 0, the buffer is available to the filler; if the use bit = 1, the buffer is available to the emptier. The use bit prevents the monitor and the user's program from interfering with each other by attempting to use the same buffer simultaneously. Buffers are advanced by the UUOs and not by the user's program. The use bit in each buffer should never be changed by the user's program except by means of the UUOs. Bits 1 through 17 of the second word of the buffer contain the size of the data area of the buffer plus 1. The size of this data area depends on the device. The right half of the third word of the buffer is reserved for a count of the number of words that actually contain data. The left half of this word is reserved for other bookkeeping purposes, depending on the particular device and the data mode.

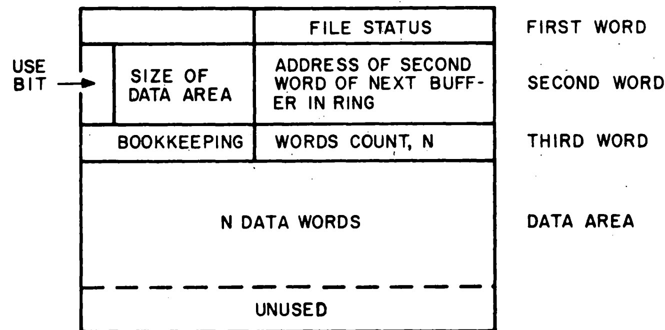


Figure 4-2 Detailed Diagram of Individual Buffer

4.3.2 Buffer Initialization

Buffer data storage areas may be established by the INBUF and OUTBUF programmed operators, or by the first IN, INPUT, OUT, or OUTPUT operator, if none exists at that time, or the user may set up his own buffer data storage area.

4.3.2.1 Monitor Generated Buffers - Each device has an associated standard buffer size (refer to Chapters 5 and 6). The monitor programmed operators INBUF D,n (operation code 064) and OUTBUF D,n (operation code 065) set up a ring of n standard size buffers associated with the input and output buffer headers, respectively, specified by the last OPEN or INIT operator on data channel

D. If n = 0 on either INBUF or OUTBUF, the default number of buffers for the specified device is set up. If no OPEN or INIT operator has been performed on channel D, the monitor stops the job and prints

I/O TO UNASSIGNED CHANNEL AT USER addr

(addr is the location of the INBUF or OUTBUF operator) on the user's terminal leaving the terminal in the monitor mode.

The storage space for the ring is taken from successive locations, beginning with the location specified in the right half of .JBFF. This location is set to the program break, which is the first free location above the program area, by RESET. If there is insufficient space to set up the ring, the monitor automatically attempts to expand the user's core allocation by 1K. If this fails, the monitor stops the job and prints

ADDRESS CHECK FOR DEVICE dev AT USER addr

(dev is the physical name of the device associated with channel D and addr is the location of the INBUF or OUTBUF operator) on the user's terminal, leaving the terminal in monitor mode.

This message is also printed when an INBUF (OUTBUF) is attempted if the last INIT or OPEN UUO on channel D did not specify an input (output) buffer header.

The ring is set up by setting the second word of each buffer with a zero use bit, the appropriate data area size, and the link to the next buffer. The first word of the buffer header is set with a 1 in the ring use bit, and the right half contains the address of the second word of the first buffer.

4.3.2.2 User Generated Buffers - The following code illustrates an alternative to the use of the INBUF programmed operator. Analogous code may replace OUTBUF. This user code operates similarly to INBUF. SIZE must be set equal to the greatest number of data words expected in one physical record.

```
GO:     OPEN I,OPNBLK                       ;INITIALIZE ASCII MODE
          JRST NOTAVL                       ;THE 400000 IN THE LEFT HALF
        MOVE 0, [XWD 400000,BUF1+1]         ;MEANS THE BUFFER WAS NEVER
                                            ;REFERENCED.

                                            ;SET UP NON-STANDARD BYTE
        MOVEM 0, MAGBUF                     ;SIZE
        MOVE 0, [POINT BYTSIZ,0,35]
                                            ;MAGNETIC TAPE UNIT 0
        MOVEM 0, MAGBUF+1                   ;INPUT ONLY
        JRST CONTIN
```

(continued on next page)

```
OPNBLK:  0                          ;GO BACK TO MAIN SEQUENCE
         SIXBIT/MTA0/               ;SPACE FOR BUFFER RING HEADER
         XWD 0,MAGBUF               ;BUFFER 1, 1ST WORD UNUSED
MAGBUF:  BLOCK 3                    ;LEFT HALF CONTAINS DATA AREA
BUF1:    0                          ;SIZE+1, RIGHT HALF HAS
         XWD SIZE+1,BUF2+1          ;ADDRESS OF NEXT BUFFER
                                    ;SPACE FOR DATA, 1ST WORD
                                    ;RECEIVES WORD-COUNT. THUS
         BLOCK SIZE +1              ;ONE MORE WORD IS RESERVED
                                    ;THAN IS REQUIRED FOR DATA
                                    ;ALONE
                                    ;SECOND BUFFER
BUF2:    0
         XWD SIZE+1,BUF3+1          ;THIRD BUFFER
         BLOCK SIZE+1               ;RIGHT HALF CLOSES THE RING
BUF3:    0
         XWD SIZE+1,BUF1+1
         BLOCK SIZE+1
```

4.4 FILE SELECTION (LOOKUP and ENTER)

The LOOKUP (operation code 076) and ENTER (operation code 077) programmed operators select a file for input and output, respectively. These operators are not necessary for nondirectory devices; however, it is good programming practice to always use them so that directory devices may be substituted at run time (refer to ASSIGN command). The monitor gives the normal return for a LOOKUP or ENTER to a nondirectory device; therefore, user programs can be coded in a device-independent fashion.

4.4.1 The LOOKUP Operator

LOOKUP selects a file for input on channel D.

```
        LOOKUP D,E
        error return
        normal return
          .
          .
          .
     E: SIXBIT/file/              ;filename, 1 to 6 characters, left-justified
        SIXBIT/ext/               ;filename extension, 0 to 3
                                  ;characters, left-justified
           —                      ;The remaining words in the argument block
           —                      ;are ignored for nondirectory devices. Refer
                                  ;to Paragraph 6.1.5.1 for the DECtape
                                  ;LOOKUP and Paragraph 6.8.2.1 for the
                                  ;disk LOOKUP.
```

If no device has been associated with channel D by an INIT or OPEN UUO, the monitor stops the job, prints

I/O TO UNASSIGNED CHANNEL AT USER LOC addr

and returns the user's terminal to monitor mode. The input side of channel D is closed if not already closed. The output side is not affected.

On DECtape, LOOKUP searches the device directory as specified by an INIT. On disk, the user's file directory as specified by the contents of location E+3 is searched. Refer to Paragraph 6.1.5.1 for details of a DECtape LOOKUP and Paragraph 6.8.2.1 for details of a disk LOOKUP.

If the device is a directory device and the file is found, the normal return is taken and information concerning the file is returned in location E+1 through E+3. The normal return is always taken if the device associated with the channel D does not have a directory. The error return is taken if 1) the file is not found, 2) the file is found but the user does not have access to it (refer to Paragraph 6.2.3 for the description of file access codes), or 3) the device associated with channel D is a non-input device. Refer to Appendix E for the error codes returned in bits 18-35 of location E+1.

4.4.2 The ENTER Operator

ENTER selects a file for output on channel D.

```
        ENTER D, E
        error return
        normal return
        .
        .
        .
E:      SIXBIT/file/            ;filename, 1 through 6
                                ;characters, left-justified
        SIXBIT/ext/             ;filename extension, 0 through 3 characters,
                                ;left-justified
          ___                   ;The remaining words in the argument block are
          ___                   ;ignored for nondirectory devices. Refer to
                                ;Paragraph 6.1.5.2 for the DECtape ENTER
                                ;and Paragraph 6.8.2.1 for the disk ENTER.
```

If no device has been associated with channel D by an INIT or OPEN UUO, the monitor stops the job, prints

I/O TO UNASSIGNED CHANNEL AT USER LOC addr

and returns the user's terminal to monitor mode. The output side of channel D is now closed (if it was not closed); the input side is not affected. On DECtape, ENTER searches the device directory as specified by an INIT. On disk, the user's file directory, as specified by the contents of location E+3, is searched.

If the device does not have a directory, the normal return is always taken. On directory devices, if the file is found and is not being written or renamed, the file is deleted (the user must have access privileges to the file), and the storage space on the device is reclaimed. On DECtape, this deletion must occur immediately on ENTER to ensure that space is available for writing the new version of the

file. On disk, the deletion of the previous version does not occur until output CLOSE time, provided bit 30 of CLOSE is 0 (refer to Paragraph 4.7.7). Consequently, if the new file is aborted when partially written, the old version remains. The normal return is taken, and the monitor makes the file entry, and records file information.

The error return is taken if:

a. The filename in location E is 0.

b. The file is found but is being written or renamed.

c. The user does not have access to the file, as supplied by the file if it exists or by the UFD if the file does not exist.

d. The device associated with channel D is a non-output device.

Refer to Paragraph 6.8.2.1 for details of a disk ENTER and Paragraph 6.1.5.2 for details of a DECtape ENTER. Refer to Appendix E for the error codes returned in bits 18-35 of location E+1.

4.4.3 RENAME Operator

The RENAME (operation code 055) programmed operator is used

a. To alter the filename, filename extension, and file access privileges

b. To delete a file associated with channel D on a directory device

```
        RENAME D,E
        error return
        normal return
        .
        .
        .
E:      SIXBIT/file/        ;filename, 1 to 6 characters
        SIXBIT/ext/         ;filename extension, 0 to 3
                            ;characters.
        —                   ;The remaining words in the
        —                   ;argument block are ignored
                            ;for nondirectory devices.
                            ;Refer to Paragraph 6.1.5.3
                            ;for the DECtape RENAME
                            ;and Paragraph 6.8.2.1 for
                            ;the disk RENAME.
```

If no device has been associated with channel D, the monitor stops the job, prints

I/O TO UNASSIGNED CHANNEL AT USER LOC addr

and returns the user's terminal to monitor mode.

The normal return is given if:

a. The device specified is a nondirectory device.

b. If the filename specified in location E is 0, the file is deleted after all read references are completed.

c. If the filename specified in location E and the filename extension specified in the left half of location E+1 are the same as the current filename and filename extension, the access protection bits are set to the contents of bits 0 to 8 of location E+2.

d. If the filename/filename extension specified differ from the current filename/filename extension, a search is made for the specified filename and filename extension. If a match is not found (1) the filename is changed to the filename in location E, (2) the filename extension is changed to the filename extension in the left half of location E+1, (3) the access protection bits are changed to the contents of bits 0-8 of location E+2, and (4) the access date is unchanged.

The error return is given if:

a. No file is selected on channel D.

b. The specified file is not found.

c. The file is found but is being written, superseded, or renamed.

d. The file is found but the user does not have the privileges to RENAME the file.

e. The filename/filename extension specified differ from the current filename/filename extension, a search is made for the specified filename and filename extension. If a match is found, the error return is taken.

f. The UFD is deleted.

Refer to Appendix E for the error codes returned in bits 18-35 of location E+1. Refer to Paragraph 6.1.5.3 for details of a DECtape RENAME and Paragraph 6.8.2.1 for details of a disk RENAME.

Examples

General Device Initialization

```
INIDEV:  0                        ;JSR HERE
         OPEN 3,OPNBLK            ;CHANNEL 3
           JRST NOTAVL            ;WHERE TO GO IF DTA5 IS BUSY

;FROM HERE DOWN IS OPTIONAL DEPENDING ON THE DEVICE AND PROGRAM
;REQUIREMENTS
         MOVE 0, JOBFF
         MOVEM 0, SVJBFF          ;SAVE THE FIRST ADDRESS OF THE BUFFER
                                  ;RING IN CASE THE SPACE MUST BE
                                  ;RECLAIMED
         INBUF 3,4                ;SET UP 4 INPUT BUFFERS
         OUTBUF 3,1               ;SET UP 1 OUTPUT BUFFER
         LOOKUP 3, INNAM          ;INITIALIZE AN INPUT FILE
           JRST NOTFND            ;WHERE TO GO IF THE INPUT FILENAME IS
                                  ;NOT IN THE DIRECTORY
         ENTER 3, OUTNAME         ;INITIALIZE AN OUTPUT FILE
           JRST NOROOM            ;WHERE TO GO IF THERE IS NO ROOM IN
                                  ;THE DIRECTORY FOR A NEW FILENAME
         JRST @INIDEV             ;RETURN TO MAIN SEQUENCE
OPNBLK:  14                       ;BINARY MODE
         SIXBIT/DTA5/             ;DEVICE DECTAPE UNIT 5
         XWD OBUF,IBUF            ;BOTH INPUT AND OUTPUT
OBUF:    BLOCK 3                  ;SPACE FOR OUTPUT BUFFER HEADER
IBUF:    BLOCK 3                  ;SPACE FOR INPUT BUFFER HEADER
```

(continued on next page)

```
INNAM:    SIXBIT/NAME/              ;FILE NAME
          SIXBIT/EXT/               ;FILE NAME EXTENSION (OPTIONALLY 0),
                                    ;RIGHT HALF WORD RECEIVES THE
                                    ;FIRST BLOCK NUMBER
          0                         ;RECEIVES THE DATE
          0                         ;UNUSED FOR NONDUMP I/O
OUTNAM:   SIXBIT/NAME/              ;SAME INFORMATION AS IN INNAM
          SIXBIT/EXT/
          0
          0
```

4.5 DATA TRANSMISSION

The programmed operators

```
INPUT D,E and IN D,E
          normal return
          error return
```

transmit data from the file selected on channel D to the user's core area. The programmed operators

```
OUTPUT D,E and OUT D,E
          normal return
          error return
```

transmit data from the user's core area to the file selected on channel D. If specified, E is the effective address of the next buffer to be written. If E is not specified, the next buffer in the sequence is implied.

If no OPEN or INIT operator has been performed on channel D, the monitor stops the job and prints

I/O TO UNASSIGNED CHANNEL AT USER addr

(addr is the location of the IN, INPUT, OUT, or OUTPUT programmed operator) on the user's terminal and the terminal is left in monitor mode. If the device is a multiple-directory device and no file is selected on channel D, bit 18 of the file status is set to 1, and control returns to the user's program. Control always returns to the location immediately following an INPUT (operation code 066) and an OUTPUT (operation code 067). A check of the file status for end-of-file and error conditions must then be made by another programmed operator. Note that to trap on a hardware write-locked device, the user should use location .JBINT (refer to Paragraph 3.1.3.2). Following an INPUT, the user program should check the word count of the next buffer to determine if it contains data. Control returns to the location immediately following an IN (operation code 056) if no end-of-file or error condition exists (i.e., if bits 18 through 22 of the file status are all 0). Control returns to the location immediately following an OUT (operation code 057) if no error condition or end-of-tape exists (i.e., if bits 18 through 21 and bit 25 are all zero). Otherwise, control returns to the second location following the IN or OUT. Note that IN and OUT UUOs are the only ones in which the error return is a skip and the normal return is not a skip.

4.5.1 Unbuffered Data Modes

Data modes 15, 16, and 17 utilize a command list to specify areas in the user's allocated core to be read or written. The effective address E of the IN, INPUT, OUT, and OUTPUT programmed operators point to the first word of the command list. Three types of entries may occur in the command list.

a. IOWD n, loc - Causes n words from loc through loc+n-1 to be transmitted. The next command is obtained from the next location following the IOWD. The assembler pseudo-op IOWD generates XWD -n, loc-1.

b. XWD 0, y - Causes the next command to be taken from location y. Referred to as a GOTO word. Up to three consecutive GOTO words are allowed in the command list. After three consecutive GOTO words, an I/O instruction must be written.

c. 0 - Terminates the command list.

Each IOWD which causes data to be transferred writes a separate record. Thus, for devices other than DECtape, the following two examples produce the same result.

```
1)  OUTPUT  D, [IOWD 100, BUF1
                IOWD 100, BUF2
                Z]
2)  OUTPUT  D, [IOWD 100, BUF1
                Z]
    OUTPUT  D, [IOWD 100, BUF2
                Z]
```

For DECtape (where space is an important consideration), the first example writes one block, and the second writes two.

The monitor does not return program control to the user until the command list has been completely processed. If an illegal address is encountered while processing the list, the job is stopped and the monitor prints

ADDRESS CHECK AT USER addr

on the user's terminal and the terminal is left in monitor mode.

Example: Dump Output

Dump input is similar to dump output. This routine outputs fixed-length records.

```
DMPINI:  0                        ;JSR HERE TO INITIALIZE A FILE
         OPEN 0,0PNBLK            ;CHANNEL 0
           JRST NOTAVL            ;WHERE TO GO IF MTA2 IS BUSY
         JRST @ DMPINI            ;RETURN

DMPOUT:  0                        ;JSR HERE TO OUTPUT THE OUTPUT AREA
         OUTPUT 0,OUTLST          ;SPECIFIES DUMP OUTPUT ACCORDING
                                  ;TO THE LIST AT OUTLIST
         STATZ 0, 740000          ;CHECK ERROR BITS
           CALL[SIXBIT /EXIT/]    ;QUIT IF AN ERROR OCCURS
         JRST @DMPOUT             ;RETURN
```

```
DMPDON:  0                              ;JSR HERE TO WRITE AN END OF FILE
         CLOSE 0,                       ;WRITE THE END OF FILE
         STATZ 0, 740000                ;CHECK FOR ERROR DURING WRITE
                                        ;END OF FILE OPERATION
           CALL[SIXBIT /EXIT/]          ;QUIT IF ERROR OCCURS
         RELEAS 0,                      ;RELINQUISH THE DEVICE
         JRST @DMPDON                   ;RETURN
OPNBLK:  16                             ;DUMP MODE
         SIXBIT /MTA2/                  ;MAGNETIC TAPE UNIT 2
         0                              ;NO RING BUFFERS
OUTLST:  IOWD BUFSIZ,BUFFER             ;SPECIFIES DUMPING A NUMBER OF
                                        ;WORDS EQUAL TO BUFSIZ, STARTING
         0                              ;AT LOCATION BUFFER
                                        ;SPECIFIES THE END OF THE COMMAND
                                        ;LIST
BUFFER:  BLOCK BUFSIZ                   ;OUTPUT BUFFER, MUST BE CLEARED
                                        ;AND FILLED BY THE MAIN PROGRAM
```

4.5.2 Buffered Data Modes

In data modes 0, 1, 10, 13, and 14 the effective address E of the INPUT, IN, OUTPUT and OUT programmed operators may be used to alter the normal sequence of buffer reference. If E is 0, the address of the next buffer is obtained from the right half of the second word of the current buffer. If E is non-zero, it is the address of the second word of the next buffer to be referenced. The buffer pointed to by E can be in an entirely separate ring from the present buffer. Once a new buffer location is established, the following buffers are taken from the ring started at E. Since buffer rings are not changed if I/O activity is pending, it is not necessary to issue a WAIT UUO.

4.5.2.1 Input - If no input buffer ring is established when the first INPUT or IN is executed, a 2-buffer ring is set up (refer to Paragraph 4.3.2).

Buffered input may be performed synchronously or asynchronously at the option of the user. If bit 30 of the file status is 1, each INPUT and IN programmed operator performs the following:

(1) Clears the use bit in the second word of the buffer with an address in the right half of the first word of the buffer header, thereby making the buffer available for refilling by the monitor.

(2) Advances to the next buffer by moving the contents of the second word of the current buffer to the right half of the first word of the 3-word buffer header.

(3) Returns control to the user's program if an end-of-file or error condition exists. Otherwise, the monitor starts the device, which fills the buffer and stops transmission.

(4) Computes the number of bytes in the buffer from the number of words in the buffer (right half of the first data word of the buffer) and the byte size, and stores the result in the third word of the buffer header.

(5) Sets the position and address fields of the byte pointer in the second word of the buffer header, so that the first data byte is obtained by an ILDB instruction.

(6) Returns control to the user's program.

Thus, in synchronous mode, the position of a device (e.g., magnetic tape), relative to the current data, is easily determined. The asynchronous input mode differs in that once a device is started, successive buffers in the ring are filled at the interrupt level without stopping transmission until a buffer

whose bit is 1 is encountered. Control returns to the user's program after the first buffer is filled. The position of the device, relative to the data currently being processed by the user's program, depends on the number of buffers in the ring and when the device was last stopped.

Example: General Subroutine to Input One Character

```
;GET -- ROUTINE TO GET ONE BYTE FROM THE INPUT FILE
;       NULLS (0) WILL BE DISCARDED
;CALL:  JSP     A,GET
;         END-OF-FILE RETURN
;       RETURN WITH BYTE IN C

GET:    SOSGE   IB+2                ;DECREMENT THE BYTE COUNT
        JRST    GETBF               ;BUFFER EMPTY--GET ANOTHER ONE
        ILDB    C,IB+1              ;SOMETHING THERE--GET IT
        JUMPN   C,1(A)              ;RETURN IF NOT NULL
                                    ;** IF NULLS ARE SIGNIFICANT, THIS
                                    ;   SHOULD BE A JRST  1(A)
        JRST    GET                 ;NULL--LOOP FOR ANOTHER CHARACTER

;HERE WHEN INPUT BUFFER IS EMPTY
;ASK THE MONITOR FOR THE NEXT BUFFER AND JUMP BACK
;RETURN TO USER IF END-OF-FILE

GETBF:  IN      I,                  ;GET BUFFER FROM MONITOR
          JRST  GET                 ;NO ERRORS OR EOF--JUMP BACK
        GETSTS  I,C                 ;GET ERROR STATUS
        TRNN    C,74B23             ;SEE IF ANY ERRORS
        JRST    GETBFE              ;NO--GO CHECK EOF
                          ;** INSERT ERROR ROUTINE HERE
                          ;   FOR EXAMPLE, TYPE C IN OCTAL
                          ;   WITH MESSAGE GIVING FILE NAME, ETC.
        TRZ     C,74B23             ;CLEAR ERROR BITS
        SETSTS  I,(C)               ;TELL MONITOR

GETBFE: TRNE    C,1B22              ;SEE IF END OF FILE
        JRST    (A)                 ;YES--GIVE NON-SKIP RETURN
        JRST    GET                 ;NO--JUMP BACK TO PROCESS DATA
```

4.5.2.2 Output - If no output buffer ring has been established (i.e., if the first word of the buffer header is 0), when the first OUT or OUTPUT is executed, a 2-buffer ring is set up (refer to Paragraph 4.3.2). If the ring use bit (bit 0 of the first word of the buffer header) is 1, it is set to 0, the current buffer is cleared to all 0s, and the position and address fields of the buffer byte pointer (the second word of the buffer header) are set so that the first byte is properly stored by an IDPB instruction. The byte count (the third word of the buffer header) is set to the maximum of bytes that may be stored in the buffer, and control is returned to the user's program. Thus, the first OUT or OUTPUT initializes the buffer header and the first buffer, but does not result in data transmission.

If the ring use bit is 0 and the bit 31 of the file status is 0, the number of words in the buffer is computed from the address field of the buffer byte pointer (the second word of the buffer header) and the buffer pointer (the first word of the buffer header), and the result is stored in the right half of the third

word of the buffer. If bit 31 of the file status is 1, it is assumed that the user has already set the word count in the right half of the third word. The buffer use bit (bit 0 of the second word of the buffer) is set to 1, indicating that the buffer contains data to be transmitted to the device. If the device is not currently active (i.e., not receiving data), it is started. The buffer header is advanced to the next buffer by setting the buffer pointer in the first word of the buffer header. If the buffer use bit of the new buffer is 1, the job is put into a wait state until the buffer is emptied at the interrupt level. The buffer is then cleared to 0s, the buffer byte pointer and byte count are initialized in the buffer header, and control is returned to the user's program.

Example: General Subroutine to Output One Character

```
;PUT -- ROUTINE TO PUT ONE BYTE INTO THE OUTPUT FILE
;CALL:  MOVE    C,BYTE
;       JSP     A,PUT
;       RETURN

PUT:    SOSG    OB+2            ;ADVANCE BYTE COUNTER
        JRST    PUTBF           ;JUMP IF BUFFER FULL (OR FIRST CALL)
PUTC:   IDPB    C,OB+1          ;PUT BYTE INTO BUFFER
        JRST    (A)             ;RETURN TO CALLER

;JUMP HERE WHEN BUFFER IS FULL AND THE NEXT ONE IS NEEDED
;GIVE THE MONITOR THE BUFFER AND JUMP BACK

PUTBF:  OUT     0,              ;GIVE BUFFER TO MONITOR
          JRST  PUTC            ;NO ERRORS--JUMP BACK
        MOVEM   C,SAVEC#        ;ERROR--SAVE AC FOR STATUS CHECKING
        GETSTS  0,C             ;GET ERROR STATUS
                            ;** INSERT OUTPUT ERROR ROUTINE HERE
                            ;   FOR EXAMPLE, TYPE C IN OCTAL
                            ;   WITH MESSAGE GIVING FILE NAME, ETC.
        TRZ     C,74B23         ;CLEAR ERROR BITS
        SETSTS  0,(C)           ;TELL MONITOR
        MOVE    C,SAVEC         ;RESTORE CHARACTER
        JRST    PUTC            ;JUMP BACK TO PROCESS CHARACTER
```

4.5.3 Synchronization of Buffered I/O

In some instances, such as recovery from transmission errors, it is desirable to delay until a device completes its I/O activities. The programmed operator

WAIT D, or CALLI D, 10

returns control to the user's program when all data transfers on channel D have finished. This UUO does not wait for a magnetic tape spacing operation, since no data transfer is in progress. An MTAPE D, 0 (refer to Paragraph 5.5.3) should be used to wait for the magnetic tape controller to be freed after completing spacing and I/O activity on magnetic tape. In addition, the UUO does not wait for physical I/O to the terminal to be completed; it waits only until the user's buffer is empty. Therefore, the usual motive for the WAIT UUO, error recovery, does not apply to the terminal. If no device is associated with data channel D, control returns immediately. After the device is stopped, the position of the device relative to the data currently being processed by the user's program can be determined by the buffer use bits.

4.6 STATUS CHECKING AND SETTING

The file status is a set of 18 bits (right-half word), which reflects the current state of a file transmission. The initial status is a parameter of the INIT and OPEN operators. Thereafter, bits are set by the monitor, and may be tested and reset by the user via the STATZ, STATO, and SETSTS monitor programmed operators. Table 4-3 defines the file status bits. All bits, except the end-of-file bit, are set immediately by the monitor as the conditions occur, rather than being associated with the buffer currently being used. However, the file status is stored with each buffer so that the user can determine which bufferful produced an error. The end-of-file bit is set when the user attempts to read past the last block of data (i.e., it is set on an IN or INPUT UUO for which there is no corresponding data;

Table 4-3
File Status Bits

Bit	Meaning
18	Improper mode (IO.IMP). Attempt to write on a software write-locked tape or file structure, or a software detected redundancy failure occurred. Usually set by monitor.
19	Hard device detected error (IO.DER), other than data parity error. This is a search, power supply, or channel memory parity error. The device is in error rather than the data on the medium. However, the data read into core or written on the device is probably incorrect. Usually set by monitor.
20	Hard data error (IO.DTE). The data read or written has incorrect parity as detected by hardware (or by software on CDR, PTR). The user's data is probably non-recoverable even after the device is fixed. Usually set by monitor.
21	Block too large (IO.BKT). A block of data from a device is too large to fit in a buffer; a block number is too large for the unit; the file structure (DSK) or unit (DTA) has filled; or the user's quota on the file structure has been exceeded. Usually set by monitor.
22	End of file (IO.EOF). The user program has requested data beyond the last record or block with an IN or INPUT UUO, or USETI has specified a block beyond the last data block of the file. When set, no data has been read into the input buffer. Usually set by monitor.
23	I/O active (IO.ACT). The device is actively transmitting or receiving data. Always set by monitor.
24-29	Device dependent parameters. Refer to Chapters 5 and 6 and Appendix D for detailed information about each device. Usually set by user.
30	Synchronous input (IO.SYN). Stops the device after each buffer is filled. Usually set by user.
31	User word count (IO.UWC). Forces the monitor to use the word count in the third word of the buffer (output only). The monitor normally computes the word count from the byte pointer in the buffer header. Usually set by user.
32-35	Data mode (IO.MOD). Refer to Tables 4-1 and 4-2. Usually set by user.

the previous IN or INPUT UUO obtained the end of the data). Therefore, when this bit is set, no data has been placed in the input buffer.

The programmed operators discussed in this section are the software equivalents of the hardwired instructions CONO, CONI, CONSO, and CONSZ. A more thorough description of bits 18 through 29 for each device is given in Chapters 5 and 6 and in Appendix D.

4.6.1 File Status Checking

The file status (refer to Table 4-3) is retrieved by the GETSTS (operation code 062) and tested by the STATZ (operation code 063) and STATO (operation code 061) programmed operators. In each case, the accumulator field of the instruction selects a data channel. If no device is associated with the specified data channel, the monitor stops the job and prints

I/O TO UNASSIGNED CHANNEL AT USER addr

(addr is the location of the GETSTS, STATZ, or STATO programmed operator) on the user's terminal and the terminal is left in monitor mode.

GETSTS D,E stores the file status of data channel D in the right half and 0 in the left half of location E.

STATZ D,E skips, if all file status bits selected by the effective address E are 0.

STATO D,E skips, if any file status bit selected by the effective address E is 1.

4.6.2 File Status Setting

The initial file status is a parameter of the INIT and OPEN programmed operators; however, the file status may be changed by the SETSTS (operation code 060) programmed operator. Error status bits IO.ERR (IO.IMP, IO.DER, IO.DTE, and IO.BKT) must be cleared by this programmed operator if the user is attempting an error recovery. In addition, the SETSTS UUO can be used to clear the end-of-file bit, but this is not sufficient to clear the end-of-file condition. Further inputs will not occur until the end-of-file condition (determined by an internal monitor flag IOEND) is cleared by a CLOSE or INIT UUO.

SETSTS D,E waits until the device on channel D stops transmitting data and replaces the current file status, except bit 23, with the effective address E. If the new data mode, indicated in the right four bits of E, is not legal for the device, the job is stopped and the monitor prints

ILL DEVICE DATA MODE FOR DEVICE dev AT USER addr

(dev is the physical name of the device and addr is the location of the SETSTS operator) on the user's terminal and the terminal is left in monitor mode. If the user program changes the data mode, it must

also change the byte size for the byte pointer in the input buffer header (if any) and the byte size and item count in the output buffer header (if any). The output item count should be changed by using the count already placed there by the monitor and dividing or multiplying by the appropriate conversion factor, rather than assuming the length of a buffer. Incorrect I/O may result if a data mode change requires a different buffer length. SETSTS does not change buffer lengths. The mode specified in the INIT is used to determine buffer sizes even though the buffer ring has not been created.

4.7 FILE TERMINATION

File transmission is terminated by the CLOSE D,N (operation code 070) programmed operator. N is usually zero, but individual options may be selected independently to control the effect of the CLOSE.

Usually a given channel is OPEN for file transmission in only one direction, and CLOSE has the effect of either closing input if INPUTs have been done or closing output if OUTPUTs have been done. However, disk and DECtape may have a single channel OPEN for both INPUT and OUTPUT, in which case the first two options below are useful.

4.7.1 CLOSE D,0

The output side of channel D is closed (bit 35=0). In unbuffered data modes, the effect is to execute a device dependent function. In buffered data modes, if a buffer ring exists, the following operations are performed:

a. All data in the buffers that has not been transmitted to the device is written.

b. Device dependent functions are performed.

c. The ring use bit (bit 0 of the first word of the buffer header) is set to 1 indicating that the buffer ring is available.

d. The buffer byte count (the third word of the buffer header) is set to 0.

e. Control returns to the user program when transmission is complete.

The input side of channel D is also closed (bit 34=0). The end-of-file flag is always cleared. Further action depends on the data mode in unbuffered data modes, the effect is to execute a device dependent function. In buffered data modes, if a ring buffer exists, the following operations are performed:

a. Wait until device is inactive.

b. The use bit of each buffer (bit 0 of the second word) is cleared indicating that the buffer is empty.

c. The ring use bit of the buffer header (bit 0 of the first word of the buffer header) is set to 1 indicating that the buffer ring is available.

d. The buffer byte count (the third word of the buffer header) is set to 0.

e. Control returns to the user program.

On output CLOSE, the unwritten blocks at the end of a disk file are automatically deallocated (bit 33=0). On input CLOSE, the access date of a disk file is updated (bit 32=0).

4.7.2 CLOSE D,1 (Bit 35=1, CL.OUT)

The closing of the output side of channel D is suppressed. Other actions of CLOSE are unaffected.

4.7.3 CLOSE D,2 (Bit 34=1, CL.IN)

The closing of the input side of channel D is inhibited; other actions of CLOSE are unaffected.

4.7.4 CLOSE D,4 (Bit 33=1, CL.DLL)†

The unwritten blocks at the end of a disk file are not deallocated. This capability is provided for users who specifically allocate disk space and wish to retain it.

4.7.5 CLOSE D,10 (Bit 32=1, CL.ACS)†

The updating of the access date on CLOSE input is inhibited. This capability is intended for use with FAILSAFE, so that files can be saved on magnetic tape without causing the disk copy to appear as if it has been accessed.

4.7.6 CLOSE D,20 (Bit 31=1, CL.NMB)†

The deleting of the NAME block and the access tables in monitor core on CLOSE input is inhibited if a LOOKUP was done without subsequent INPUT. This bit is used by the COMPIL program to retain the core block in order to speed up the subsequent access by the system program called by COMPIL.

4.7.7 CLOSE D,40 (Bit 30=1, CL.RST)†

The deleting of the original file, if any, is inhibited if an ENTER which creates or supersedes was done. The new copy of the file is discarded. This bit is used by the queue manager (QMANGR) to create a file or a unique name and not supersede the original file.

4.7.8 CLOSE D,100 (Bit 29=1,CL.DAT)†

The NAME block and access tables are deleted from the disk data base and the space is returned to free core.

---

† Meaningful with disk files only, ignored with non-disk files.

Any combinations of the above bit settings are legal.

Example: Terminating a File

```
DROPDV:  0                      ;JSR HERE
         CLOSE 3,               ;WRITE END OF FILE AND TERMINATE
                                ;INPUT
         STATZ 3, 740000        ;RECHECK FINAL ERROR BITS
           JRST OUTERR          ;ERROR DURING CLOSE
         RELEAS 3,              ;RELINQUISH THE USE OF THE
                                ;DEVICE, WRITE OUT THE DIRECTORY
         MOVE 0, SVJBFF
         MOVEM 0, JOBFF         ;RECLAIM THE BUFFER SPACE
         JRST @ DROPDV          ;RETURN TO MAIN SEQUENCE
```

4.8 DEVICE TERMINATION AND REASSIGNMENT

4.8.1 RELEASE

When all transmission between the user's program and a device is finished, the program must relinquish the device by performing a

RELEASE D,

RELEASE (operation code 071) returns control immediately, if no device is associated with data channel D. Otherwise, both input and output sides of data channel D are CLOSEd and the correspondence between channel D and the device, which was established by the INIT or OPEN programmed operators, is terminated. Any errors that occurred are recorded in the BAT block if a super USETI/USETO was used with channel D. If the device is neither associated with another data channel nor assigned by the ASSIGN or MOUNT monitor command, it is returned to the monitor's pool of available facilities. Control is returned to the user's program.

4.8.2 RESDV. AC, or CALLI AC, 117

This UUO allows a user program to reset a single channel. It is similar to the RELEASE UUO except any files and buffers are not closed. Files that are open on the channel are deleted; any older version with the same filename remains. All I/O transmissions on the channel are stopped, and device allocations made by the INIT or OPEN UUOs on the specified channel are cleared. The device is returned to the monitor pool unless it has been assigned by the ASSIGN or MOUNT monitor command. The call is:

```
MOVEI AC, channel number
RESDV. AC,                  ; or CALLI AC, 117
error return
normal return
```

On an error return, either the AC is unchanged if the UUO is not implemented, or AC contains -1 if there is no device associated with the channel.

On a normal return, the channel is reset.

4.8.3 REASSIGN AC, or CALLI AC, 21

This UUO reassigns a device under program control to the specified job and clears the directory currently in core, but does not clear the logical name assignment. A device can be reassigned if it is assigned to the current job, or if it is both not assigned to any job and is not detached. A RELEASE UUO is performed unless the job issuing the UUO is reassigning the device to itself by specifying -1 in AC or is deassigning the device by specifying 0 in AC. If the device is restricted when it is deassigned with a 0 in AC, it is returned to the restricted pool of devices and can be reassigned to a non-privileged job by a privileged job. (This is the method by which the MOUNT command is implemented.)

The call is:

```
MOVE AC, job number
MOVE AC+1, [SIXBIT /DEVICE/]        ;or MOVEI AC+1, channel number
REASSIGN AC,                        ;or CALLI AC, 21
return                              ;error and normal
```

If on return the contents of AC = 0, the specified job has not been initialized. If the contents of AC+1=0, the device has not been assigned to the new job, the device is the job's controlling terminal, the logical name is duplicated, or the logical name is a physical name in the system and the job reassigning the device is either logged in under a different project-programmer number or is not the operator.

4.8.4 DEVLNM AC, or CALLI AC, 107[1]

This UUO sets the logical name for the specified device. Upon call of the UUO, AC contains either the device name or the channel number associated with the device. The call is:

```
MOVE AC, [SIXBIT /dev/]             ;or MOVEI AC, channel no.
MOVE AC+1, [SIXBIT /log.name/]
DEVLNM AC,                          ;or CALLI AC, 107
error return
normal return
```

On an error return, AC contains one of the following:

AC = unchanged if the UUO is not implemented.
AC = -1 if a non-existent device or channel number was specified.
AC = -2 if the logical name is already in use.
AC = -3 if device is neither assigned by a console command (ASSIGN, MOUNT) nor by the program (INIT, OPEN).

On a normal return, AC and AC+1 are unchanged.

[1]This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

4.9 EXAMPLES

4.9.1 File Reading

The following UUO sequence is required to read a file:

OPEN	Establishes a file structure-channel correspondence (or a set of file structure channel correspondences).
LOOKUP	Establishes a file-channel correspondence. Invokes a search of the UFD. Returns information from the file system.
INBUF	(Optional) Sets up 1 to N ring buffers in the top of core, expand core if necessary.
INPUT	Sets up 2-buffer ring if no INBUF was done.
•	
•	
•	
INPUT	Requests buffers of data from the monitor.
CLOSE	Breaks file-channel correspondence.
RELEASE	Breaks device-channel correspondence.

4.9.2 File Writing

The following UUO sequence is required to write a file:

OPEN	Forms file structure-channel correspondence (or a set of file structure channel correspondences).
ENTER	Forms file-channel correspondence. The monitor creates some temporary storage for interlocking and shared access purpose for the filename. No directory entry is made.
OUTPUT	
•	
•	
•	
OUTPUT	Passes buffers of data to monitor for transmission to storage device. Should not be used for the final buffer because CLOSE completes the action of ENTER.
CLOSE	Completes the action of ENTER. Adds filename to file system. Normally returns allocated, but unused, blocks to the file system.
RELEASE	Breaks device-channel correspondence.

4.9.3 File Reading/Writing

```
        TITLE    FILTRN -- SAMPLE I/O PROGRAM

;A PROGRAM THAT READS 7-BIT ASCII CHARS FROM FILE INFILE.DAT
;ON DEVICE DATA AND OUTPUTS THEM TO FILE OUTFIL.LST ON DEVICE LIST
;NOTE THAT DEVICES DATA AND LIST ARE LOGICAL NAMES.  THUS
;THE PHYSICAL NAMES ARE DETERMINED AT RUN TIME TO PROVIDE DEVICE
;INDEPENDENCE.
;BOTH INPUT AND OUTPUT FILES ARE ACCESSED SEQUENTIALLY.

START:  RESET                       ;DEVICE RESET (IN CASE PROGRAM
                                    ;  IS RESTARTED)
        OPEN    1,[     1           ;CONNECT DEVICE DATA TO PROG ON CH 1
                SIXBIT /DATA/
                XWD 0,IBUF1]        ;INBUF1 IS THE INPUT BUFFER HEADER
          HALT  .                   ;ERROR RETURN
        OPEN    2,[     1           ;CONNECT DEVICE LIST TO CH 2
                SIXBIT /LIST/
                XWD OBUF2,0]        ;OBUF2 IS OUTPUT BUFFER HEADER
          HALT  .
        LOOKUP  1,L1                ;OPEN FILE INFILE.DAT FOR INPUT
          HALT  .                   ;ERROR RETURN
        ENTER   2,E2                ;OPEN FILE OUTFIL.LST FOR OUTPUT
          HALT  .
        INBUF   1,3                 ;CREATE 3 INPUT BUFFERS
                                    ;SINCE NO BUFFERS SPECIFIED FOR OUTPUT
                                    ;  ON FIRST OUTPUT THE MONITOR WILL
                                    ;  MAKE 2

;THIS IS THE BASIC I/O LOOP FOR THE JOB

NEWCHR:  JSR     GET                ;GO GET ONE INPUT CHARACTER
         JSR     PUT                ;GO PUT THE CHARACTER RECEIVED
         JRST    NEWCHR             ;LOOP FOR NEXT ONE

;GET -- ROUTINE TO GET ONE CHARACTER FROM THE INPUT
;IT ENDS THE PROGRAM AT INPUT END-OF-FILE

GET:     Z                          ;ENTRY/EXIT
GET1:    SOSGE   IBUF1+2            ;IS INPUT BUFFER EMPTY?
         JRST    GETBF              ;YES--INPUT FROM DEVICE
         ILDB    3,IBUF1+1          ;GET A CHARACTER FROM INPUT BUFFER
         JUMPE   3,GET1             ;IF NULL, THROW IT AWAY AND GET NEXT
                                    ;  CHARACTER.  THIS IS CONVENTIONAL FOR
                                    ;  ASCII DATA.
         JRST    @GET               ;RETURN WITH CHARACTER IN AC 3

GETBF:   IN      1,                 ;DO INPUT FROM DEVICE
           JRST  GET1               ;LOOP IF NO ERRORS AND NOT EOF
         STATZ   1,74B23            ;SEE IF ERROR READING
           HALT  .                  ;YES--GIVE UP

FINISH:  CLOSE   1,                 ;EOF--CLOSE INPUT
         CLOSE   2,                 ;CLOSE OUTPUT
         RELEAS  1,                 ;RELEASE DEVICE DATA
         RELEAS  2,                 ;RELEASE DEVICE LIST
         EXIT                       ;EXIT TO MONITOR
```

(continued on next page)

```
;PUT--ROUTINE TO PUT ONE CHARACTER ONTO THE OUTPUT

PUT:    Z                       ;ENTRY/EXIT
        SOSG    OBUF2+2         ;IS OUTPUT BUFFER FULL?
        JRST    PUTBF           ;YES--GO OUTPUT IT
PUTC:   IDPB    3,OBUF2+1       ;PUT CHARACTER IN BUFFER
        JRST    @PUT            ;RETURN

PUTBF:  OUT     2,              ;OUTPUT BUFFER TO DEVICE
          JRST PUTC             ;OK, NOW STORE CHARACTER IN BUFFER
        HALT                    ;GIVE UP IF OUTPUT ERROR

;DATA STORAGE AREA

L1:     SIXBIT /INFILE/         ;INPUT FILE NAME
        SIXBIT /DAT/            ;INPUT EXTENSION
        Z                       ;PROTECTION AND CREATION DATE RETURNED
        Z                       ;INPUT DIRECTORY.  0 MEANS MY OWN

E2:     SIXBIT /OUTFIL/         ;OUTPUT FILE NAME
        SIXBIT /LST/            ;OUTPUT EXTENSION
        Z                       ;PROTECTION CAN GO HERE.  0 MEANS STD.
        Z                       ;OUTPUT DIRECTORY.  0 MEANS MY OWN

IBUF1:  BLOCK   3               ;INPUT BUFFER HEADER
OBUF2:  BLOCK   3               ;OUTPUT BUFFER HEADER

        END     START
```

4.10 DEVICE INFORMATION

4.10.1 DEVSTS AC, or CALLI AC, 54[1]

This UUO retrieves the DEVSTS word of the device data block for an INITed device. The DEVSTS word is used by a device service routine to save the results of a CONI after each interrupt from the device. Refer to Appendix D for the device status bits. Devices that use the DEVSTS UUO are the following: CDR, CDP, MTA, DTA, PTR, PTP, DSK, LPT, and PLT.

The call is:

```
MOVEI AC, channel number of device    ;or MOVE AC, [SIXBIT /dev/]
DEVSTS AC,                            ;or CALLI AC, 54
error return                          ;UUO not implemented for any devices
normal return                         ;AC contains the DEVSTS
                                      ;word of the DDB.
```

On return, the contents of the DEVSTS word is returned in AC. Therefore, if the device service routine does not store a CONI, useless information may be returned to user. Note that an error return is not indicated if the device service routine does not use the DEVSTS word for its intended purpose. Devices with both a control and data interrupt store the controller CONI (MTS, DTS, DSK, DSK2, DPC, DPC2).

[1] This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

The DEVSTS UUO is not meaningful when used in asynchronous buffered I/O mode unless a WAIT UUO (see Paragraph 4.5.3) is issued first to ensure synchronization of the actual data transferred with the device status returned.

4.10.2 DEVCHR AC, or CALLI AC, 4

This UUO allows the user to determine the physical characteristics associated with a device name. When the UUO is called, AC must contain either the logical or physical device name as a left-justified SIXBIT quantity, or the channel number of the device as a right-justified quantity.

The call is:

```
MOVE AC, [SIXBIT/DEV/]          ;or MOVEI AC, channel number of device
DEVCHR AC,                      ;or CALLI AC,4
return
```

If the device is not found or the channel is not INITed, the contents of AC is zero on return. If the device is found, the following information is returned in AC.

Name	Bit	Explanation
DV.DRI	Bit 0 = 1	DECtape directory is in core. This bit is cleared by an ASSIGN or DEASSIGN to that unit.
DV.DSK	Bit 1 = 1	Device is a disk.
DV.CDR	Bit 2 = 1	Device is a card reader (DV.IN = 1) or card punch (DV.OUT = 1).
DV.LPT	Bit 3 = 1	Device is a line printer.
DV.TTA	Bit 4 = 1	TTY is controlling a job.
DV.TTU	Bit 5 = 1	TTY is in use as a user terminal (even if detached).
DV.TTB	Bit 6 = 1	Free bit left from SCNSRF.
DV.DIS	Bit 7 = 1	Device is a display.
DV.LNG	Bit 8 = 1	Device has a long dispatch table (that is, UUOs other than INPUT, OUTPUT, CLOSE, and RELEASE perform real actions).
DV.PTP	Bit 9 = 1	Device is a paper-tape punch.
DV.PTR	Bit 10 = 1	Device is a paper-tape reader.
DV.DTA	Bit 11 = 1	Device is a DECtape.
DV.AVL	Bit 12 = 1	Device is available to this job or is already assigned to this job.
DV.MTA	Bit 13 = 1	Device is a magnetic tape.
DV.TTY	Bit 14 = 1	Device is a TTY.
DV.DIR	Bit 15 = 1	Device has a directory (DTA or DSK).

(continued on next page)

Name	Bit	Explanation
DV.IN	Bit 16 = 1	Device can perform input.
DV.OUT	Bit 17 = 1	Device can perform output.
DV.ASC	Bit 18 = 1	Device is assigned by a console command.
DV.ASP	Bit 19 = 1	Device is assigned by program (INIT or OPEN).
	Remaining bits	If bit 35-n contains a 1, then mode n is legal for that device. The mode number (0 through 17) must be converted to decimal (e.g., mode 17_8 is represented by bit $35-15_{10}$ or bit 20).

4.10.3 DEVTYP AC, or CALLI AC, 53

The device-type UUO is used to determine properties of devices. This UUO accepts, as an argument, a device name in SIXBIT or a right-justified channel number. The call is:

```
MOVE AC, [SIXBIT/dev/]        ;or MOVEI AC, channel no.
DEVTYP AC,                    ;or CALLI AC, 53
error return
normal return
```

The error return is given if the UUO is not implemented. In this case, the DEVCHR UUO should be used. On a normal return, if AC=0, the specified device does not exist or the channel is not INITed. If the device exists, the following information is returned in AC.

Name	Bit	Explanation
TY.MAN	Bit 0 = 1	LOOKUP/ENTER mandatory.
	Bits 1-11	Reserved for the future.
TY.AVL	Bit 12 = 1	Device is available to this job.
TY.SPL	Bit 13 = 1	Spooled on disk. (Other bits reflect properties of real device, except variable buffer size.)
TY.INT	Bit 14 = 1	Interactive device (output after each break character).
TY.VAR	Bit 15 = 1	Capable of variable buffer size (user can set his own buffer lengths).
TY.IN	Bit 16 = 1	Capable of input.
TY.OUT	Bit 17 = 1	Capable of output.
TY.JOB	Bits 18-26	Job number that currently has device INITed or ASSIGNed.

(continued on next page)

Name	Bit	Explanation
	Bits 27-28	Reserved for the future.
TY .RAS	Bit 29	Device is a restricted device (i.e., can be assigned only by a privileged job or the MOUNT command).
TY .DEV	Bits 30-35	Device type code. Code 0 (.TYDSK) Disk of some sort Code 1 (.TYDTA) DECtape Code 2 (.TYMTA) Magnetic tape Code 3 (.TYTTY) TTY or equivalent Code 4 (.TYPTR) Paper-tape reader Code 5 (.TYPTP) Paper-tape punch Code 6 (.TYDIS) Display Code 7 (.TYLPT) Line printer Code 10 (.TYCDR) Card reader Code 11 (.TYCDP) Card punch Code 12 (.TYPTY) Pseudo-TTY Code 13 (.TYPLT) Plotter Code 14-57 Reserved for Digital Code 60-77 Reserved for customer

4.10.4 DEVSIZ AC, or CALLI AC, 101

This UUO is used to determine the buffer size for a device if the user wants to allocate core himself. The call is:

```
        MOVE AC, [EXP LOC]
        DEVSIZ AC,                  ;or CALLI AC, 101
        error return
        normal return
    LOC: EXP STATUS                 ;first word of the OPEN block
    LOC+1: SIXBIT /dev/             ;second word of the OPEN block
```

The error return is given if the UUO is not implemented. On a normal return, AC contains one of the following values:

If the mode is illegal, AC contains -2.

If the device does not exist, AC contains -1.

If the device exists, but its data mode is dump mode, AC contains 0.

If the device exists and the data mode is legal, AC contains in bits 0-17 the default number of buffers, and in bits 18-35 the default buffer size (including the first three words of the buffer).

4.10.5 WHERE AC, or CALLI AC, 63[1]

This UUO returns the physical station number of the specified device. When the UUO is called, AC contains either the channel number of the device as a right-justified quantity, or the device name as a left-justified SIXBIT quantity. The call is:

```
MOVE AC, [SIXBIT /dev/]          ;or MOVEI AC, channel no.
WHERE AC,                        ;or CALLI AC, 63
error return
normal return
```

If OPR is specified as the device name, the station number at which the job is logically located is returned; if OPR0 is specified, the station number of the central station is returned; and if TTY is specified, the station number at which the job's TTY is located is returned.

On a normal return, the LH of AC contains the station's status, and the RH of AC contains the station number associated with the device. The station's status is represented by the following bits:

Bit 13 = 1	if the station is dial-up (RM.SDU).
Bit 14 = 1	if the station is loaded (.RMSUL).
Bit 15 = 1	if the station is in the loading procedure (.RMSUG).
Bit 16 = 1	if the station is down (.RMSUD).
Bit 17 = 1	if the station is not in contact (.RMSUN).

The error return is taken if the UUO is not implemented, the specified channel is not INITed, or the requested device does not exist.

4.10.6 DEVNAM AC, or CALLI AC, 64

This UUO returns the physical name of a device obtained through either a generic INIT/OPEN or a logical device assignment. When the UUO is called, AC contains either channel number of the device as a right-justified quantity, or the device name as a left-justified SIXBIT quantity. The call is:

```
MOVE AC, [SIXBIT /dev/]          ;or MOVEI AC, channel no.
DEVNAM AC,                       ;or CALLI AC, 64
error return
normal return
```

The normal return is taken if the specified device is found, and AC contains the SIXBIT physical device name.

The error return is taken if the UUO is not implemented (AC is unchanged), the specified channel is not INITed, or no such device exists.

---

[1] This UUO depends on FTREM which is normally off in the DECsystem-1040.

CHAPTER 5
I/O PROGRAMMING FOR NONDIRECTORY DEVICES

This chapter explains the unique features of each standard nondirectory I/O device. Each device accepts the programmed operators explained in Chapter 4, unless otherwise indicated. Table 5-1 is a summary of the characteristics of all nondirectory devices. Buffer sizes are given in octal and include three bookkeeping words. The user may determine the physical characteristics associated with a logical device name by calling the DEVCHR UUO (refer to Paragraph 4.10.2).

Table 5-1
Nondirectory Devices

Device	Physical Name	Controller Number	Unit Number	Programmed Operators	Data Modes	Buffer Size (Octal)†
Card Punch	CDP	-	CP10A	OUTPUT, OUT	A, AL, I, IB, B	35
Card Reader	CDR, CDR1	-	CR10A 461 (PDP-6)	INPUT, IN	A, AL, I, IB, B, SI	36
Console Terminal	CTY	-	LT33A, LT33B LT35A, LT37AC 626 (PDP-6)	INPUT, IN OUTPUT, OUT	A, AL, I	23
Display	DIS	-	VR30, VP10 340B, 30	INPUT, OUTPUT	ID	Dump only
Line Printer	LPT, LPT1		LP10C	OUTPUT	A, AL, I	37
Magnetic Tape	MTA0, MTA1, ..., MTA7	TM10A TM10B 516(PDP-6)	TU20A, TU20B TU30A, TU30B	INPUT, IN OUTPUT, OUT MTAPE	A, AL, I IB, B DR, D	203††
Paper-Tape Punch	PTP	-	PC09 761(PDP-6)	OUTPUT, OUT	A, AL, I IB, B	43
Paper-Tape Reader	PTR	-	PC09 760(PDP-6)	INPUT, IN	A, AL, I IB, B	43

†Buffer sizes are subject to change and should be calculated rather than assumed by user programs. A DEVSIZ UUO may be employed.

††The buffer size for magnetic tape may be changed with the SET BLOCKSIZE monitor command (refer to the DECsystem-10 Operating System Commands).

Table 5-1 (Cont)
Nondirectory Devices

Device	Physical Name	Controller Number	Unit Number	Programmed Operators	Data Modes	Buffer Size (Octal)†
Plotter	PLT	XY10	XY10A XY10B	OUTPUT, OUT	A, AL, I IB, B	46
Pseudo-TTY	PTY	-	-	INPUT, IN OUTPUT, OUT	A, AL	23
Terminal	TTY0, TTY1, ..., TTY177	DC10 DC68A 630(PDP-6)	LT33A, LT33B LT35A, LT37AC VT06	INPUT, IN OUTPUT, OUT TTCALL	A, AL, I	23

†Buffer sizes are subject to change and should be calculated rather than assumed by user programs. A DEVSIZ UUO may be employed.

5.1 CARD PUNCH

The device mnemonic is CDP; the buffer size is dependent on the data mode.

Data Mode	Buffer Size
A, AL	23_8 (20_8 data) words - 80 7-bit ASCII characters
I, IB	36_8 (33_8 data) words - 80 12-bit bytes
B	35_8 (32_8 data, 33_8 punched) words - 26 data words, word count and checksum punched.

5.1.1 Concepts

The header card is the first card of an ASCII file and identifies the card code used (refer to Appendix C). This card is not punched for data modes other than ASCII. The header card has the same punches in all columns. This punch depends on the card code used; for example, in DEC026, the header card has 12-2-4-8 punched in columns 1-80.

The end-of-file (EOF) card is the last card of each output file. This card is punched for all data modes. The end-of-file card has a 12-11-0-1-6-7-8-9 punch in columns 1 through 80.

5.1.2 Data Modes

5.1.2.1 ASCII, Octal Code 0 - ASCII characters are converted to card codes and punched (up to 80 characters per card). Tabs are simulated by punching from 1 to 8 blank columns; form-feeds and carriage returns are ignored.

Line-feeds cause a card to be punched. All other nontranslatable ASCII characters cause a question mark to be punched. Cards can be split between buffers. Attempting to punch more than 80 columns per card causes the error bit IO.BKT (bit 21 of status word) to be set. The CLOSE will punch the last partial card and then punch an EOF card.

Cards are normally punched with DEC026 card codes. If bit 29 (octal 100) of the status word is on (from INIT, OPEN, or SETSTS), cards are punched with DEC029 codes (refer to Appendix C). The first card of any file (the header card) indicates the card code used (12-0-2-4-6-8 punched in column 1 for DEC029 card codes; 12-2-4-8 punched in column 1 for DEC026 card codes).

5.1.2.2 ASCII Line, Octal Code 1 - The same as ASCII mode.

5.1.2.3 Image, Octal Code 10 - In image mode, each buffer contains 27 words, each of which contains three 12-bit bytes. Each byte corresponds to one card column. Since there is room for 81 columns in the buffer (3 x 27) and there are only 80 columns on a card, the last word contains only 2 bytes of data; the third byte is thrown away. If the byte size is set by the program to be 12-bit bytes (the monitor normally sets 36-bit bytes), the program must skip the last byte in the buffer. Image binary causes exactly one card to be punched for each output. A program should not force an output every 80 columns since, if the program is in spooled mode, it will waste a large amount of disk space. The CLOSE punches the last partial card and then punches an EOF card.

5.1.2.4 Image Binary, Octal Code 13 - Same as Image.

5.1.2.5 Binary, Octal Code 14 - Column 1 contains the word count in rows 12-3. A 7-9 punch is in column 1. Column 2 contains a checksum as described for the paper-tape reader (refer to Paragraph 5.7.1.5); columns 3 through 80 contain up to 26 data words, 3 columns per word. Binary causes exactly one card to be punched for each output. The CLOSE punches the last partial card and then punches an EOF card.

5.1.3 Special Programmed Operator Service

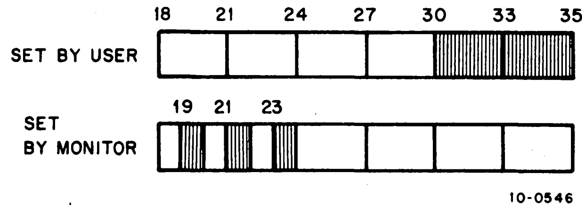
Following a CLOSE, an EOF card is punched. Columns 2 through 80 of the header card and the EOF card contain the same punches that appear in column 1 of the respective card for easy file identification. These laced punches are ignored by the card reader service routine.

After each interrupt, the card punch stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

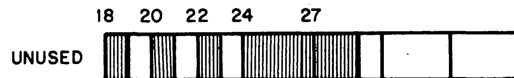
5.1.4 File Status (Refer to Appendix D)

The file status of the card punch is shown below.

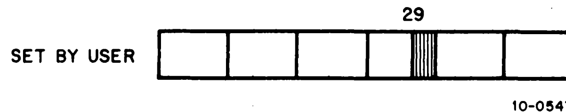
Standard Bits



Bit 19 - IO.DER	Punch error
Bit 21 - IO.BKT	Reached end-of-card with data remaining in buffer.
Bit 23 - IO.ACT	Device is active.



Device Dependent Bits



Bit 29 - IO.D29	If 1, punch DEC029 card codes in ASCII mode. If 0, punch DEC026 codes.

5.2 CARD READER

The card reader device mnemonic is CDR; the buffer size is 36_8 (33_8 data) words.

5.2.1 Concepts

For ASCII input, a header card can be the first card of the file and identifies the card code used (DEC026 or ANSI standard). The header card is used only when changing from or back to installation standard on ASCII input. The header card must not be present with any other data modes; if present, the header card is treated as incorrect format or read as data. Refer to Appendix C for the card codes.

An EOF card may have one of three forms: 12-11-0-1 punched in column 1, 6-7-8-9 punched in column 1, or a logical OR of the two punched in column 1. Columns 2 through 80 are ignored. The EOF card has the same effect as the EOF key on the card reader. This key must be depressed or the end-of-file card must be present at the end of each input file for all data modes.

To be compatible with PDP-11 operating systems, the DECsystem-10 card reader service accepts several other header card-code cards and EOF cards. Only column 1 is looked at; columns 2-80 are ignored.

	Punched by DECsystem-10	Also accepted
EOF	12-11-0-1-6-7-8-9[†]	12-11-0-1 6-7-8-9
DEC026	12-2-4-8	12-11-8-9[†]
ANSI	12-0-2-4-6-8	12-0-7-9

5.2.2 Data Modes

5.2.2.1 ASCII, Octal Code 0 - All 80 columns of each card are read and translated to 7-bit ASCII code. Blank columns are translated to spaces. At the end of each card a carriage return/line feed is appended. As many complete cards as can fit are placed in the input buffer, but cards are not split between two buffers. Using the standard-sized buffer, only one card is placed in each buffer.

Cards are normally translated as DEC026 card codes (refer to PDP-10 System Reference Manual). If a DEC029 header card is encountered, any following cards are translated as DEC029 codes (refer to Appendix C) until the 029 conversion mode is turned off. The 029 mode is turned off either by a RELEASE command or by a DEC026 header card. Columns 2 through 80 of both of these cards are ignored.

5.2.2.2 ASCII Line, Octal Code 1 - This mode is the same as ASCII mode.

5.2.2.3 Image, Octal Code 10 - All 12 punches in all 80 columns are packed into the buffer as 12-bit bytes. The first 12-bit byte is in column 1. The last word of the buffer contains columns 79 and 80 as the left and middle bytes, respectively. The EOF button is processed as in ASCII mode. Cards are not split between two buffers.

5.2.2.4 Image Binary, Octal Code 13 - This mode is the same as Image.

5.2.2.5 Binary, Octal Code 14 - Card column 1 must contain a 7-9 punch to verify that the card is in binary format. Column 1 also contains the word count in rows 12 through 3. The absence of the 7-9 punch results in setting the IO.IMP (bit 18 of status word) flag in the card reader status word. Card column 2 must contain a 12-bit checksum as described for the paper-tape binary format. Columns 3 through 80 contain binary data, 3 columns per word for up to 26 words. Cards are not split between two buffers. The EOF button is processed the same as in ASCII mode.

5.2.2.6 Super-Image, Octal Code 110[††] - Super-image mode may be initialized by setting bit 29 of the card reader's IOS word. This mode causes the 36 bits read from the I/O bus to be BLKI'd directly to the user's buffer. For this mode, the default size of the input buffer is 81_{10} words (80_{10} data words).

[†] These cards are symmetric in the sense that the pattern of the punches is the same if the card is turned upside down.

[††] This mode depends on FTCDRSI which is normally off in the DECsystem-1040.

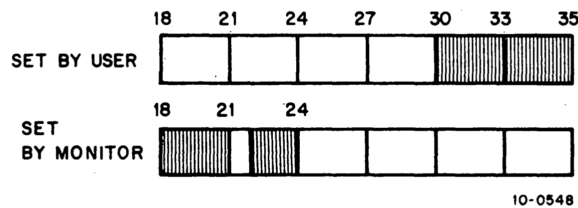
5.2.3 Special Programmed Operator Service

The card reader, after each interrupt, stores the results of a CONI in the DEVSTS word in the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

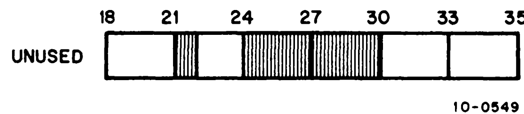
5.2.4 File Status (Refer to Appendix D)

The file status of the card reader is shown below.

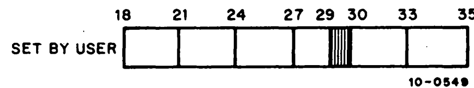
Standard Bits



Bit 18 - IO.IMP	7-9 punch absent in column 1 of a presumed binary card. The card reader is stopped.
Bit 19 - IO.DER	Photocell error, card motion error, data missed. The card reader is stopped.
Bit 20 - IO.DTE	Computed checksum is not equal to checksum read on binary card. The card reader is stopped.
Bit 22 - IO.EOF	EOF card read or EOF button pressed.
Bit 23 - IO.ACT	Device is active.



Device-Dependent Bits



Bit 29 - IO.SIM Super-Image mode.

5.3 DISPLAY WITH LIGHT PEN

The device mnemonic is DIS; there is no buffer because the display uses device-dependent dump mode only.

5.3.1 Data Modes

For IMAGE DUMP, Octal Code 15, an arbitrary length in the user area may be displayed on the scope. The command list format is as described in Chapter 4 with the addition for the Type 30, VR30 and VP10 display, that, if RH = 0, and LH ≠ 0, then LH specifies the intensity for the following data (4 to 13).

5.3.2 Background

During timesharing on a heavily-loaded system, the monitor service routine for the Type 30, VR30, and VP10 guarantees a flicker-free picture on the display if the job is locked in core. To maintain this picture, the picture data must be available for the display at least every two jiffies. If the system is lightly loaded, it is not necessary to keep the job in core. When the job is swapped, a minimum amount of flicker may occur, but the job has high priority to be swapped-in again.

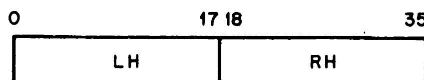
5.3.3 Display UUOs

The I/O UUOs for both displays operate as follows:

```
INIT D, 15              ;MODE 15 ONLY
SIXBIT /DIS/            ;DEVICE NAME
0                       ;NO BUFFERS USED
ERROR RETURN            ;DISPLAY NOT AVAILABLE
NORMAL RETURN
CLOSE D,                ;STOPS DISPLAY AND
   or                   ;RELEASES DEVICE AS
RELEAS D,               ;DESCRIBED IN CHAPTER 4
```

5.3.3.1 INPUT D, ADR - If a light pen hit has been detected since the last INPUT command, then C(ADR) is set to the location of last light pen hit. If no light pen hit has been detected since last INPUT command, then C(ADR) is set to -1.

5.3.3.2 OUTPUT D, ADR - ADR specifies the first address of a table of pointers. This table is composed of pointers with the following format:



10-0550

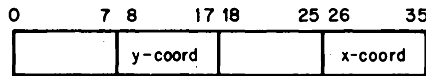
For the Type 30, VR30 and VP10 Display:

If LH = 0 and RH = 0, then this is the end of the command list.

If LH ≠ 0 and RH = 0, then LH is the desired intensity for the following data or commands. The intensity ranges from 4 to 13, where 4 is the dimmest and 13 is the brightest.

If LH = 0 and RH ≠ 0, then RH is the address of the next pointer. Successive pointers are interpreted beginning at RH.

If LH ≠ 0 and RH ≠ 0, then -LH words beginning at address RH + 1 are output as data to the display. The format of the data word is the following:



For the Type 340B Display:

If RH = 0, then this is the end of the command list.

If LH = 0 and RH ≠ 0, then RH is the address of the next pointer. Successive pointers are interpreted beginning at RH.

If LH ≠ 0 and RH ≠ 0, then -LH words beginning at address RH+1 are output as data to the display. The format of the data word is described in the Precision Incremental CRT Display Type 340 Maintenance Manual.

An example of a valid pointer list for the VR-30 display is:

```
OUTPUT  D, LIST                 ;OUTPUT DATA
                                ;POINTED TO BY LIST
LIST;   XWD     5, 0            ;INTENSITY 5 (DIM)
        IOWD    1, A            ;PLOT A
        IOWD    5,SUBP1         ;PLOT SUBPICTURE 1
        XWD     13,0            ;INTENSITY 13 (BRIGHT)
        IOWD    1,C             ;PLOT C
        IOWD    2,SUBP2         ;PLOT SUBPICTURE 2
        XWD     0,LIST1         ;TRANSFER TO LIST 1

LIST1:  XWD     10,0            ;INTENSITY 10 (NORMAL)
        IOWD    1,B             ;PLOT B
        IOWD    1,D             ;PLOT D
        XWD     0,0             ;END OF COMMAND LIST
        OUTPUT  D, LIST         ;OUTPUT DATA
                                ;POINTED TO BY LIST
A:      XWD     6,6             ;Y=    6, X=6
B:      XWD    70,105           ;Y=   70, X=105
C:      XWD   105,70            ;Y=  105, X=70
D:      XWD  1000,200           ;Y=1000, X=200

SUBP1:  BLOCK   5               ;SUBPICTURE 1
SUB2:   BLOCK   2               ;SUBPICTURE 2
```

An example of a valid pointer list for the Type 340B Display is:

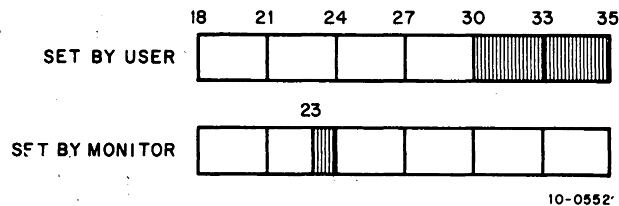
```
        OUTPUT   D, LIST         ;OUTPUT DATA POINTED
                                 ;TO BY POINTER IN LIST

LIST:   IOWD     1,A             ;SET STARTING POINT TO (6,6)
        IOWD     5,SUBP1         ;DRAW A CIRCLE
        IOWD     1,C             ;SET STARTING POINT TO (70,105)
        IOWD     5,SUBP1         ;DRAW A CIRCLE
        IOWD     1,B             ;SET STARTING POINT TO (105,70)
        IOWD     2,SUBP2         ;DRAW A TRIANGLE
        IOWD     0,LIST1         ;TRANSFER TO LIST1

LIST1:  IOWD     1,D             ;SET STARTING POINT TO
                                 ;(100,-200)
        IOWD     5,SUBP1         ;DRAW A CIRCLE
        IOWD     1,A             ;SET STARTING POINT TO (6,6)
        IOWD     2,SUBP2         ;DRAW A TRIANGLE
        XWD      0,0             ;STOP

A:      X=6      Y=6
B:      X=105    Y=70
C:      X=70     Y=105
D:      X=1000   Y=-200

SUBP1:  BLOCK    5               ;DRAW A CIRCLE
SUBP2:  BLOCK    2               ;DRAW A TRIANGLE
```

The example shows the flexibility of this format. The user can display a subpicture by setting up a pointer. He can also display the same subpicture in many different places by setting up pointers to the subpicture, each preceded by a pointer to commands for the display to reset its coordinates.

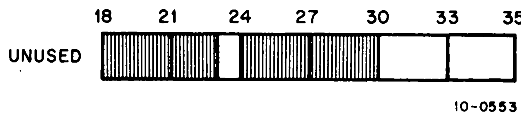
5.3.4 File Status (See Appendix D)

The file status of the display is shown below.

Standard Bits



Bit 23 - IO.ACT Device is active.



Device Dependent Bits - None.

5.4 LINE PRINTER

The device mnemonic is LPT; the buffer size is 37_8 (36_8 data) words.

5.4.1 Data Modes

5.4.1.1 ASCII, Octal Code 0 - ASCII characters are transmitted to the line printer exactly as they appear in the buffer. Refer to the PDP-10 System Reference Manual for a list of the vertical spacing characters.

5.4.1.2 ASCII Line, Octal Code 1 - This mode is exactly the same as ASCII and is included for programming convenience. All format control must be performed by the user's program; this includes placing a RETURN, LINE-FEED sequence at the end of each line.

5.4.1.3 Image, Octal Code 10 - This mode is the same as ASCII mode.

5.4.2 Special Programmed Operator Service

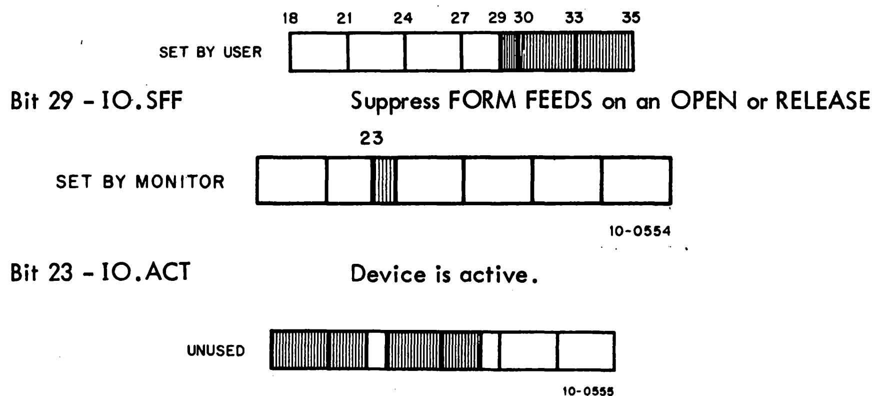
The first output programmed operator of a file and the CLOSE at the end of a file cause an extra form-feed to be printed to keep files separated.

After each interrupt, the line printer stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

5.4.3 File Status (See Appendix D)

The file status of the line printer is shown below.

Standard Bits



Device dependent bits - None.

5.5 MAGNETIC TAPE

Magnetic tape format is industry compatible, 7- or 9-channel 200, 556, and 800 bits/in. (see description below). The device mnemonic is MTA0, MTA1, ..., MTA7; the buffer size is 203_8 (200_8 data) words. The user may change the density and/or the blocksize of a magnetic tape by using the SET DENSITY and SET BLOCKSIZE monitor commands (refer to the DECsystem-10 Operating System Commands).

5.5.1 Data Modes

5.5.1.1 ASCII, Octal Code 0 - Data appears to be written on magnetic tape exactly as it appears in the buffer. No processing or checksumming of any kind is performed by the service routine. The parity checking of the magnetic tape system is sufficient assurance that the data is correct. Normally, all data, both binary and ASCII, is written with odd parity and at 800 bits per inch unless changed by the installation. A maximum of 200_8 words per record is allowed if the monitor has set up the buffer ring. If the user builds his own buffers, he may specify any number of words per record. The word count is not written on the tape. If an I/O error occurs or an end-of-tape is reached, reading ahead ceasing on input and implied output ceases on output.

5.5.1.2 ASCII Line, Octal Code 1 - The mode is the same as ASCII.

5.5.1.3 Image, Octal Code 10 - The mode is the same as ASCII, but data consists of 36-bit words.

5.5.1.4 Image Binary, Octal Code 13 - The mode is the same as Image.

5.5.1.5 Binary, Octal Code 14 - The mode is the same as Image.

5.5.1.6 DR (Dump Records), Octal Code 16 - Standard fixed length records (128 words is the standard unless installation standard is changed at MONGEN time) are read into or written from anywhere in the user's core area without regard to the standard buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is described in Chapter 4. For input operations a new record is read for each word in the command list (except GOTO words); if the record terminates before the command word is satisfied, the service routine reads the next records. If the command word runs out before the record terminates, the remainder of the record is ignored. For each output command word, enough standard length records are followed by one short record to exactly write all of the words on the tape. If an I/O error occurs or the end-of-tape is reached, no additional commands are retrieved from a dump mode command list, and I/O is terminated. When the end of file is read, the user receives the standard EOF return (the error return from the INPUT or IN UUO) and the IO.EOF bit is set in the file status word. This bit can be retrieved with the GETSTS

UUO. The EOF character (17_8 for 7-channel tapes or 23_8 for 9-channel tapes) is read into the user's buffer. The next INPUT or IN UUO will read the next record on the tape.

5.5.1.7 D (Dump), Octal Code 17 - Variable length records are read into or written from anywhere in the user's core area without regard to the standard buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is described in Chapter 4. For input operations a new record is read for each word in the command list (except GOTO words); if the record terminates before the command word is satisfied, the service routine skips to the next command word. If the command word runs out before the record terminates, the remainder of the record is ignored. For each output command word, exactly one record is written. Refer to Paragraph 5.5.1.6 for the description of EOF handling in dump mode.

5.5.2 Magnetic Tape Format

Magnetic tape format can generally be described as unlabelled, industry-compatible format. That is, as far as the user is concerned, the tape contains only data records and EOF marks, which signal the end of the data set or the end of the file.

An EOF mark consists of a record containing a 17_8 (for 7-channel tapes) or a 23_8 (for 9-channel tapes). EOF marks are used in the following manner:

a. No EOF mark precedes the first file on a magnetic tape.
b. An EOF mark follows every file.
c. Two EOF marks follow a file if that file is the last or only file on the tape.

Files are sequentially written on and read from a magnetic tape. A file consists of an integral number of physical records, separated from each other by interrecord gaps (area on tape in which no data is written). There may or may not be more than one logical record in each physical record.

5.5.3 Special Programmed Operator Service

CLOSE performs a special function for magnetic tape. When an output file is closed (both dump and nondump), the I/O service routine automatically writes two EOF marks and backspaces over one of them. If another file is opened, the second EOF mark is wiped out leaving one EOF mark between files. At the end of the in-use portion of the tape, however, a double EOF character, which is defined as the logical end of tape, appears.

After each interrupt, the magnetic tape service routine stores the results of a CONI in the DEVSTS word. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

5.5.3.1 MTAPE UUO - The MTAPE programmed operator provides for tape manipulation functions such as rewind, backspace record, backspace file, and 9-channel tape initialization. The format is

MTAPE D, FUNCTION

where D is the device channel on which the magnetic tape unit is initialized. FUNCTION is selected according to Table 5-2.

Table 5-2
MTAPE Functions

Symbol	Function	Action
MTWAT.	0	No operation; wait for spacing and I/O to finish.
MTREW.	1	Rewind to load point.
MTEOF.	3	Write EOF.
MTSKR.	6	Skip one record.
MTBSR.	7	Backspace record.
MTEOT.	10	Space to logical end of tape; terminates at either two consecutive EOF marks or at the end of first record beyond end of tape marker.
MTUNL.	11	Rewind and unload.
MTBLK.	13	Write 3 in. of blank tape.
MTSKF.	16	Skip one file; implemented by a series of skip record operations.
MTBSF.	17	Backspace files; implemented by a series of backspace record operations.
MTDEC.	100	Initialize for Digital-compatible 9-channel.[†]
MTIND.	101	Initialize for industry-compatible 9-channel tape.[††]

[†]Digital-compatible mode writes (or reads) 36 data bits in five frames of a 9-track magnetic tape. It can be any density, any parity, and is not industry compatible. This mode is in effect until a RELEAS D, or a MTIND.D, is executed.

[††]Industry-compatible 9-channel mode writes (or reads) 32 data bits per word in four frames of a 9-track magtape and ignores the low order four bits of a word. It must be 800 bits/in. density, odd parity.

MTAPE waits for the magnetic tape unit to complete the action in progress before performing the indicated function, including no operation (0). Bits 18 through 25 of the status word are then cleared, the indicated function is initiated, and control is immediately returned to the user's program. It is important to remember that when performing buffered input/output, the I/O service routine can be reading several blocks ahead of the user's program. MTAPE affects only the physical position of the tape and does not change the data that has already been read into the buffers. Therefore, an INPUT

or OUTPUT following a MTAPE may not retrieve the buffer containing the block requested. However, a single buffer ring retrieves the expected block since the device must stop after each INPUT or OUTPUT. Alternatively, if bit 30 (IO.SYN) of the file status word is set via the INIT or SETSTS UUO, the device stops after each buffer is filled on an INPUT or OUTPUT. Thus, the MTAPE will apply to the buffer supplied on the next INPUT or OUTPUT.

MTAPE functions must be followed by MTAPE 0 if subsequent operations depend on the completion of the MTAPE function. If this is not done, subsequent input and output UUOs are ignored until the magnetic tape control is freed. This problem occurs frequently in programs that issue a REWIND at the beginning of the program. The tape may actually be positioned at the beginning of the tape; however, the processing of the MTAPE function may cause the first input to be ignored.

Issuing a backspace file command to a magnetic tape unit moves the tape in the reverse direction until the tape has:

a. passed the end of file mark
b. reached the beginning of the tape.

The end of the backspace file operation positions the tape heads either immediately in front of a file mark or at the beginning of the tape.

In most cases it is desirable to skip forward over this file mark. This is decidedly not the case if the beginning of the tape is reached; in this case, giving a skip file command would skip the entire first file on the tape stopping at the beginning of the second file, rather than leaving the tape positioned at the beginning of the first file. Therefore, a typical (incorrect) sequence for backspace file would be:

```
MTBSF. MT,              ;Backspace file
WAIT MT,                ;Wait for completion
STATO MT,4000           ;Beginning of tape?
MTSKF. MT,              ;No, skip over file mark
```

It is necessary to wait after the backspace file instruction to ensure that the tape is moved to the EOF mark or the beginning of the tape before testing to see whether or not it is the beginning of the tape. The instruction WAIT MT, cannot be used for this purpose; it waits only for the completion of I/O transfer operation. (Backspace file is a spacing operation, not an I/O transfer operation.) Instead, use the following sequence for backspace file:

```
MTAPE MT,17             ;Backspace file
MTAPE MT,0              ;Wait for completion
STATO MT,4000           ;Beginning of tape?
MTAPE MT,16             ;No, skip over file mark
```

The device service routine must wait until the magnetic tape control is free before processing the MTAPE MT, 0 command, which tells the tape control to do nothing. Thus, the service routine achieves the waiting period necessary for the completion of the previous operation and the proper positioning of the tape.

5.5.3.2 MTCHR. AC, or CALLI AC, 112[1] - This UUO returns the characteristics (presently only the density is returned) of the specified magnetic tape drive. The density of the drive can be specified by setting bits 27 and 28 in the status word when the drive is INITed and can be changed with the SET DENSITY command. The call is:

```
MOVE AC, [SIXBIT/dev/]          ;or MOVEI AC, channel number
MTCHR. AC,                      ;or CALLI AC, 112
error return
normal return
```

In determining the density to return, the monitor examines the initial file status specified with the INIT UUO and returns the indicated density value. If this value is zero, the monitor then determines if the user specified a density with the SET DENSITY system command. If no density has been specified in this way, the monitor returns the system default density.

The MTCHR. UUO is used to obtain more complete information than that returned with the GETSTS UUO. The GETSTS UUO returns only the density specified in the INIT UUO and if the density is specified as zero (for the system default), zero is returned, not the actual system default. The density specified in the SET DENSITY command cannot be returned with the GETSTS UUO.

The error return is given if the UUO is not implemented (AC remains unchanged) or if there is no device on the specified channel or if the device is not a magnetic tape (AC contains -1).

On a normal return, bits 34 and 35 of AC contain the current density of the magnetic tape drive:

AC contains 1	200 bpi
AC contains 2	556 bpi
AC contains 3	800 bpi

5.5.4 9-Channel Magtape

Nine-channel magtape may be written and read in two ways: normal Digital-compatible format and industry-compatible format.

[1]This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

5.5.4.1 Digital-Compatible Mode - Digital-compatible mode, the usual mode, allows old 7-channel user mode programs to read and write 9-channel tapes with no modification. Digital-compatible mode writes 36 data bits in five bytes of a nine track magtape. It can be any density, and parity, and is not industry compatible. The software mode is specified in the usual manner during initialization or with a SETSTS. User mode I/O is handled precisely as 7-track magtape. It is assumed that most DEC magtapes will be written and read in Digital-compatible mode.

For the data word in core there are 5 magnetic tape bytes per 36-bit word. Parity bits are unavailable to the user. Bits are written on tape as shown above; bits 30 and 31 are written twice and tracks 8 and 9 of byte 5 contain 0. On reading, parity bits and tracks 8 and 9 of byte 5 are ignored, the OR of bits (B30) is read into bit 30 of the data word, the OR of bits (B31) is read into bit 31.

Data Word on Tape

Tracks								
9	8	7	6	5	4	3	2	1
B0	B1	B2	B3	B4	B5	B6	B7	P
B8	B9	B10	B11	B12	B13	B14	B15	P
B16	B17	B18	B19	B20	B21	B22	B23	P
B24	B25	B26	B27	B28	B29	(B30)	(B31)	P
0	0	(B30)	(B31)	B32	B33	B34	B35	P
P = Parity BN = Bit N in core.								

5.5.4.2 Industry-Compatible Mode - For reading and writing industry-compatible 9-channel magtapes, an MTAPE D, 101 UUO must be executed to set the status. MTAPE D, 101 is meaningful for 9-channel magtape only and is ignored for all other devices. In the left half of the status word, bit 2 (which cannot be read by the user program) may be cleared, thus, the device is returned to 9-channel Digital-compatible status by a RELEAS, a call to EXIT, or an MTAPE D, 100 UUO. These MTAPE UUOs act only as a switch to and from industry-compatible mode and affect I/O status only by setting the density to 800 bits/in. and odd parity.

On INPUT, four 8-bit bytes are read into each word in the buffer, left justified, with the remaining four bits of the word containing character parity error indicators corresponding to the 8-bit bytes.

On OUTPUT, the leftmost four 8-bit bytes of each word in the buffer are written out in four frames, with the remaining four rightmost bits of the word being ignored.

Data Word on Tape

Tracks								
9	8	7	6	5	4	3	2	1
B0 B8 B16 B24	B1 B9 B17 B25	B2 B10 B18 B26	B3 B11 B19 B27	B4 B12 B20 B28	B5 B13 B21 B29	B6 B14 B22 B30	B7 B15 B23 B31	B32 B33 B34 B35

For data word in core, four magnetic tape bytes carry four 8-bit bytes from the data word. Parity bits are obtained as shown above when reading. The rightmost four bits (32-35) are ignored on writing.

5.5.4.3 Changing Modes - MTAPE CH, 101 automatically sets density at 800 bits (i.e., 800 eight-bit bytes) per inch and sets odd parity. Note that buffer headers are set up, when necessary by the monitor in the usual manner according to the I/O mode in which the device is initialized. In order to operate on eight-bit bytes, the user must insert the byte size in the byte pointer before the first IN or OUT.

5.5.5 File Status (refer to Appendix D)

The file status of the magnetic tape is shown below.

Standard Bits

SET BY USER: bits 18, 21, 24, 27, 30, 33, 35
SET BY MONITOR: bits 18, 21, 24

10-0556

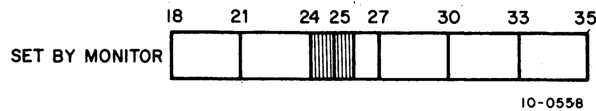
Bit 18 - IO.IMP	Unit was write-locked when output was attempted, or illegal operation was specified to the magnetic-tape control.
Bit 19 - IO.DER	Data was missed, tape is bad, or transport is hung.
Bit 20 - IO.DTE	Parity error.
Bit 21 - IO.BKT	Record read from tape exceeds buffer size.
Bit 22 - IO.EOF	EOF mark encountered. A 17_8 (for 7-channel tapes) or a 23_8 (for 9-channel tapes) appears in buffer.
Bit 23 - IO.ACT	Device is active.

Device Dependent Bits

SET BY USER: bits 18, 21, 24, 26, 27, 30, 33, 35

10-0557

Bit 26 - IO.PAR	I/O parity. 0 for odd parity, 1 for even parity. Odd parity is preferred. Even parity should be used only when creating a tape to be read in binary coded decimal (BCD) on another computer.
Bit 27-28 - IO.DEN	I/O density. 00 = System standard. Defined at MONGEN time and can be changed with the SET DENSITY command. 01 = 200 bits/in. 10 = 556 bits/in. 11 = 800 bits/in.
Bit 29 - IO.NRC	I/O no read check. Suppress automatic error correction if bit 29 is 1. Normal error correction repeats the desired operation 10 times before setting an error status bit.



Bit 24 - IO.BOT I/O beginning of tape. Unit is at beginning of tape mark.

Bit 25 - IO.EOT I/O tape end. Physical end of tape mark encountered.

5.6 PAPER-TAPE PUNCH

The device mnemonic is PTP; the buffer size is 43_8 (40_8 data) words.

5.6.1 Data Modes

5.6.1.1 ASCII, Octal Code 0 - The eighth hole is punched when necessary in order to make even parity. Tape-feed without the eighth hole (000) is inserted after form-feed. A rubout is inserted after each vertical or horizontal tab. Null characters (000) appearing in the buffer are not punched.

5.6.1.2 ASCII Line, Octal Code 1 - The mode is the same as ASCII mode. Format control must be performed by the user's program.

5.6.1.3 Image, Octal Code 10 - Eight-bit characters are punched exactly as they appear in the buffer with no additional processing.

5.6.1.4 Image Binary, Octal Code 13 - Binary words taken from the output buffer are split into six 6-bit bytes and punched with the eighth hole punched in each line. There is no format control or checksumming performed by the I/O routine. Data punched in this mode is read back by the paper-tape reader in the IB mode.

5.6.1.5 Binary, Octal Code 14 - Each bufferful of data is punched as one checksummed binary block as described for the paper-tape reader. Several blank lines are punched after each bufferful for visual clarity.

5.6.2 Special Programmed Operator Service

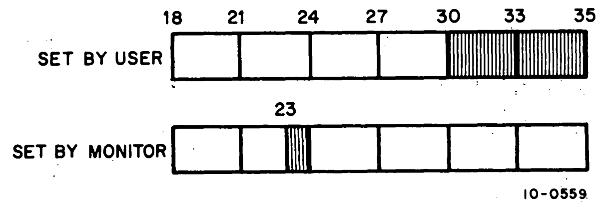
The first output programmed operator of a file causes approximately two fanfolds of blank tape to be punched as leader. Following a CLOSE, an additional fanfold of blank tape is punched as trailer. No EOF character is punched automatically.

After each interrupt, the paper-tape punch stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

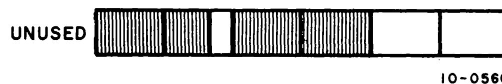
5.6.3 File Status (Refer to Appendix D)

The file status for the paper-tape punch is shown below.

Standard Bits



Bit 23 - IO.ACT Device is active.



Device Dependent Bits - None.

5.7 PAPER-TAPE READER

The device mnemonic is PTR; the buffer size is 43_8 (40_8 data) words.

5.7.1 Data Modes (Input Only)

NOTE

To initialize the paper-tape reader, the input tape must be threaded through the reading mechanism and the FEED button must be depressed.

5.7.1.1 ASCII, Octal Code 0 - Blank tape (000), RUBOUT (377), and null characters (200) are ignored. All other characters are truncated to seven bits and appear in the buffer. The physical end of the paper tape serves as an EOF, but does not cause a character to appear in the buffer.

5.7.1.2 ASCII Line, Octal Code 1 - Character processing is the same as for ASCII mode. The buffer is terminated by LINE FEED, FORM, or VT.

5.7.1.3 Image, Octal Code 10 - There is no character processing. The buffer is packed with 8-bit characters exactly as read from the input tape. Physical end of tape is the EOF indication but does not cause a character to appear in the buffer.

5.7.1.4 Image Binary, Octal Code 13 - Characters not having the eighth hole punched are ignored. Characters are truncated to six bits and packed six to the word without further processing. This mode is useful for reading binary tapes having arbitrary blocking format.

5.7.1.5 Binary, Octal Code 14 - Checksummed binary data is read in the following format. The right half of the first word of each physical block contains the number of data words that follow and the left contains half a folded checksum. The checksum is formed by adding the data words using 2's complement arithmetic, then splitting the sum into three 12-bit bytes and adding these using 1's complement arithmetic to form a 12-bit checksum. The data error status flag (refer to Table 4-3 in Paragraph 4.6.2) is raised if the checksum miscompares. Because the checksum and word count appear in the input buffer, the maximum block length is 40. The byte pointer, however, is initialized so as not to pick up the word count and checksum word.

Again, physical end of tape is the EOF indication, but does not result in putting a character in the buffer.

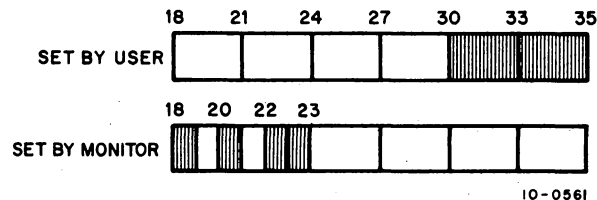
5.7.2 Special Programmed Operator Service

After each interrupt, the paper-tape reader stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

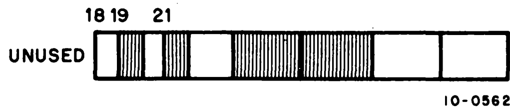
5.7.3 File Status (Refer to Appendix D)

The file status of the paper-tape reader is shown below.

Standard Bits



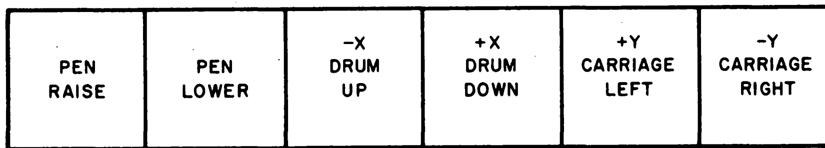
Bit 18 - IO.IMP	Binary block is incomplete.
Bit 20 - IO.DTE	Bad checksum in binary mode.
Bit 22 - IO.EOF	Physical end of tape is encountered. No character is stored in the buffer.
Bit 23 - IO.ACT	Device is active.



Device dependent bits - None

5.8 PLOTTER

The device mnemonic is PLT; the buffer size is 43_8 (40_8 data) words. The plotter takes 6-bit characters with the bits of each character decoded as follows:



Do not combine PEN RAISE or LOWER with any of the position functions. (For more details on the incremental plotter, refer to the PDP-10 System Reference Manual.)

5.8.1 Data Modes

5.8.1.1 ASCII, Octal Code 0 - Five 7-bit characters per word are transmitted to the plotter exactly as they appear in the buffer. The plotter is a 6-bit device; therefore, the leftmost bit of each character is ignored.

5.8.1.2 ASCII Line, Octal Code 1 - This mode is identical to ASCII mode.

5.8.1.3 IMAGE, Octal Code 10 - Six 6-bit characters per word are transmitted to the plotter exactly as they appear in the buffer.

5.8.1.4 IMAGE BINARY, Octal Code 13 - This mode is identical to Image mode.

5.8.1.5 BINARY, Octal Code 14 - This mode is identical to Image mode.

5.8.2 Special Programmed Operator Service

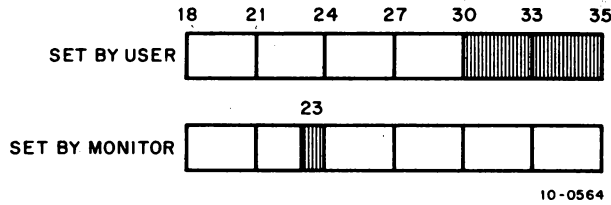
The first OUTPUT operator causes the plotter pen to be lifted from the paper before any user data is sent to the plotter. The CLOSE operator causes the plotter pen to be lifted after all user data is sent to the plotter. These two pen-up commands are the only modifications the monitor makes to the user output file.

After each interrupt, the plotter stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

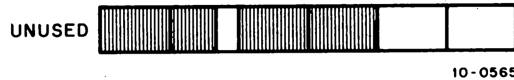
5.8.3 File Status (Refer to Appendix D)

The file status of the plotter is shown below.

Standard bits



Bit 23 - IO.ACT Device is active.



Device dependent bits - None

5.9 PSEUDO-TTY

The device mnemonic is PTY0, PTY1, ..., PTYn. (The number of pseudo-TTYs is specified at MONGEN time.) The buffer size is 23_8 (20_8 data) words.

5.9.1 Concepts

Each job in the DECsystem-10 is usually initiated by a user at a physical terminal. Except in the case of a DETACH operation, the job remains under the control of the user's terminal until it is terminated by either the KJOB command or the LOGOUT UUO. For each physical terminal there is a block of core in the monitor, containing information about the physical terminal and including two buffers as the link between the physical terminal and the job. It is through these buffers that the terminal sends input to the job, and the job returns output to the terminal.

Sometimes it is desirable to allow a job in the DECsystem-10 to be initiated by a program instead of by a user. Since a program cannot use a physical terminal in the way a user can, some means must be provided in the monitor for the program to send input to and accept output from the job it is controlling. The monitor provides this capability via the pseudo-TTY (PTY). The PTY is a simulated terminal and is not defined by hardware. Like hardware-defined terminals, each PTY has a block of core associated with it. This block of core is used by the PTY in the same manner as a hardware-defined terminal uses its block of core. Figure 5-1 shows the parallel between a hardware-defined terminal and a software-defined PTY.

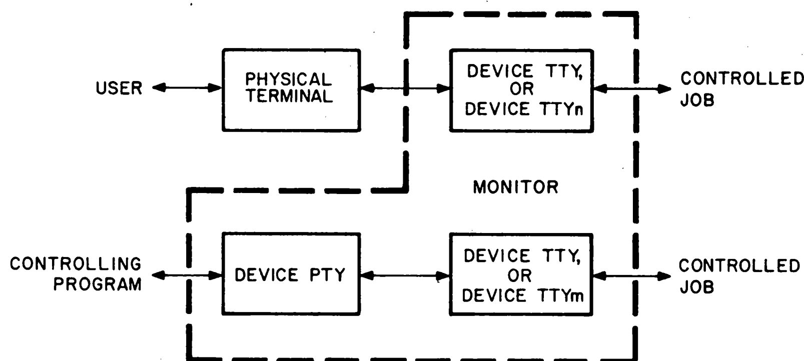


Figure 5-1 Pseudo-TTY

The controlling program, most commonly the batch processor, uses the PTY in the same way a user uses a physical device. It initiates the PTY, inputs characters to and waits for output from the PTY, and closes the PTY using the appropriate programmed operators. The job controlled by the program performs I/O to the PTY as though the PTY were a physical terminal.

A controlled job may go into a loop and not accept any input from its associated buffer; therefore, it is not possible for the controlling program to simply rely on waiting for activity in the controlled job. A controlling program may also wish to drive more than one controlled job, and be able to respond to any of these jobs; therefore, the controlling program cannot wait for any particular PTY. For these two reasons, the PTY differs from other devices in that it is never in a I/O wait state. Timing is accomplished by the HIBER UUO and the status bits of the PTY.

5.9.2 The HIBER UUO

The HIBER UUO (refer to Paragraph 3.1.4.2) allows the controlling program to temporarily suspend its operation until either there is activity in the controlled job or the specified amount of sleep time runs out, whichever occurs first. If bit 12 in the AC is set in the HIBER UUO call, any PTY activity since

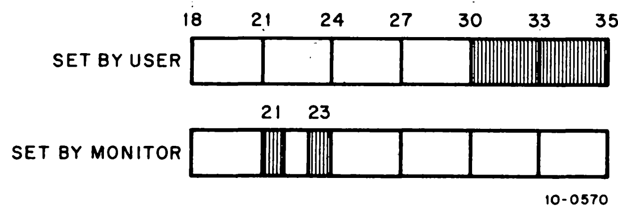
the last HIBER UUO causes the controlling program to be awakened. If no PTY activity occurs before the limit of sleep time is reached, the controlling program is activated, and it checks the controlled job's run time or other criteria to determine whether the job should be interrupted. If the job should be interrupted, the controlling program may output two control-C characters to stop the job. (A timesharing user stops a running job in the same way.) If the job should not be interrupted, the controlling program should repeat the HIBER UUO.

If bit 12 in AC is not set, unnecessary delays might result if activity occurred on a PTY while the controlling job was sleeping. To avoid these delays, a check is made when a PTY status bit changes to determine if the controlling program is in a sleep. If it is, the sleep time is cleared so the controlling program can service the PTY.

5.9.3 File Status (Refer to Appendix D)

The file status of the pseudo-TTY is shown below.

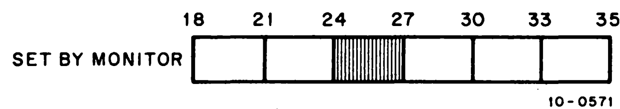
Standard Bits



Bit 21 - IO.BKT

Bit 23 - IO.ACT Device is active.

Device Dependent Bits



Bit 24 - IO.PTI	Job is in a TTY input wait. The controlling job should perform an OUTPUT to the PTY.
Bit 25 - IO.PTO	The TTY buffer has output to be read by an INPUT from the PTY.
Bit 26 - IO.PTM	Any characters typed into the TTY buffer (by OUTPUT to the PTY) are read by the monitor command decoder instead of by the controlled job (i.e., the controlled job is in monitor mode).

5.9.4 Special Programmed Operator Service

5.9.4.1 OUT, OUTPUT UUOs - The first OUTPUT operation after an INIT or OPEN causes the special actions of the RELEASE UUO (refer to Paragraph 5.9.4.3) and then the following normal output operations.

a. Characters from the controlling program's buffer ring are placed in the input buffer of the TTY linked to the PTY.

b. The IO.PTI bit is cleared.

c. The IO.PTM bit is set or cleared as determined by the state of the TTY.

The following are exceptions to the normal output action:

a. NULLS (ASCII 000) are discarded.

b. If more OUTPUTS are performed than are accepted by the controlled job and if the limit on this excess is exceeded, the IO.BKT bit is set and the remainder of the controlling program's buffer is discarded.

c. Lower case characters sent to the controlled job are translated to upper case if the appropriate bit in the TTY is set.

5.9.4.2 IN, INPUT UUOs - Characters are read from the output buffer of the TTY and are placed in the buffer ring of the controlling program. If there are no characters to read, an empty buffer is returned. The INPUT UUO does not cause a WAIT.

All the available characters are passed to the controlling program. If there are more characters to read than can fit in the buffer of the controlling program, the IO.PTO bit remains set and another INPUT should be done. If the output buffer of the TTY is exhausted by the INPUT UUO, the IO.PTO bit is cleared.

5.9.4.3 RELEASE UUO - The RELEASE UUO causes the following special actions:

a. Any characters in the output buffer of TTY are discarded.

b. If the controlled job is still attached to TTY, it is detached.

c. The PTY is disassociated from the software channel.

CAUTION

Haphazard use of the PTY and subsequent RELEASE operations may leave detached jobs tying up core and other system resources.

5.9.4.4 JOBSTS UUO - This UUO provides status information about devices TTY and/or the controlled job in order to allow complete and accurate checking of a controlled job.

The call is:

```
MOVEI AC, user channel number      ;or MOVNI AC, job number
JOBSTS AC,                         ;or CALLI AC, 61
error return
normal return
```

When the UUO is called, AC contains a number n specifying the job and/or the TTY to be checked. If n is from 0 to 17, the specified TTY and job are those currently INITed on the user's channel n. If n is negative, the job to be checked is job number (-n).

The error return is given if one of the following is true:

a. the UUO is not implemented. If this is the case, check the I/O status word.

b. n is out of range.

c. there is no PTY INITed on channel n.

Otherwise the normal return is given and AC contains the following status information:

Name	Bit	Explanation
JB.UJA	Bit 0 = 1	Job number is assigned.
JB.ULI	Bit 1 = 1	Job is logged in.
JB.UML	Bit 2 = 1	TTY is at monitor level.
JB.UOA	Bit 3 = 1	TTY output is available.
JB.UDI	Bit 4 = 1	TTY is at user level and in input wait, or TTY is at monitor level and can accept a command. In other words, there is no command awaiting decoding or being delayed, the job is not running, and the job is not stopped waiting for operator device action.
JB.UJC	Bit 5 = 1	JACCT is set. In particular, ↑C↑C will not work.
	Bits 6-17	Reserved for the future.
JB.UJN	Bits 18-35	Job number being checked or 0 if no job number is assigned.

5.9.4.5 CTLJOB UUO - This UUO is used to determine the job number of the program (job) that is controlling the specified job, if any.

The call is:

```
MOVE AC, job number      ;-1 means user's job
CTLJOB AC,               ;or CALLI AC, 65
error return
normal return
```

On a normal return, AC contains the job number of the program (job) that is controlling the controlled job. If AC = -1, the specified job is not being controlled via a PTY.

An error return is given if the UUO is not implemented or the job number is too large.

5.10 TERMINALS

The device mnemonic is TTY0, TTY1, ..., TTY176, TTY177, CTY; the buffer size is 23_8 (20_8 data) words.

Line number n of the Data Line Scanner DC10, PDP-8 680 System, or PDP-8/I DC68A System is referred to as TTYn. The console terminal is CTY. The DECsystem-10 monitor automatically gives the logical name TTY to the user's terminal when a job is initialized.

Terminal device names are assigned dynamically. For interconsole communication by program, one of the two users must type DEASSIGN TTY to make the terminal available to the other user's program as an I/O device. Typing ASSIGN TTYn is the only way to reassign a terminal that has been deassigned.

In a full-duplex terminal service, the two functions of a console, typein and typeout, are handled independently, and do not need to be handled in the strict sense of output first and then input. For example: if two operations are desired from PIP, the request for the second operation can be typed before receiving the asterisk after completion of the first. To stop unwanted output, a Control O is typed. Also, the command Control C does not stop a program instantly; the Control C will be delayed until the program requests input from the keyboard, and then the program will be stopped. When a program must be stopped instantly, as when it gets into a loop, Control C typed twice stops the program.

If, during output operations on a half-duplex terminal (not a local copy terminal), an echo-check failure occurs (i.e., the received character was not the same as the transmitted character), the I/O routine suspends output until the user types the next character. If that character is ↑C, the terminal is immediately placed in monitor mode. If it is ↑O, all TTY output buffers that are currently full are ignored, thus cutting the output short. All other characters cause the service routine to continue output. The user may cause a deliberate echo check by typing in while typeout is in progress. For example, to return to monitor control mode while typeout is in progress, the user must type any character ("X", for example) until an echo check occurs and output is suspended; then he types ↑C.

Programs waiting for TTY output are awakened ten characters before the output buffer is empty, causing them to be swapped in sooner and preventing pauses in typing. Programs waiting for TTY input will be awakened ten characters before the input buffer is filled, thus reducing the possibility of lost typein.

5.10.1 Data Modes

5.10.1.1 ASCII, Octal Code 0 and ASCII Line, Octal Code 1 - The input handling of all control characters is as follows. Characters with ASCII codes of 000 to 037 echo as ↑x and are passed to the program as a control character unless noted otherwise.

000	NULL		Ignored on input; suppressed on output.
001	↑A		No special action.
002	↑B		No special action.
003	↑C		Not passed to program. The user's terminal is switched to monitor mode the next time input is requested by the program. Two successive ↑Cs cause the terminal to be immediately switched to monitor mode. Performs a ↑U and a ↑O. For user program control of ↑C, refer to Paragraph 3.1.3.2.
004	↑D	(EOT)	Not echoed; therefore typing in a control-D (EOT) does not cause a full-duplex data phone to hang up.
005	↑E	(WRU)	No special action.
006	↑F		No special action.
007	↑G	(Bell)	Echoes as Bell and is a break character.
010	↑H	(Backspace)	Echoes as backspace.
011	↑I	(TAB)	Echoes as a TAB or an equivalent number of spaces. Refer to the SET TTY TAB command.
012	↑J	(Linefeed)	Echoes as a linefeed and is a break character.
013	↑K	(Vertical tab)	Echoes as a vertical tab or 4 linefeeds. Refer to the SET TTY FORM command.
014	↑L	(Form)	Echoes as a FORMFEED or 8 linefeeds. Refer to the SET TTY FORM command.
015	↑M	(Carriage return)	Passed to program if terminal is in a paper-tape input mode; otherwise, supplies a linefeed echo, is passed to program as a CR and LF, and is a break character due to the LF.
016	↑N		No special action.
017	↑O		Not passed to program. Complements output suppression bit allowing users to turn output on or off. INPUT, INIT, and OPEN clear the output suppression bit. This bit is also cleared by any other INPUT-class operation, such as DDTIN and TTCALLS 0, 2, 4, and 5, by input test TTCALLS 13 and 14, and by returning to monitor command level via ↑C or EXIT operations. Echoed as ↑O followed by carriage return/linefeed.
020	↑P		No special action.
021	↑Q	(XON)	Starts paper-tape mode if .TTY TAPE command has been given; refer to Paragraphs 5.10.8 and 5.10.9.
022	↑R	(TAPE)	No special action.
023	↑S	(XOFF)	Ends paper-tape mode; refer to Paragraphs 5.10.8 and 5.10.9.

024	↑T	(NO TAPE)	No special action.
025	↑U		Deletes input line back to last break character. Echoed as ↑U followed by a carriage return/linefeed; is a break character. Passed to program if full character-set mode is true.
026	↑V		No special action.
027	↑W		No special action.
030	↑X		No special action.
031	↑Y		No special action.
032	↑Z		Acts as EOF on TTY input. Echoes as ↑Z followed by carriage return/linefeed. Is a break character.
033	↑[	(ESC)	The standard ASCII escape. Echoed as $; is a break character.
034	↑\		No special action.
035	↑]		No special action.
036	↑↑		No special action.
037	↑←		No special action.
040-137			Printing characters, no special action.
140-174			Lower case ASCII; translated to upper case, unless lower case mode is on. Echoes as upper case if translated to upper case.
175 and 176			Old versions of altmode; converted to the standard escape (033) unless in full character set mode INIT or TRMOP. UUO) or no ALTmode conversion is specified (TRMOP. UUO or SET TTY NO ALT command).
177			RUBOUT or DELETE: a. Completely ignored if in paper-tape mode (XON). b. Break character, passed to program if either DDT mode or full character-set mode is true. c. Otherwise (ordinary case) causes a character to be deleted for each rubout typed. All the characters deleted are echoed between a single pair of backslashes. If no characters remain to be deleted, echoes as a carriage return/linefeed.

On output, all characters are typed just as they appear in the output buffer with the exception of TAB, VT, and FORM, which are processed the same as on type-in. Programs should avoid sending ↑D, because it may have catastrophic effects (e.g., it may hang up certain data sets).

5.10.1.2 Image, Octal Code 10 - Image mode is legal for TTY input and output, except for terminals controlled by pseudo-TTYs (refer to Paragraph 5.9). Note that the terminal to be INITed in image mode must be ASSIGNed to a job. An attempt to do input to an unassigned terminal receives an error return with the IO.IMP bit set in the file status word. Image mode is available only in the 5.02 monitor and later.

Because, on input, any sequence of input characters must be allowed, ↑C and ↑Z may not cause their usual escape functions. This means that if the user program accepts all characters and does not release the terminal from image mode, no typein will release the user from this state; consequently, the terminal would effectively become dead to the system. The break character cannot be used to escape from this situation, because DC10 and the 630 do not detect the break character. To solve this design problem, an image input state is defined. If during the image input state, no characters are received for 10 seconds the end-of-file is forced. After another 10 seconds, the image input state is terminated by SCNSER (scanner service) and a ↑C is simulated. Therefore, if the user discovers that his program has failed because of this condition, he simply stops typing until a ↑C appears.

The image input state begins when the program goes into I/O wait because of an INPUT UUO in image mode. It ends when the program executes any non-image terminal output operation. If no output is desired, the TTCALL UUO can be executed to output a null string.

When using image mode input to read binary tapes, echoing should be suppressed by setting bit 28 in the TTY status word.

NOTE

> Because there are no break characters in image mode, characters are transferred a character at a time instead of a line at a time. Therefore, an input buffer may only have one character in it when control is returned to the user program.

On output, the low-order eight bits of each word in the user's buffer are output. These characters are transmitted exactly as supplied by the user. Parity is neither checked nor added, and filler characters are not generated. Image mode affects buffered output (INIT, OUTPUT UUOs) only, except for one TTCALL function (refer to Paragraph 5.10.3).

5.10.2 DDT Submode[1]

To allow a user's program using buffered I/O and the DDT debugging program to use the same terminal without interferring with one another, the TTY service routine provides the DDT submode. This mode does not affect the TTY status if it is initialized with the INIT operator. It is not necessary to use INIT to perform I/O in the DDT submode. I/O in DDT mode is always to the user's terminal and not to any other device.

In the DDT submode, the user's program is responsible for its own buffering. Input is usually one character at a time, but if the typist types characters faster than they are processed, the TTY service routine supplies buffers full of characters at the same time.

[1] The usage described in this section is obsolete; new programs should use the TTCALL UUO (refer to Paragraph 5.10.3).

To input characters in DDT mode, use the sequence

```
MOVEI AC,BUF
CALL AC, [SIXBIT/DDTIN/]
```

BUF is the first address of a 21-word block in the user's area. The DDTIN operator delays, if necessary, until one character is typed in. Then all characters (in 7-bit packed format) typed in since the previous occurrence of DDTIN are moved to the user's area in locations BUF, BUF+1. The character string is always terminated by a null character (000). RUBOUTs are not processed by the service routine but are passed on to the user. The special control characters ↑O and ↑U have no effect. Other characters are processed as in ASCII mode.

To perform output in DDT mode, use the sequence

```
MOVEI AC,BUF
CALL AC,[SIXBIT /DDTOUT/]
```

BUF is the first address of a string of packed 7-bit characters terminated by a null (000) character. The TTY service routine delays until the previous DDTOUT operation is complete, then moves the entire character string into the monitor, begins outputting the string, and restarts the user's program. Character processing is the same as for ASCII mode output.

5.10.3 Special Programmed Operator Service

The TTCALL UUO is used to extend the capabilities of the terminal. The TTCALL operations are performed for a physical terminal (not a logical name TTY) and most operations reference the terminal controlling the job which executed the UUO. (There are exceptions, such as in the case of GETLCH.)

The general form of the TTCALL (operation code 051) programmed operator is as follows:

TTCALL AC, ADR

The AC field describes the particular function desired, and the argument (if any) is contained in ADR. ADR may be an AC or any address in the low segment above the job data area (137). It may be in high segment for AC fields 1 and 3. The functions are:

AC Field	Mnemonic[†]	Action
0	INCHRW	Input character and wait
1	OUTCHR	Output a character
2	INCHRS	Input character and skip
3	OUTSTR	Output a string
4	INCHWL	Input character, wait, line mode
5	INCHSL	Input character, skip, line mode
6	GETLCH	Get line characteristics
7	SETLCH	Set line characteristics
10	RESCAN	Reset input stream to command
11	CLRBFI	Clear type-in buffer
12	CLRBFO	Clear type-out buffer
13	SKPINC	Skip if a character can be input
14	SKPINL	Skip if a line can be input
15	IONEOU	Output as an image character
16-17		(Reserved for expansion)

†The TTCALL mnemonics are defined in a separate MACRO assembler table, which is scanned if an undefined OP CODE is found. If the symbol is found in the TTCALL table, it is defined as though it had appeared in an appropriate OPDEF statement, for example:

TYPE: OUTCHR CHARAC

If OUTCHR is undefined, it will be assembled as though the program contained the statement:

OPDEF OUTCHR TTCALL 1,

This facility is available in MACRO V.44 and later.

INPUT and INPUT TEST operations (TTCALLs 0, 2, 4, 5, 13 and 14) also clear the effect of the previous ↑O type in.

5.10.3.1 INCHRW ADR or TTCALL 0, ADR - This command inputs a character into the low-order seven bits of location ADR. If there is no character yet typed, the program waits.

5.10.3.2 OUTCHR ADR or TTCALL 1, ADR - This command outputs to the user's terminal the character in location ADR. Only the low order 7 bits of the contents of ADR are used. The remaining bits do not need to be zeroes.

If there is no room in the output buffer, the program waits until room is available. ADR may be in high segment.

5.10.3.3 INCHRS ADR or TTCALL 2, ADR - This command is similar to INCHRW, except that it skips on a successful return, and does not skip if there is no character in the input buffer; it never puts the job into a wait.

```
TTCALL 2,ADR
 JRST  NONE
JRST   DONE
```

5.10.3.4 OUTSTR ADR or TTCALL 3, ADR - This command outputs a string of characters in ASCIZ format:

```
TTCALL 3,MESSAGE
MESSAGE: ASCIZ /TYPE THIS OUT/
```

ADR may be in high segment.

5.10.3.5 INCHWL ADR or TTCALL 4, ADR - This command is the same as INCHRW, except that it decides whether or not to wait on the basis of lines rather than characters; as such, it is the preferred way of inputting characters, because INCHRW causes a swap to occur for each character rather than each line (compare DDT and PIP input). In other words, INCHWL returns the next character in the line if a break character has been typed.[1] If a break character has not been typed, INCHWL waits. Repeated uses of INCHWL return each of the successive characters of the line.

Note that a control-C character in the input buffer is sufficient to satisfy the condition of a pending line. Therefore, when the input is done, the control-C is interpreted and the job is stopped. This definition of a line also applies to TTCALL 5, and TTCALL 14,.

5.10.3.6 INCHSL ADR or TTCALL 5, ADR - This command is the same as INCHRS, except that its decision whether to skip is made on the basis of lines rather than characters.

5.10.3.7 GETLCH ADR or TTCALL 6, ADR - This command takes one argument, from location ADR, and returns one word, also in ADR. The argument is a number, representing a TTY line. Bits 18 and 19 of the line number are ignored since terminal numbers begin at 200000. If the argument is negative, the line number controlling the program is assumed. If the line number is greater than those defined in the system, a zero answer is returned.

---

[1] If the input buffer becomes nearly filled, the waiting-of-line condition is satisfied even though no break character appears. This is true of all line-mode input operations.

The normal answer format is as follows:

Name	Bit	Meaning
GL.ITY	0	Line is a pseudo TTY.
GL.CTY	1	Line is the CTY.
GL.DSP	2	Line is the display console.
GL.DSL	3	Line is the dataset data line.
	4	Obsolete.
GL.HDP	5	Line is half-duplex.
GL.REM	6	Line is a remote TTY.
GL.RBS	7	Line is at a remote batch station.
GL.LIN	11	A line has been typed in by the user.
	12	Obsolete.
GL.LCM	13	Lower case input mode is on.
GL.TAB	14	Terminal has tabs.
GL.LCP	15	Terminal input is not echoed, because device is local copy.
GL.PTM	16	Control Q (paper-tape) switch is on.
	17	Obsolete.
	18-35	200000 + line number.

5.10.3.8 SETLCH ADR or TTCALL 7, ADR - This command allows a program to set and clear some of the bits for GETLCH. They may be changed only for the job's controlling TTY. Bits 13, 14, 15, and 16 can be modified. Bits 18 and 19 of the line number argument are ignored.

Example:

```
SETO     AC,0
GETLCH   AC
TLZ      AC,BIT 13
TLO      AC,BIT 14
SETLCH   AC
```

5.10.3.9 RESCAN or TTCALL 10, 0 - This command is intended for use only by the COMPIL program. It causes the input buffer to be rescanned from the point where the last command began. If bit 35 of E is 1, the error return is given if there is a command in the input buffer. If the input buffer is empty, the skip return is given. Obviously, if the UUO is executed other than before the first input, that command may no longer be in the buffer. ADR is not used, but it is address checked.

5.10.3.10 CLRBFI or TTCALL 11, 0 - This command causes the input buffer to be cleared as if the user had typed a number of CONTROL Us. It is intended to be used when an error has been detected (e.g., if a user did not want any commands that he might have typed ahead to be executed).

5.10.3.11 CLRBFO or TTCALL 12, 0 - This command causes the output buffer to be cleared as if the user had typed CONTROL O. It should be used rarely, because usually one wants to see all output, up to the point of an error. This command is included primarily for completeness.

5.10.3.12 SKPINC or TTCALL 13, 0 - This command skips if the user has typed at least one character. It does not skip if no characters have been typed; however, it never inputs a character. It is useful for a computer-based program that wants to occasionally check for input and, if any, go off to another routine (such as FORTRAN operating system) to actually do the input.

5.10.3.13 SKPINL or TTCALL 14, 0 - This command is the same as SKPINC, except that a skip occurs if the user has typed at least one line.

5.10.3.14 IONEOU ADR or TTCALL 15, E - This command outputs the low-order eight bits of the contents of E as an image character to the terminal.

5.10.4 GETLIN AC, or CALLI AC, 34 - This UUO returns the SIXBIT physical name of the terminal that the job is attached to.

The call is:

```
        GETLIN AC,          ;OR CALLI AC, 34
```

The name is returned left justified in the AC. If the job issuing the UUO is currently detached, the left half of AC contains a 0 on return. The right half of AC contains the right half of the physical name of the terminal to which the job was most recently attached. Therefore, by testing the left half of AC, jobs can determine if they are attached to a terminal.

Example:

```
        CTY or TTY3 or TTY30
```

This UUO is used by the LOGIN program to print the TTY name.

5.10.5 TRMNO. AC, or CALLI AC, 115[1]

This UUO is used to obtain the number of the terminal currently controlling a particular job. This terminal number can then be used as the argument to the GETLCH (refer to Paragraph 5.10.3.7) and TRMOP. (refer to Paragraph 5.10.6) UUOs.

---

[1]This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

The call is:

```
MOVE AC, job number
TRMNO. AC,                  ;or CALLI AC, 115
error return
normal return
```

On a normal return, the right half of AC contains the universal I/O index (.UXxxx) for the terminal. The range of values is 200000 to 200777 octal. The symbol .UXTRM (octal value 200000) is the offset for the terminal indices.

On an error return, if the AC is unchanged, the UUO is not implemented. If the AC contains zero, one of three errors occurred:

1) The job is currently detached and therefore, no terminal is controlling it.
2) The job number is unassigned; i.e., there is no such job.
3) The job number is out of range and therefore illegal.

The particular error condition can be determined from the JOBSTS UUO (refer to Paragraph 5.9.4.4). For example,

```
MOVEI AC, number
TRMNO. AC,
   JRST .+2
JRST OK
JUMPN AC, not implemented
MOVNI AC, number
JOBSTS AC,
   JRST illegal number
JUMPL AC, detached
JRST no job assigned.
```

5.10.6 TRMOP. AC, or CALLI AC, 116[1]

This UUO allows the user to control, examine, and modify information about any terminal connected to the system. Many of the functions of this UUO are extensions to the TTCALL input and output functions (refer to Paragraph 5.10.3). Certain functions are privileged, or require that the user have the terminal ASSIGNed. Generally, any function is legal for the terminal on which the job issuing the UUO is running. In addition, any READ or SKIP function is legal for any terminal if the job issuing the UUO 1) has the privilege bit JP.SPM set, 2) is running with the JACCT bit set, or 3) is logged in as [1,2]. A SET or output function is legal for any terminal if the job 1) has the privilege bit JP.POK set, 2) is running with the JACCT bit set, or 3) is logged-in as [1,2].

---

[1] This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

The call is:

```
        MOVE AC, [XWD N, ADR]
        TRMOP. AC,                  ;or CALLI AC, 116
        error return
        normal return
ADR:    function code
ADR+1:  universal I/O index
```

ADR is the address of the argument block and N is the length (N must be at least 2). The first word of the argument block contains the code for the requested function. The second word contains the universal I/O index of the terminal to be affected (.UXTRM + line number). This index is in the same format as returned by the TRMNO. UUO (refer to Paragraph 5.10.5). Remaining arguments in the argument block depend on the particular function used.

Function codes are defined within the following ranges:

0000-0777	Perform a specific action.
1000-1777	Read a parameter.
2000-2777	Set a parameter.
3000-3777	Reserved for DEC customers.

The functions within the range 0000-0777 are as follows:

.TOSIP	1	Skip if terminal input buffer is not empty.
.TOSOP	2	Skip if terminal output buffer is not empty.
.TOCIB	3	Clear terminal input buffer.
.TOCOB	4	Clear terminal output buffer.
.TOOUC	5	Output character to terminal from ADR+2 (not yet implemented).
.TOOIC	6	Output image mode (8-bit) character from ADR+2 (not yet implemented).
.TOOUS	7	Output ASCIZ string to terminal from address at ADR+2 (not yet implemented).
.TOINC	10	Input character from terminal to AC, normal mode (not yet implemented).
.TOIIC	11	Input character from terminal to AC, image mode (not yet implemented).
.TODSE	12	Enable modem for outgoing call.
.TODSC	13	Enable and place outgoing call on modem with dialer. Phone number of up to 17 digits is stored in 4-bit bytes in ADR+2 and ADR+3 and is terminated by a 17 byte. If caller must wait for a second dial tone (e.g., after dialing a 9), a 16 byte results in a 5 second wait.
.TODSF	14	Hang up modem (i.e., disconnect call).

The READ (1000-1777) and SET (2000-2777) functions are parallel; i.e., if function 1002 reads a particular parameter, then function 2002 sets the same parameter. Values for the READ functions are returned in AC; arguments to the SET functions are given in ADR+2. One-bit quantities are not range-checked; instead bit 35 of ADR+2 is stored. The following description of the READ function codes indicate if there is a corresponding SET function code.

Read Code	Range	Description	Corresponding SET
1000	1 bit	Output in progress (.TOOIP)	No
1001	1 bit	Terminal at monitor mode (.TOCOM)	No
1002	1 bit	Paper tape mode (.TOXON)	Yes
1003	1 bit	Lower case (if set, no lower case) (.TOLCT)	Yes
1004	1 bit	Slave switch (.TOSLV)	Yes
1005	1 bit	Tab switch (if 0 = spaces, if 1 = tab) (.TOTAB)	Yes
1006	1 bit	Form switch (if 0 = linefeeds, if 1 = formfeeds) (.TOFRM)	Yes
1007	1 bit	Local copy switch (if set, no echo) (.TOLCP)	Yes
1010	1 bit	Free CR-LF switch (if set, no CR-LF) (.TONFC)	Yes
1011	0 to 377	Horizontal position of carriage (.TOHPS)	No
1012	16. to 200.	Carriage width (.TOWID)	Yes
1013	1 bit	TTY GAG bit (if set, NO GAG) (.TOSND)	Yes
1014	1 bit	Half-duplex line (.TOHLF)	Yes, privileged
1015	1 bit	Remote line (.TORMT)	Yes, privileged
1016	1 bit	Display terminal (.TODIS)	Yes, privileged
1017	0 to 3	Filler class (.TOFLC)	Yes
1020	1 bit	Paper tape enabled (.TOTAP)	Yes
1021	1 bit	Paged display mode (also set and cleared by SET TTY PAGE)(.TOPAG)	Yes
1022	1 bit	Suspended output (need XON to resume) (also set by XOFF, formfeed, or page size exceeded, if paged display mode)(.TOSTP) Not implemented.	Yes
1023	0 to 63.	Page size (number of lines) (also set by SET TTY PAGE)(.TOPSZ) Not implemented.	Yes
1024	0 to 63.	Page counter (number of lines output this page)(.TOPCT)	Yes
1025	1 bit	Suppress blank lines on output (0 = normal output and 1 = suppress multiple linefeeds) and convert formfeeds and vertical tabs to linefeeds (also set and cleared by SET TTY BLANK)(.TOBLK)	Yes

Read Code	Range	Description	Corresponding SET
1026	1 bit	Suppress ALTmode conversion on input (0 = 175 and 176 converted to 033 and 1 = no conversion) (also set and cleared by SET TTY ALT)(.TOALT)	Yes

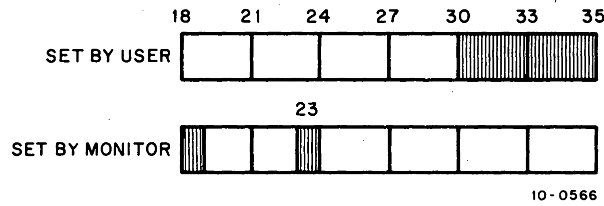
On an error return, AC is either unchanged or contains an error code.

AC	Name	Meaning
unchanged		UUO is not implemented.
0		The requested function is not implemented.
1	TOPRC%	User is not privileged to perform this function.
2	TORGB%	Argument is out of range.
3	TOADB%	Argument list length or address is illegal.
4	TOIMP%	Dataset activity to a non-dataset terminal.
5	TODIL%	Subfunction failed (e.g., call not properly completed from dialer).

5.10.7 File Status (Refer to Appendix D)

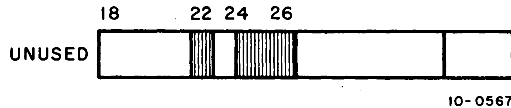
The file status of the terminal is shown below.

Standard Bits

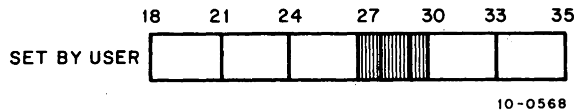


Bit 18 - IO.IMP — TTY is not assigned to a job (for image mode input processing).

Bit 23 - IO.ACT — Device is active.



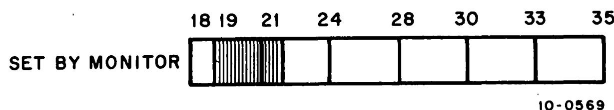
Device Dependent Bits



Bit 27 - IO.TEC — This bit causes 001 through 037, 175, and 176 (octal) to echo the character exactly as received by the monitor. There is no special echo (e.g., $ or ↑ x).

Bit 28 - IO.SUP — Suppresses echoing on the terminal.

Bit 29 - IO.FCS — Full character set. Pass all characters except lower case and ↑ C. Lower case is controlled by the SET TTY LC command and its corresponding TRMOP. UUO function.



Bit 19 - IO.DER Ignore interrupts for three-fourths of a second.

Bit 20 - IO.DTE Echo failure has occurred on output.

Bit 21 - IO.BKT Character was lost on typein.

5.10.8 Paper-Tape Input from the Terminal (Full-Duplex Software)

Paper-tape input is possible from a terminal equipped with a paper-tape reader that is controlled by the XON (↑Q) and XOFF (↑S) characters. When commanded by the XON character, the terminal service reads paper tapes, starting and stopping the paper tape as needed, and continuing until the XOFF character is read or typed in. While in this mode of operation, any RUBOUTS will be discarded and no free line feeds will be inserted after carriage returns. Also, TABS and FORMFEEDS will not be simulated on a Teletype Model 33 to ensure output of the reader control characters. To use paper tape processing, the terminal with a paper-tape reader must be connected by a full-duplex connection and only ASCII paper tapes should be used.

The correct operating sequence for reading a paper tape in this way is as follows:

```
.R PIP↵
*DSK:FILE←TTY:↑Q↵
THIS IS WHAT IS ON TAPE
MORE OF THE SAME
LAST LINE ↑Z
*↑C
```

5.10.9 Paper-Tape Output at the Terminal (Full-Duplex Software)

Paper-tape output is possible on any terminal-mounted paper-tape punch, which is controlled by the TAPE, AUX ON (↑R) and ~~TAPE,~~ AUX OFF (↑T) characters. The punch is connected in parallel with the keyboard printer, and therefore, when the punch is on, all characters on the keyboard are punched on tape.

LT33B or LT33H Teletypes can have the reader and punch turned off and on under program control. When commanded by the AUX ON character, the TTY service punches paper tapes until the AUX OFF character is read or typed in. The AUX OFF character is the last character punched on tape.

When writing programs to output to the terminal paper-tape punch, the user should punch several inches of blank tape before the AUX OFF character is transmitted. This last character may then be torn off and discarded.

CHAPTER 6
I/O PROGRAMMING FOR DIRECTORY DEVICES

This chapter explains the unique features of the standard directory devices. Each device accepts the programmed operators explained in Chapter 4, unless otherwise indicated. Table 6-1 is a summary of the characteristics of the directory devices. Buffer sizes are given in octal and include three book-keeping words. The user may determine the physical characteristics associated with a logical device name by calling the DEVCHR UUO (refer to Paragraph 4.10.2).

Table 6-1
Directory Devices

Device	Physical Name	Controller Number	Unit Number	Programmed Operators	Data Modes	Buffer Sizes (Octal †)
DECtape	DTA0, DTA1, ...,DTA7 DTB0, DTB1, ...,DTB7††	TD10 551(PDP-6)	TU55 555(PDP-6)	INPUT, IN OUTPUT, OUT LOOKUP, ENTER MTAPE, USETF, USETO, USETI UTPCLR	A,AL,I B,IB DR,D	202
Fixed-Head Disk	DSK, FHA, FHA0, ..., FHA3	RC10	RD10 RM10B	INPUT, IN OUTPUT, OUT LOOKUP, ENTER RENAME, SEEK USETO, USETI	A,AL,I B,IB DR,D	203
Disk Pack	DSK, DPA, DPA0, ..., DPA7	RP10	RP01 RP02	INPUT, IN OUTPUT, OUT LOOKUP, ENTER RENAME, SEEK USETO, USETI	A,AL,I B,IB DR,D	203

†Buffer sizes are subject to change and should be calculated rather than assumed by user programs. A DEVSIZ UUO may be employed.
††Recognized if dual DECtape controller is supported.

6.1 DECTAPE

The device mnemonic is DTA0, DTA1, ..., DTA7; the buffer size is 202_8 words (177_8 user data, 200_8 transferred). On systems with dual DECtape controllers, the drives on the second controller have the mnemonic DTB0, DTB1, ..., DTB7.

6.1.1 Data Modes

Two hundred words are written. The first word is the link plus word count. The following 177_8 words are data supplied to and from user programs.

6.1.1.1 Buffered Data Modes - Data is written on DECtape exactly as it appears in the buffer and consists of 36-bit words. No processing or checksumming of any kind is performed by the service routine. The self-checking of the DECtape system is sufficient assurance that the data is correct. Refer to Paragraph 6.1.2 for further information concerning blocking of information.

6.1.1.2 Unbuffered Data Modes - Data is read into or written from anywhere in the user's core area without regard to the standard buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is described in Chapter 4. On the KI10, if the IOWD list is modified as the result of I/O performed (i.e., an INPUT UUO reads into the IOWD list) and the word count of any of the IOWDs read into the list is greater than the following value:

(maximum word count specified in original list-2)/512 + 2

then the job is stopped and the monitor types

ADDRESS CHECK AT USER adr

File-structured dump mode data is automatically blocked into standard-length DECtape blocks by the DECtape service routine. Each block read or written contains 1 link word plus 1 to 177_8 data words. Unless the number of data words is an exact multiple of the data portion of a DECtape block (177_8), the remainder of the last block written after each output programmed operator is wasted. The input programmed operator must specify the same number of words that the corresponding output programmed operator specified to skip over the wasted fractions of blocks.

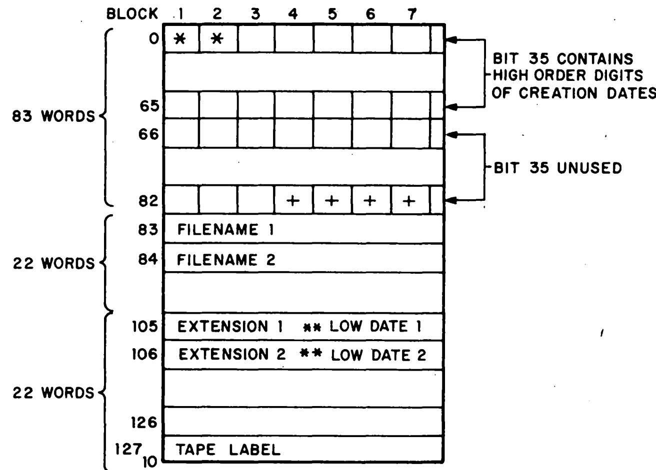
6.1.2 DECtape Format

A standard reel of DECtape consists of 578 (1102_8) prerecorded blocks each capable of storing 128 (200_8) 36-bit words of data. Block numbers that label the blocks for addressing purposes are recorded between blocks. These block numbers run from 0 to 1101_8. Blocks 0, 1, and 2 are normally not used during timesharing and are reserved for a bootstrap loader. Block 100_{10} (144_8) is the directory block, which contains the names of all files on the tape and information relating to each file. Blocks 3_{10} through 99_{10} (1-143_8) and 101_{10} through 577_{10} (145-1101_8) are usable for data.

If, in the process of DECtape I/O, the I/O service routine is requested to use a block number larger than 1101_8 or smaller than 0, the monitor sets the IO.BKT flag (bit 21) in the file status and returns.

6.1.3 DECtape Directory Format

The directory block (block 100_{10}) of a DECtape contains directory information for all files on that tape; a maximum of 22 files can be stored on any one DECtape (see Figure 6-1).



NOTES:

* Reserved for system, contains 36 as does block 144_8 for the directory.

** For zero-compressed files, this area holds the number of 1K blocks (-1) needed to load the file (up to 64K).

\+ Represents blocks 1102 through 1105, which are not available contains 37_8.

10-0572

Figure 6-1 DECtape Directory Format

The first 83 words (0 through 82_{10}) of the directory block contains slots for blocks 1 through 577 on a DECtape. Each slot occupies five bits (seven slots are stored per word) and represents a given block on the DECtape. Each slot contains the number of the file ($1-26_8$) occupying the given block. This allows for 581 slots (83 words x 7 slots per word). The four extra slots represent nonexistent blocks 1102 through 1105_8.

Bit 35 of the first 66 words (0 through 65_{10}) of the directory block contain the high order 3 bits of the 15-bit creation date of each file on the DECtape. (Note that the low order 12 bits of the creation date of each file are contained in words 105 through 126_{10}. This split format allows for compatibility among monitors and media as old as 1964.) The high order 3 bits of the 15-bit creation date for file 1 are contained in bit 35 of words 0, 22, and 44. Word 44 contains the first (most significant) digit; word

22 contains the second and word 0 contains the third. The high order digits for file 2 are contained in bit 35 of words 1, 23, and 45 with the digits in the same order as described for file 1. The high order digits for the remaining files are organized in the same fashion.

Words 83 through 104_{10} of the directory block contain the filenames of the 22 files that reside on the DECtape. Word 83 contains the filename for file 1, word 84 contains the filename for file 2. Filenames are stored in SIXBIT code.

The next 22 words of the directory block (words 105 through 126_{10}) primarily contain the filename extensions and the low order part of the creation dates of the 22 files that reside on the DECtape, in the same relative order as their filenames. The bits for each word are as follows:

Bits 0 - 17_{10}	The filename extension in SIXBIT code.
Bits 18 - 23_{10}	The number of 1K blocks minus 1 needed to load the file (maximum value is 63). This information is stored for zero-compressed files only.
Bits 24 - 35_{10}	The low order 12 bits of the date on which the file was created. (Note that the high order digits are encoded in bit 35 of words 0 through 65_{10}). The creation date is computed with the following formula: ((year-1964) * 12 + (month-1)) * 31 + day -1

Word 127_{10} of the directory block is the tape label.

The message

BAD DIRECTORY FOR DEVICE DTAn: EXEC CALLED FROM USER LOC n

occurs when any of the following conditions are detected:

a. A parity error occurred while reading the directory block.

b. No slots are assigned to the file number of the file.

c. The tape block, which may be the first block of the file (i.e., the first block for the file encountered while searching backwards from the directory block), cannot be read.

Ordinary user programs never manipulate DECtape directories explicitly since the LOOKUP and ENTER programmed operators (refer to Paragraphs 6.1.5.1 and 6.1.5.2) automatically record all necessary entries in the directory for the user. These programmed operators have all the capability needed to process the name and creation date of a file. However, a small number of special purpose programs do process directories by explicit action rather than using the LOOKUP and ENTER operators. For such programs, the following examples illustrate methods for 1) assembling the 15-bit creation date and 2) storing the 15-bit creation date. The number of the file (an integer from 1 to 22) is in register P1 and the directory block begins at location DIRECT.

Example 1 Special Purpose Assembly of the Creation Date

```
LDB     T1, [POINT 12, DIRECT+↑D104 (P1), 35]   ;GET LOW PART
MOVEI   T2, 1                                   ;SET UP TO TEST LOW BIT
TDNE    T2, DIRECT-1 (P1)                       ;IF SET IN DIRECTORY
TRO     T1, 1B23                                ;THEN SET BIT IN DATE
TDNE    T2, DIRECT+↑D21 (P1)                    ;REPEAT FOR EACH BIT IN
TRO     T1, 1B22                                ;HIGH PART OF DATE
TDNE    T2, DIRECT+↑D43 (P1)                    ;
TRO     T1, 1B21                                ;
```

Example 2 Special Purpose Storage of the Creation Date

```
DPB     T1, [POINT 12, DIRECT+↑D104 (P1), 35]   ;SAVE LOW PART
MOVEI   T2, 1                                   ;SET UP TO MARK LOW BIT
ANDCAM  T2, DIRECT-1 (P1)                       ;CLEAR DIRECTORY BIT
TRNE    T1, 1B23                                ;IF BIT IN DATE SET,
IORM    T2, DIRECT-1 (P1)                       ;SET DIRECTORY BIT
ANDCAM  T2, DIRECT+↑D21 (P1)                    ;
TRNE    T1, 1B22                                ;REPEAT FOR EACH BIT IN
IORM    T2, DIRECT+↑D21 (P1)                    ;HIGH PART OF DATE
ANDCAM  T2, DIRECT+↑D43 (P1)                    ;
TRNE    T1, 1B21                                ;
IORM    T2, DIRECT+↑D43 (P1)                    ;
```

6.1.4 DECtape File Format

A file consists of any number of DECtape blocks.

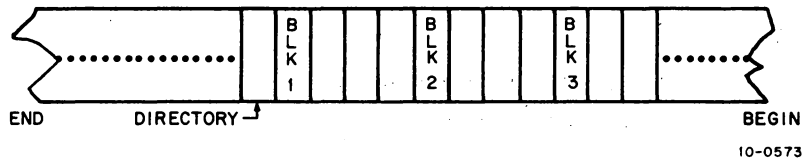


Figure 6-2 Format of a File on Tape

Each block contains the following:

Word 0	Left half	The link. The link is the block number of the next block in the file. If the link is zero, this block is the last in the file.
	Right half	Bits 18 through 27: the block number of the first block of the file. Bits 28 through 35: a count of the number of words in this block that are used (maximum 177_8).
Words 1 through 177_8		Data packed exactly as the user placed in his buffer or in dump mode files, the next 177 words of memory.

LINK	FIRST BLOCK NUMBER	WORD COUNT
DATA		

10-0574

Figure 6-3 Format of a DECtape Block

6.1.4.1 Block Allocation - Normally, blocks are allocated by starting with the first free block nearest the directory and going backwards to the front of the tape (block 0). When the end of the tape is reached, the direction of the scan is reversed. Blocks are not written contiguously; rather they are separated by a spacing factor. This allows the drive to stop and restart to read the next block of the file without having to back up the tape. The spacing factor is normally four, but for dump mode and UGETF followed by an ENTER, the spacing factor is two (refer to Paragraph 6.1.6.3).

6.1.5 I/O Programming

DECtape is a directory device; therefore, file selection must be performed by the user before data is transferred. File selection is accomplished with LOOKUP and ENTER UUOs. The UUO format is as follows:

UUO D, E

where D specifies the user channel associated with this device, and E points to a four-word parameter block. The parameter block has the following format:

E	FILE			
E+1	EXT	HIGH DATE	0	BLOCK #
E+2	0	# OF 1K BLOCKS		LOW DATE
E+3	-N	ADR-1		

10-0575

(continued on next page)

where

FILE is the filename in SIXBIT ASCII.

EXT is the filename extension in SIXBIT ASCII.

HIGH DATE contains the high order 3 bits of the creation date.

BLOCK # is the number of the first block of the file.

# of 1K blocks is the number of blocks needed to load the file if the file is a zero-compressed file (bits 18-23).

LOW DATE contains the low order 12 bits of the date on which the file was originally created (bits 24-35). The format is the same as that used by the DATE UUO.

-N is the negative word length of the zero-compressed file.

ADR-1 is the core address of the first word of the file minus 1.

Location E + 3 is used for zero-compressed files.

6.1.5.1 LOOKUP D, E - The LOOKUP programmed operator sets up an input file on channel D. The contents of location E and E + 1 (left half) are matched against the filenames and filename extensions in the DECtape directory. If no match is found, the error return is taken (refer to Appendix E). If a match is found, locations E + 1 through E + 3 are filled by the monitor, and the normal return is taken (refer to Table 6-2). Refer to Section d. of Paragraph 6.2.8.1 for sample code of assembling the 15-bit creation date.

Table 6-2
LOOKUP Parameters

On Call			On Return		
Parameter	Use†	Contents	Parameter	Use†	Contents
E	A	SIXBIT /FILE/	E	V	SIXBIT /FILE/
E + 1	A	SIXBIT /EXT/	E + 1	V	LH = SIXBIT /EXT/ RH = high order 3 bits of 15-bit creation date (Bits 18-20) unused (Bits 21-25) first block # (Bits 26-35)
E + 2	I	-	E + 2	V	LH = 0 RH = # of 1K blocks (Bits 18-23) †† low order 12 bits of 15-bit creation date (Bits 24-35)††
E + 3	I	-	E + 3	V	IOWD LENGTH, ADR††

†A = argument from user program, V = value from monitor, I = ignored.

††For zero-compressed files only.

The first block of the file is then found as follows:

a. The first 83 words of the DECtape directory are searched backwards, beginning with the slot immediately prior to the directory block, until the slot containing the desired file number is found.

b. The block associated with this slot is read in and bits 18 through 27 of the first word of the block (these bits contain the block number of the first block of the file) are checked. If the bits are equal to the block number of this block, then this block is the first block; if not, then the block with that block number is read as the first block of the file.

6.1.5.2 ENTER D, E - The ENTER programmed operator sets up an output file on channel D. The DECtape directory is searched for a filename and filename extension that match the contents of location E and the left half of location E + 1. If no match is found and there is room in the directory, the monitor records the information in locations E through E + 2 in the DECtape directory (refer to Table 6-3). An error return is given if there is no room in the directory for the file (refer to Appendix E). Refer to Paragraph 6.2.8.3 for a special note on error recovery. If a match is found, the new entry replaces the old entry, the old file is reclaimed immediately, and the monitor records the file information. This process is called superseding and differs from the process on disk in that, because of the small size of DECtape, the space is reclaimed before the file is written rather than after. Refer to Section d. of Paragraph 6.2.8.1 for sample code for setting the 15-bit creation date.

Table 6-3
ENTER Parameters

On Call			On Return		
Parameter	Use †	Contents	Parameter	Use †	Contents
E	A	SIXBIT /FILE/	E	V	SIXBIT /FILE/
E + 1	A	LH = SIXBIT /EXT/ RH = high order 3 bits of 15-bit creation date (bits 18-20).	E + 1	V	LH = SIXBIT /EXT/ RH = high order 3 bits of 15-bit creation date (bits 18-20).
E + 2	A	RH = low order 12 bits of desired 15-bit creation date or 0. (0 implies current date)	E + 2	V	RH = low order 12 bits of 15-bit creation date (bits 24-35).
E + 3	I	-	E + 3	I	-

†A = argument from user program, V = value from monitor, I = ignored.

6.1.5.3 RENAME D, E - The RENAME programmed operator alters the filename or filename extension of an existing file, or deletes the file directory from the DECtape associated with channel D. If location E contains a 0, RENAME deletes the directory of the specified file; otherwise, RENAME searches for the file and enters the information specified in location E and E + 1 into the DECtape directory (refer to Table 6-4). RENAME must be preceded by a LOOKUP or an ENTER, to select the file that is to be RENAMED, and a CLOSE. The error return is given if a LOOKUP has not been done (refer to Appendix E). Refer to Paragraph 6.2.8.3 for a special note on error recovery.

Table 6-4
RENAME Parameters

On Call			On Return		
Parameter	Use †	Contents	Parameter	Use †	Contents
E	A	SIXBIT /FILE/ or 0	E	V	SIXBIT /FILE/
E + 1	A	LH = SIXBIT /EXT/ RH = high order 3 bits of 15-bit creation date (bits 18-20).	E + 1	V	LH = SIXBIT /EXT/ RH = high order 3 bits of 15-bit creation date (bits 18-20).
E + 2	A	RH = low order 12 bits of 15-bit creation date or 0 (0 implies current date).	E + 2	V	RH = low order 12 bits of 15-bit creation date (bits 24-35).
E + 3	I	-	E + 3	I	-

†A = argument from user program, V = value from monitor, I = ignored.

Unlike on disk, a DECtape RENAME works on the last file LOOKUPed and ENTERed for the device, not the last file for this channel. The UUO sequence required to successfully RENAME a file on DECtape is as follows:

```
LOOKUP    D,E
CLOSE D,
RENAME    D,E1

or

ENTER     D,E
CLOSE D,
RENAME    D,E1
```

6.1.5.4 INPUT, OUTPUT, CLOSE, RELEASE - When performing nondump input operations, the DECtape service routine reads the links in each block to determine what block to read next and when to raise the EOF flag.

When an OUTPUT is given, the DECtape service routine examines the left half of the third word in the output buffer (the word containing the word count in the right half). If this half contains -1, it is replaced with a 0 before being written out, and the file is thus terminated. If this half word is greater than 0, it is not changed and the service routine uses it as the block number for the next OUTPUT. If this half word is 0, the DECtape service routine assigns the block number of the next block for the next OUTPUT.

For both INPUT and OUTPUT, block 100 (the directory) is treated as an exception case. If the user's program gives

USETI D, 144_8

to read block 100, it is treated as a 1-block file.

The CLOSE operator places a -1 in the left half of the first word in the last output buffer, thus terminating the file.

The RELEASE operator writes the copy of the directory, which is normally kept in core onto block 100, but only if any changes have been made. Certain console commands, such as KJOB or CORE 0, perform an implicit RELEASE of all devices and, thus, write out a changed directory even though the user's program failed to give a RELEASE.

6.1.6 Special Programmed Operator Service

Several programmed operators are provided for manipulating DECtape. These UUOs allow the user to manipulate block numbers and to handle directories.

6.1.6.1 USETI D, E - The USETI programmed operator sets the DECtape on channel D to input block E next. Since the monitor reads as many buffers as it can on INPUT, it is difficult to determine which buffer the monitor is processing when the USETI is given. Therefore, the INPUT following the USETI may not obtain the buffer containing the block specified. However, if a single buffer ring is used, the desired block is retrieved since the device must stop after each INPUT. Alternatively, if bit 30 (IO.SYN) of the file status word is set via an INIT, OPEN, or SETSTS UUO, the device stops after each bufferful of data on an INPUT so that the USETI will apply to the buffer supplied by the next INPUT.

6.1.6.2 USETO D, E - The USETO programmed operator sets the DECtape on channel D to output block E next. With multiple-buffered I/O, the output following the USETO may not apply to the buffer containing the specified block, since the monitor transfers as many buffers as possible with each OUTPUT. Therefore, a single buffer ring should be used, or bit 30 (IO.SYN) of the file status word should be set. Refer to Paragraph 6.1.6.1.

6.1.6.3 UGETF D, E - The UGETF programmed operator places the number of the next free block of the file in the user's location E.

If UGETF is followed by an ENTER, the monitor modifies its algorithm in the following manner:

1) the first block is written nearest the front of the tape instead of nearest the directory.
2) the spacing factor is changed to 2 instead of 4 so that very large programs can fit almost entirely in a forward direction.

This feature allows user programs, such as PIP, to write SAV format files which can be read by the executive mode utility program TENDMP (see the DECsystem-10 Software Notebooks).

6.1.6.4 UTPCLR AC, or CALLI AC, 13 - The UTPCLR programmed operator clears the directory of the DECtape on the device channel specified in the AC field. A cleared directory has zeroes in the first 83 words except in the slots related to blocks 0, 1, 2, and 100_{10} and nonexistent blocks 1102 through 1105_8. Only the directory block is affected by UTPCLR. This programmed operator is a no-operation if the device on the channel is not a DECtape.

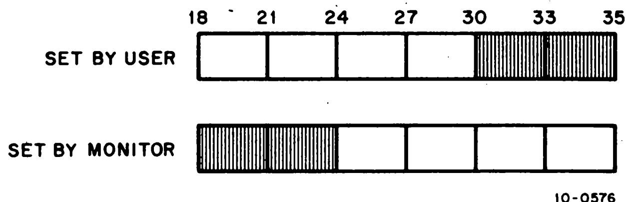
6.1.6.5 MTAPE D, 1 and MTAPE D, 11 - MTAPE D, 1 rewinds the DECtape and moves it into the end zone at the front of the tape. MTAPE D, 11 rewinds and unloads the tape, pulling the tape completely onto the left-hand reel, and clears the directory-in-core bit. These commands affect only the physical position of the tape, not the logical position. When either is used, the user's job can be swapped out while the DECtape is rewinding; however, the job cannot be swapped out if an INPUT or OUTPUT is done while the tape is rewinding.

6.1.6.6 DEVSTS UUO - After each interrupt, the DECtape service routine stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

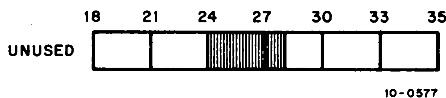
6.1.7 File Status (Refer to Appendix D)

The file status of the DECtape is shown on the next page.

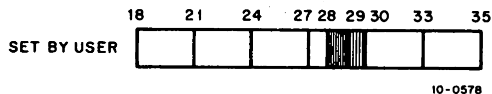
Standard Bits



Bit 18 - IO.IMP	An attempt was made to read block 0 in nonstandard dump mode.
Bit 19 - IO.DER	Data was missed.
Bit 20 - IO.DTE	Parity error.
Bit 21 - IO.BKT	Block number is too large or tape is full on OUTPUT.
Bit 22 - IO.EOF	EOF mark encountered on input. No special character appears in buffer.
Bit 23 - IO.ACT	Device is active.



Device Dependent Bits



Bit 28 - IO.SSD — DECtape is in semi-standard I/O mode. The setting of this bit is recognized only if bit 29 (nonstandard I/O mode) is on. Semi-standard mode is similar to nonstandard mode except, 1) block numbers are checked for legality, and 2) the tape is started in the same direction as it was previously going.

Bit 29 - IO.NSD — DECtape is in a nonstandard-I/O mode format as opposed to standard-I/O mode. No file-structured operations are performed on the tape. Blocks are read or written sequentially; no links are generated (output) or recognized (input). The first block to be read or written must be set by a USETI or USETO. In nonstandard-I/O mode, up to 200_8 words per block are read or written as user data (as opposed to the standard mode of 1 link plus word count followed by 177_8 words). No dead reckoning is used on a search for a block number as the tape may be composed of blocks shorter than 200 words. The ENTER, LOOKUP, and UPTCLR UUOs are treated as no-ops. Block 0 of the tape may not be read or written in dump mode if bit 29 is on, because the data must be read in a forward direction and block 0 normally cannot be read forward.

6.1.8 Important Considerations

When positioning to a desired block on DECtape, the technique of dead reckoning is used. This means that the DECtape service routine starts the DECtape spinning and computes the time it should take to reach the desired block. Meanwhile, the service routine performs a service for another user, if any, and then returns just before the computed time has elapsed. If the desired block has not been reached, this process is repeated until it is successful. This technique is used to keep the controller free for other uses while the DECtape is spinning.

When an attempt is made to write on a write-locked tape or to access a drive that has no tape mounted, the message

DEVICE DTAn OPERATOR zz ACTION REQUESTED

is given to the user. When the situation has been rectified, CONT may be typed to proceed. However, if this message is output because of an attempt to write on a write-locked tape and any operation that causes a RESET to be performed (e.g., a GET or RUN command) is then executed, a RELEASE will be done on the DECtape. This RELEASE causes any attempt to write the directory to output the same message. To avoid the second output of the message, the user should ASSIGN the DECtape again thus causing the DECtape service routine not to write the directory on the RELEASE.

The DECtape service routine reads the directory from a tape the first time it is required to perform a LOOKUP, ENTER, or UGETF; the directory image remains in core until a new ASSIGN command is executed from the console. To inform the DECtape service routine that a new tape has been mounted on an assigned unit, the user uses an ASSIGN command. The directory from the old tape can be transferred to the new tape, thus destroying the information on that tape unless the user reassigns the DECtape transport every time he mounts a new reel.

Although DECtape is a file-structured blocked device, there is a limit to the number of files that may be opened simultaneously on a single DECtape. A given DECtape may be OPENed or INITed on two software channels (maximum) at the same time, once for INPUT and once for OUTPUT. An attempt to INIT on two channels for INPUT or two channels for OUTPUT generates no error indication, and only the most recent INIT is effective. This restriction explains why the following examples do not work.

Example 1:

```
.R FILCOM
*TTY:=DTA1:P1,DTA1:P2
```

FILCOM accepts the command string but the comparison does not work because the DECtape cannot be associated with the input side of two software channels at the same time.

Example 2:

```
.R MACRO
*DTA1:BIN,DTA1:LST←DTA2:PROG
```

MACRO accepts the command string but does not produce the desired results because a single DECtape cannot be associated with the output side of two software channels at the same time. However, the following example works, because only one file is opened for reading and one file for writing.

```
.R MACRO
*DTA1:BIN←DTA1:SOURCE
```

6.2 DISK

The device mnemonic is DSK, FHA, DPA; the buffer size is 203_8 (200_8 data) words.

6.2.1 Data Modes

6.2.1.1 Buffered Data Modes - Data is written on the disk exactly as it appears in the buffer. Data consists of 36-bit words.

CAUTION

All buffered mode operations utilize a 200 octal word data buffer. Attempts to set up non-standard buffer sizes are ignored. In particular, attempting to use buffer sizes smaller than 200 words for input result in data being read in past the end of the buffer destroying what information was there (e.g., the buffer header of the next buffer).

6.2.1.2 Unbuffered Data Modes - Data is read into or written from anywhere in the user's core area without regard to the normal buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is described in Chapter 4. The disk control automatically measures dump data into standard-length disk blocks of 200 octal words. Unless the number of data words is an exact multiple of the standard length of a disk block (200 words) after each command word in the command list, the remainder of that block is wasted.

6.2.2 Structure of Disk Files

The file structures of a disk system minimize the number of disk seeks for sequential or random access during either buffered or unbuffered I/O. The assignment of physical space for data is performed automatically by the monitor when logical files are written or deleted by user programs. Files may be any

length, and each user may have as many files as he wishes, as long as disk space is available and the user has not exceeded his logged-in quota. Users or their programs do not need to give initial estimates of file length or number of files. Files may be simultaneoulsy read by more than one user at a time, thus allowing data sharing. A new version of a file may be recreated by one user while other users continue to read the old version, thus allowing for smooth replacement of shared programs and data files. Finally, one user may selectively update portions of a file, rather than create a new one.

6.2.2.1 Addressing by Monitor - The file structure described in this section is generally transparent to the user, and a detailed knowledge of this material is not essential for effective user-mode use of the disk. One set of disk-independent file handling routines in the monitor services all disks and drums. This set of routines interprets and operates upon file structures, processed disk UUOs, queues disk requests, and makes optimization decisions. The monitor deals primarily with logical units within file structures and converts to physical units in the small device-dependent routines just before issuing I/O commands. All queues, statuses, and flags are organized by logical unit rather than by physical unit. The device-dependent routines perform the I/O for specific storage devices and translate logical block numbers to physical disk addresses.

All references made to disk addresses refer to the logical or relative addresses used by the system and not to any physical addressing scheme involving records, sectors, or tracks, that may pertain to a particular physical device. The basic unit that may be addressed is a logical disk block, which consists of 200_8 36-bit words.

6.2.2.2 Storage Allocation Table (SAT) Blocks - Unique to each file structure is a file named SAT.SYS. This file reflects the current status of every addressable block on the disk. Only the monitor can modify the contents of SAT.SYS as a result of file creation, deletion, or space allocation, although this file may be read by any user. The SAT file consists of bits indicating both the portion of file storage in use and the portion that is available. To reduce the size of SAT.SYS, each bit can be used to represent a contiguous set of blocks called a <u>cluster</u>. Monitor overhead is decreased by assigning and releasing file storage in clusters of blocks rather than single blocks.

If a particular bits is on, it indicates that the corresponding cluster is bad or nonexistent or has been allocated to a file. It may or may not contain data (i.e., files may contain allocated but unwritten clusters). If the bit is off, it indicates that the corresponding cluster is empty, or available to be written on.

It is recommended that cluster sizes should evenly divide blocks on a unit. In the 5.02 monitor, the refresher rounds up the number of clusters to the next highest full cluster. In the 5.03 and later monitors, the refresher truncates to the largest number of full clusters. With truncation, the last few

blocks are not included in the addressing space, but may be used for swapping; therefore, they are not part of SWAP.SYS even though they are in the swapping space. In addition, any bad blocks in the extra blocks are not included in SWAP.SYS.

6.2.2.3 File Directories - A directory is a file which contains as data pointers to other files on the disk. There are three levels of directories in each file structure:

a. The master file directory (MFD).
b. The user file directories (UFDs).
c. The sub-file directories (SFDs).[1]

The master file directory consists of two-word entries; the entries are the names of the user file directories on the file structure. The first word of each entry contains the project-programmer number of the user. The left half of the second word of each entry contains the mnemonic UFD in SIXBIT and the right half contains a pointer to the first cluster of the user file directory (see Figure 6-4). The main function of the master file directory is to serve as a directory of individual user file directories. A continued MFD is the MFDs on all file structures in the job's search list.

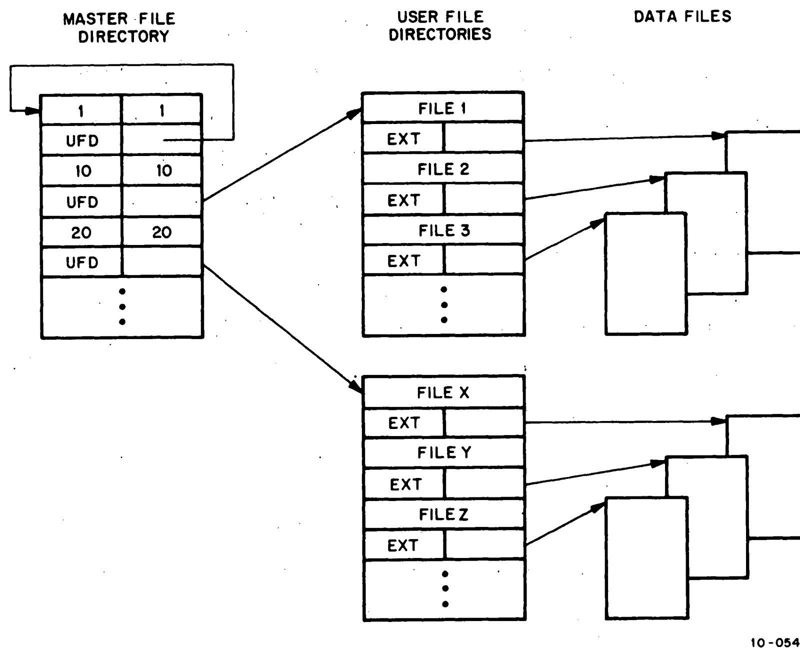


Figure 6-4 Basic Disk File Organization for Each File Structure

[1] Sub-file directories depend on FTSFD which is normally off in the DECsystem-1040.

The entries within a user file directory are the names of files existing in a given project-programmer number area within the file structure. The first word of each entry contains the filename in SIXBIT. The left half of the second word contains the filename extension in SIXBIT, and the right half contains a pointer to the first cluster of the file (see Figure 6-4). This pointer specifies both the unit and the super-cluster of the file structure in which the file appears. The right half of the directory entry is referred to as a compressed file pointer (CFP). A continued UFD is all the UFDs for the same project-programmer number on all file structures in the job's search list.

When the user is logged-in, each file structure for which he has a quota contains a UFD for his project-programmer number. Each UFD contains the names of all the user's files for that file structure only. UFDs are created only by privileged programs (i.e., LOGIN in response to a LOGIN command, and OMOUNT in response to a MOUNT command). A user is not prevented from attempting to read a file in another user's UFD on a file structure for which he does not have a UFD. Whether or not the user is successful depends on the protection specified for the file being referenced.

As an entry in the user file directory, the user can include a sub-file directory (SFD). The sub-file directory is similar to the other types of directories in that it contains as data all the names of files within the directory. This directory is pointed to by a UFD or a higher-level SFD nested in any arbitrary tree structure. The maximum number of nested SFDs allowed is defined via a MONGEN question and can be obtained from a GETTAB table (GETTAB table .GTLVD, item 17). Files can be written or read in SFDs nested deeper than the maximum but SFDs cannot be created. (There is an absolute maximum of 6, including the UFD.) Unlike UFDs, a sub-file directory can be created by any program. A continued SFD, or sub-directory, is all of the SFDs on all file structures in the job's search list with the same name and path.

This third level of directory allows groups of files belonging to the same user to be separate from each other. This is useful when organizing a large number of files according to function. In addition, simultaneous batch runs of the same program for a single user can use the same filenames without conflicting with each other. As long as the files are in different sub-file directories, they are unique.

A file is uniquely identified in the system by a file structure name, a directory path, a filename and an extension. The directory path is an ordered list of directory names, starting with a UFD, which uniquely specifies a directory without regard to a file structure. The PATH. UUO is used to set or read the default directory path for a job (refer to Paragraph 6.2.9.1). Default paths can be a job's UFD, an SFD in a job's UFD, a UFD different from the job's UFD, or an SFD in another UFD. If a default path is not specified, it is the job's UFD. The notation FILE.EXT [PPN,A,B,...,N] designates the file named FILE.EXT in the UFD [PPN] in the SFD N, which is in the SFD..., which is in the SFD A. The path to the file named FILE.EXT is [PPN,A,B,...,N].

MASTER FILE DIRECTORY | USER FILE DIRECTORY | USER FILES

10-0542

Figure 6-5 Disk File Organization

To improve disk access and core searching times, only UFD names are kept in the MFD (project-programmer number 1,1). All system programs and monitor file structure files are contained in another project-programmer number directory called the system library. For convenience both to users typing commands and to user programs, device name SYS is interpreted as the system library; therefore, no special programming is required to read as a specific file from device SYS. In command strings, the abbreviation SYSx: represents the system library on file structure DSKx; i.e., SYSA: represents the system library on DSKA.

6.2.2.4 File Format - All disk files (including directories) are composed of two parts:

a. pure data.
b. information needed by the system to retrieve this data.

Each data block contains exactly 200_8 words. If a partially filled buffer is output to the disk by a user, a full block is written with trailing zeros filling in to make 200_8 words. A partial block input later appears to have a full 200_8 data words. Word counts associated with individual blocks are not retained by the system except in the case of the last block of the file.

There are three links in the chain by which the system references data on the disk. This chain is transparent to the user, who might look on the directory as having four-word entries analogous to DECtapes. The first link is the two-word directory entry that points to the second link, the retrieval information block (RIB). The RIB, in turn, points to the third link, the individual data blocks of the file (see Figure 6-5).

The retrieval block contains all the pointers to the entire file. Retrieval information associated with each file is stored and accessed separately from the data; therefore, system reliability is increased because the probability of destroying the retrieval information is reduced. System performance is improved because the number of positionings necessary for random access is reduced.

For recovery purposes, a copy of the retrieval information block is written immediately after the last data block of the file when a CLOSE is completed. If the first RIB is lost or bad, the monitor can recover by allowing a recovery program to use the second RIB; therefore, a data file of n blocks has two additional overhead blocks: relative block 0, containing the primary RIB; and relative block n + 1, containing the redundant RIB (refer to Appendix H).

6.2.3 Access Protection

Nine bits of the retrieval information of a file are used to indicate the protection of that file. This protection is necessary because the disk is shared by many users, each of whom may desire to keep certain files from being written on, read, or deleted by other users. The nine bits are divided into

three classes because the users are divided into three categories: 1) the owner of the file, 2) the users with the same project number as the owner, and 3) all other users.

Ordinarily, the owner of a file is any user whose programmer number is the same as the programmer number of the UFD containing the file, regardless of whether the two project numbers match. Therefore, in order to maintain only one owner for each file, the installation should not assign the same programmer number to different users, no matter how many projects the installation has. A user working on more than one project, but having the same programmer number, can reference all his files as an owner under each of his project-programmer numbers.

However, some installations may decide that a user is the owner of a file only when both the project and programmer numbers under which the user is logged in match the pair identifying the UFD. If this is the case, the same programmer number can be assigned to different users in different projects. This allows the task of assigning programmer numbers to be delegated to project leaders without concern for duplication since the project numbers will be different from one project to another. However, a user working on more than one project cannot have the same owner access to all files that he has written.

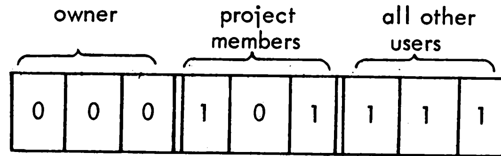
The definition of the owner of a file is specified at monitor generation time with MONGEN(INDPPN). No matter how the installation defines an owner, project numbers 0 to 7 are always independent of the project-programmer number (i.e., a user with a project number from 0 to 7 is considered the owner of all files with that project number).

A member of the owner's project is any user whose logged-in-project number is the same as the owner's, regardless of his programmer number.

The three bits associated with each category of users are encoded as follows:

Code	Access Protection
7	Greatest protection, which means no access privileges. However, the owner may LOOKUP the file so that he can change the protection to a less restrictive code via a RENAME. Thus for the owner, this code is equivalent to codes 6 and 5.
6	Execute-only. This disables user meddling and examining (DUMP, DCORE, D, E, SAVE, SSAVE, START n, CSTART n, DDT, COREn) with the error message ?ILLEGAL WHEN EXECUTE ONLY. An error return is given on a LOOKUP to an execute-only file to all users except the owner of the file.
5	Read, execute.
4	Append, read, execute.
3	Update, append, read, execute.
2	Write, update, append, read, execute.
1	Rename, write, update, append, read, execute.
0	Change protection, rename, write, update, append, read, execute.

The following example illustrates the nine-bit protection field of a file that has a protection of 057.



This code means:

1) The owner has complete privileges (code 0).
2) The project members have read and execute privileges (code 5).
3) All other users have no access privileges (code 7).

The greatest protection a file can have is 7, and the least is 0. Usually the owner's field is 0 or 1. However, it is always possible for the owner of a file to change the access protection associated with the file even if the owner-protection is not set to 0. Thus codes 0 and 1 are equivalent when they appear in the owner's field. Access protection can be changed by executing a RENAME UUO or by using the PROTECT monitor command as follows:

PROTECT file.ext <nnn >↵

When an ENTER UUO specifies a protection code of 000 and the file does not exist, the monitor substitutes the standard protection code as defined by the installation. The normal system standard is 057. This protection prevents users in different projects from accessing another user's files; however, a standard protection of 055 is recommended for in-house systems where privacy is not as important as the capability of sharing files among projects. No program should be coded to assume knowledge of the standard protection. If it is necessary to use this standard, it should be obtained through the GETTAB UUO.

To preserve files with LOGOUT, a protection code of 1 in the owner's field should be associated with the files. LOGOUT preserves all files in a UFD for which the protection code for the owner is greater than zero. The PRESERVE monitor command can be used to obtain a protection code of 1 in the owner's field.

6.2.3.1 UFD and SFD Privileges - The protection code associated with each file completely describes the access rights to that file independently of the protection code of the UFD. UFDs and SFDs may be read in the same manner as files but cannot be written explicitly, because they contain RIB pointers to particular disk blocks. For UFD and SFD privileges, users are divided into the same three categories as for files. Each category has three independent bits:

Bit	Access Privileges
4	Allow LOOKUPs in UFD or SFD.
2	Allow CREATEs in UFD or SFD.
1	Allow the UFD or SFD to be read as a file.

The owner is permitted to control access to his own UFD and SFD. It is always legal for the owner to issue a RENAME to change the protection of his directories. Any program can create or delete SFDs; however, only privileged programs are allowed to create, supersede, or delete a UFD. The monitor checks for the following types of privileged programs:

a. Jobs logged in under project-programmer number [1,2] (FAILSAFE).

b. Jobs running with the JACCT bit set in JBTSTS (LOGIN, LOGOUT).

Privileged programs are allowed to:

a. Create UFDs (and SFDs).
b. Delete UFDs (and SFDs).
c. Set privileged LOOKUP, ENTER, and RENAME arguments.
d. Ignore file protection codes.

UFD and SFD privileges are similar with the exception being that SFDs can be RENAMEd and deleted by both privileged programs and the owner of the SFD if his protection byte is 7.

6.2.4 Disk Quotas[1]

Each project-programmer number in each file structure is associated with two quotas that limit the number of blocks that can be stored under the UFD in the particular file structure. The quotas are:

a. Logged-in quota.
b. Logged-out quota.

When the user logs in, he automatically starts using his logged-in quota. Because this is not a guaranteed amount of space, the user competes with other users for it. The logged-out quota is the amount of space that the user must be within in order to log off the system. Normally, the logged-out quota is less than or equal to the logged-in quota, so that the user must delete temporary files.

If a user exceeds his logged-in quota, the monitor types the following message:

[EXCEEDING QUOTA ON fs]

where fs is the name of the file structure. The message appears in square brackets (like the TECO core expansion message) to suggest a warning rather than an error. Unlike most monitor messages, this message indicates that the user program may continue to run, and the console remains in user mode. The user program can no longer create or supersede files (ENTER gives an error return). Files already ENTERed are allowed to continue for a specified number of blocks. This amount is called the overdrawn amount and is a parameter of the file structure. The overdrawn amount specifies the number of blocks by which the logged-in UFD may exceed its logged-in quota. When the user exceeds the overdrawn amount, the IO.BKT bit is set, and further OUTPUTs are not allowed. A CLOSE operates successfully, including the writing of the last buffers and the RIBs.

When the user logs in, the LOGIN program reads the logged-in quota from the file AUXACC.SYS for all public file structures in which the user is allowed to have a UFD. This information is passed to the monitor where it is kept in core. If the quota has changed since the user logged in last, LOGIN updates (or creates) the RIB of each UFD with the new quotas.

---

[1]Quota checking depends on FTDQTA which is normally off in the DECsystem-1040.

6.2.5 Simultaneous Access

In its core area, the monitor maintains two four-word blocks called access blocks. These blocks control simultaneous access to a single file by a number of user channels. All active files have access blocks that contain file status information. The access blocks ensure that a maximum of one user channel supersedes or updates a given file at a given time.

6.2.6 File Structure Names

Each file structure has a SIXBIT name specified by the operator at system initialization time. This name can consist of four or less alphanumeric characters and must not duplicate any device, unit, or existing file structure name or its abbreviation. The recommended names for the file structures in the public pool are DSKA, DSKB, ..., DSKN (in order of decreasing speed).

When a specific file structure is INITed (e.g., DSKA), LOOKUP and ENTER searches are restricted to that file structure. Usually a channel is INITed with the generic name DSK, in which case all file structures in the active search list of the job are searched (refer to Paragraph 6.2.7).

6.2.6.1 Logical Unit Names - When a single file structure name is specified, the set of all the units in that file structure is implied; however, it is possible to specify a particular logical unit within a file structure (e.g., DSKA0, DSKA1, DSKA2 are three logical units in the file structure DSKA). The monitor deals with file structures rather than with individual units; therefore, when reading files, specifying a logical unit within a file structure is equivalent to specifying the file structure itself. The monitor locates the file regardless of which unit it is on within a file structure. However, in writing a file, the monitor uses the logical unit name as a guide in allocating space and will, if possible, write the file on the unit specified. In this way, a user can apportion files among different units for increased throughput.

6.2.6.2 Physical Controller Class Names - In addition to DSK, single file structure names (DSKA), and logical unit names (DSKA0), it is possible to specify a class of controllers. If the system has one controller of the type specified, the result is the same as if the user had specified the physical controller name. The controller classes supported by DEC are:

DR (future drum), FH, DP

6.2.6.3 Physical Controller Names - It is possible to specify any of the units on a particular controller. The monitor relates that name to the file structures, which contain at least one unit on the specified controller. More than one file structure may be specified when a physical controller name is used. The controllers that DEC supports are:

DRA, DRB (future drum), FHA, FHB, DPA, DPB

6.2.6.4 Physical Unit Names - When a physical controller name is specified, all units on that controller are implied. It is possible to specify a physical unit name on a particular controller. The physical unit names that DEC supports are:

DRA0, DRB0	Reserved for future drum (RX10).
FHA0, ..., FHA3	Mixture of Burroughs fixed-head disks (RD10) and Bryant drums (RM10B) on RC10 control.
FHB0, ..., FHB3	Mixture of Burroughs fixed-head disks (RD10) and Bryant drums (RM10B) on second RC10 control.
DPA0, ..., DPA7	Mixture of RP02 and RP03 disk packs on RP10 control.
DPB0, ..., DPB7	Mixture of RP02 and RP03 disk packs on second RP10 control.

6.2.6.5 Unit Selection on Output - If the user specifies a file structure name on an ENTER, the monitor chooses the emptiest unit on the file structure which does not currently have an open file (UFD's are not considered opened) for the job. This selection improves disk throughput by distributing files for a particular job on different units. For example, in a MACRO assembly with two output files and one input file, it is probable that the monitor would allocate the output files on units separate from each other and from the input file. If this were the only job running, there would be almost no seeks. Therefore, to take advantage of this, programs should LOOKUP input files before ENTERing output files.

6.2.6.6 Abbreviations - Abbreviations may be used as arguments to the ASSIGN command and the INIT and OPEN UUOs. The abbreviation is checked for a first match when the ASSIGN, INIT, or OPEN is executed. The file structure or device eventually represented by the particular abbreviation depends on whether a LOOKUP or ENTER follows. A LOOKUP applies to as wide a class of units as possible, whereas an ENTER applies to a restricted set to allow files to be written on particular units at the user's option. For example, consider the following configuration:

File Structure		Physical Unit
DSKA	=	FHA0, FHA1, FHA2
DKSB	=	FHB0, FHB1
DSKC	=	DPA0, DPA1, DPA2, DPA3
DSKD	=	DPB0, DPB1, DPB2
PRVA	=	DPB3

Table 6-5 shows the file structures and units implied by the various names and abbreviations.

Table 6-5
File Structure Names

Argument Supplied to ASSIGN, MOUNT, INIT, OPEN	File Structures or Units Implied	
	LOOKUP	ENTER
D, DS, DSK	Generic DSK according to job search list (refer to Paragraph 6.2.7)	
P, PR, PRV, PRVA	DPB3	DPB3
F, FH, FHA	DSKA, DSKB	FHA0
FHB	DSKB	FHB0
FHA0	DSKA	FHA0
FHB0	DKSB	FHB0
DP	DKSC, DSKD, PRVA†	DSKC
DPA	DSKC	DSKC
DPB	DSKD, PRVA†	DSKD
DPA0	DSKC	DPA0
DPB2	DSKD	DPB2
DPB3	PRVA	PRVA
†Only if user has done a MOUNT.		

6.2.7 Job Search List

To a user, a file structure is like a device; that is, a file structure or a set of file structures may be specified by an INIT or OPEN UUO or by the first argument of the ASSIGN or MOUNT command. A console user specifies a file structure by naming the file structure and following it with a colon.

There is a flexible naming scheme that applies to file structures; however, most user programs INIT device DSK, which selects the appropriate file structure, unless directed to do otherwise by the user. The appropriate file structure is determined by a job search list. A job search list is divided into two parts:

a. an active search list (usually referred to as the job search list), and
b. a passive search list.

The active search list is an ordered list of the file structures that are to be searched on a LOOKUP or ENTER when device DSK is used. The passive search list is an unordered list of file structures maintained by the monitor for LOGOUT time. At this time, LOGOUT requires that the total allocated

blocks on each UFD in both the active and passive search lists be below the logged-out quota. Each job has its own active search list (established by LOGIN) with file structures in the order that they appear in the administrative control file AUXACC.SYS. Thus, a user has a UFD for his project-programmer number in each file structure in which LOGIN allows him to have files. With the MOUNT command, mounted file structures may be added to the active search list. The following is an example of a search list:

DSKB, DSKA, FENCE, DSKC

DSKB and DSKA comprise the active search list. These file structures are represented by generic name DSK for this job. DSKC is the name of a file structure that was previously in the active search list. FENCE represents the boundary between the active and passive search list.

Each file structure in a job search list may be modified by setting one of two flags with the JOBSTR UUO:

a. Do not create in this structure if just generic DSK is specified.
b. Do not write in this structure.

Setting the "do not create" flag indicates that no new files are to be created on this file structure unless explicitly stated. For example: if the "don't create" flag is set

DSKA: FOO ←

allows FOO to be created on DSKA, but

DSK: FOO ←

does not. For LOOKUPs on device DSK, the monitor searches the structures in the order specified by the job search list. For ENTERs when the filename does not exist (creating, see below), the file is placed on the first file structure in the search list that has space and does not have the "do not create" flag set. For ENTERs when the filename already exists on any file structure in the search list (superseding, see below), the file is placed on the same structure that contains the older file. If the write-lock bit is set for the file structure, a write-lock error (ERWLK%) is given on the supersede. Because superseding is treated differently from creating, a user may explicitly place a file on a particular file structure, for example, a fast one with the do not create bit set, so that subsequent supersedes will remain on that file structure even though generic DSK is used.

6.2.8 User Programming

Three types of writing on the disk may be distinguished. If a user does an ENTER with a filename which did not previously exist in his UFD, he is said to be <u>creating</u> that file. If the filename previously existed in his UFD, he is said to be <u>superseding</u> that file; the old version of the file stays on the disk

(and is available to anyone who wants to read it) until the user does the output CLOSE. At the time of the CLOSE, the user's UFD is changed to point to the new version of the file and the old version is either deleted immediately or marked for deletion later if someone is currently reading it; the space occupied by deleted files is always reclaimed in the SAT tables (refer to Paragraph 6.2.2.2). Finally, if a user does a LOOKUP followed by an ENTER (the order is important) on the same filename on the same user channel, he will be able to modify selected blocks of that file, using USETO and USETI UUOs (refer to Paragraph 6.2.9.2) without creating an entirely new version; this third type of writing, called updating, eliminates the need to copy a file when making a small number of changes. A LOOKUP followed by an ENTER and OUTPUT (in that order) writes the output at the beginning of the file. To append information to the file, a USETI -1 is used before the OUTPUT.

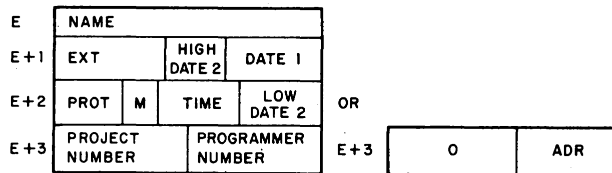
As a standard practice, user programs should read, create, and supersede (new file with same filename) files on different user channels. However, for compatibility with DECtapes, it is possible to read and create, or read and supersede, two files on the same user channel as long as all OUTPUTs and the CLOSE output are done before the LOOKUP and the first input, or vice versa. In other words, a CLOSE UUO is required between successive LOOKUPs and ENTERs unless updating is intended.

The actual file structure of the disk is generally transparent to the user. In programming for I/O on the disk, a format analogous to that of DECtapes is used; that is, the user assumes a four-word directory entry similar in form to the first four words of retrieval information. The UUO format is approximately the same as for DECtapes:

UUO D,E

where UUO is an I/O programmed operator, and D specifies the user channel associated with this device. E points either to a four-word directory entry or an extended argument block in the user's program.

6.2.8.1 Four-Word Arguments for LOOKUP, ENTER, RENAME UUOs - The four-word argument block has the following format:



where

NAME is the filename in SIXBIT, or, if a UFD, is the project number in the left half and the programmer number in the right half.

EXT is the filename extension in SIXBIT ASCII. (continued on next page)

HIGH DATE 2 contains the high order 3 bits of the date on which the file was originally created (bits 18-20).

DATE 1 is the date the file was last referenced (RENAME, ENTER, or INPUT) in the format of the DATE UUO (bits 21-35).

PROT is the protection code for the file (bits 0-8).

M is the data mode (ASCII, binary, dump) (bits 9-12).

TIME is the time that the file was originally created, represented as the number of minutes past midnight of the creation date (bits 13-23).

LOW DATE 2 is the low order 12 bits of the date (in the same format as the DATE UUO) on which the file was originally created (bits 24-35).

NOTE

The two-part format for DATE 2 (creation date) is used to maintain compatibility with monitors and media as old as 1964.

The programmed operators (UUOs) operate as follows:

a. ENTER UUO - ENTER D, E causes the monitor to store the four-word directory entry for later entry into the proper UFD or SFD when user channel D is closed or released.

NAME	The filename must be nonzero; otherwise, an error return results.
EXT	The filename extension may be zero; if so, the monitor leaves it as zero.
HIGH DATE 2	If a nonzero date is obtained by concatenating the high order 3 bits in this field with the low order 12 bits in LOW DATE 2, then the monitor uses that value as the creation date for the file. If the date is zero, the monitor supplies the high order digits of the current date from the overflow in E + 2.
DATE 1	The date may be zero, in which case the monitor substitutes the current date. The date must not be in the future; if this is so, the current date is used.
PROT	If the protection code is 0, the monitor substitutes the installation standard as specified at MONGEN time. If the protection code is 0 and this ENTER is superseding a file, the protection of the new file is copied from the old file. RENAME may be used to change the protection after a file has been completely written and when it is being closed.
M	The data mode is supplied by the monitor. It was set by the user in the last INIT or SETSTS UUO on channel D.

TIME, LOW DATE 2	If these are 0, and bits 18-20 of E + 1 are zero the monitor supplies the current date and time as the creation date and time for the file. The high order digits of the creation date overflow to bits 18-20 of E + 1 (HIGH DATE 2). If either is nonzero, the monitor uses the HIGH DATE 2 supplied by the user in E + 1 and the TIME and LOW DATE 2 supplied in E + 2. Thus, files may be copied without changing the original creation time and date.
PROJECT NUMBER PROGRAMMER NUMBER	If this word is 0, the file will be written in the default directory. (For example, if the default path is [10, 10, A], the file will be written in SFD A which is contained in [10, 10] .UFD.) The default path is determined by the PATH. UUO (refer to Paragraph 6.2.9.1). If a default path has not been specified via the PATH. UUO, it is the job's UFD (i.e., the project-programmer number under which the user is logged in). If this word is a project-programmer number, the file will be written in the UFD specified (i.e., sub-directories will not be scanned). This allows the program to write in the disk area under which the job is logged in although the default directory is different. Note that it is generally not possible to create (ENTER) files in another user's area of the disk, because UFDs are usually protected from all but the owner when creating files. If this word is XWD 0, ADR, the file will be written according to the path specified by ADR. The argument block beginning at ADR is the same as in the PATH. UUO (refer to Paragraph 6.2.9.1) except that the first two arguments (ADR and ADR + 1) are ignored. The scan switch (ADR + 1) is not needed since if the file is found in the specified directory, it will be superseded, and if not found, it will be created at the end of the path of the specified directory, even if a file with the same name appears in an upper-level directory. A path specification in the ENTER block overrides any default path specification given in the PATH. UUO.

With certain types of error returns peculiar to the disk, the right half of E + 1 is set to a specific number to indicate the error that caused the return. For example, if the extension UFD is specified and bit 18 (RP.DIR) of the file status status word is not set, the right of E + 1 is set to 2 (protection failure). Refer to Paragraph 6.2.8.3 for a special note on error recovery. Refer to Appendix E for the error codes returned on the ENTER UUO.

When issuing a supersede ENTER (an ENTER after a LOOKUP on the same channel), the user should check that locations E through E + 3 are as he desires.

When an ENTER is executed by the monitor on a file that exists, a new file by that name is written, and those bits in the SAT blocks that correspond to the blocks of the old file are zeroed when the CLOSE (or RELEAS) UUO is executed provided that bit 30 of the CLOSE is 0 (refer to Paragraph 4.7.7). Space is thereby retrieved and available to other users after the new file has been successfully written. If a file structure is INITed on channel D, the monitor maximizes the job's throughput by selecting the emptiest unit for which the job has no opened files (refer to Paragraphs 6.2.6.5 and 6.2.6.6).

b. LOOKUP UUO - LOOKUP D, E causes the monitor to read the appropriate UFD or SFD. If a later version of the file is being written, the old version pointed to by the UFD is read.

NAME	The same as on an ENTER.
EXT	The same as on an ENTER.
DATE 1, PROT, M, TIME, LOW and HIGH DATE 2	These arguments are ignored. The monitor returns these quantities to the user in E + 1 and E + 2.
PROJECT NUMBER PROGRAMMER NUMBER	If this word is 0, the file will be read from the user's default directory path. The entire path is searched only if the scan switch is set via the PATH. UUO (refer to Paragraph 6.2.9.1). If a default path has not been specified, it is the project-programmer number under which the user is logged in. If a project-programmer number is specified, the file will be read from the UFD specified (i.e., sub-directories will not be scanned). Thus, it is possible to read files in other user's directories, provided the file's protection mask permits reading and the UFD permits LOOKUPs. If this word is XWD 0, ADR, the file will be read according to the path specified by ADR. The argument block beginning at ADR is the same as in the PATH. UUO except that the first argument is ignored, and the second argument, if 0, uses the default value of the scanning switch (refer to the PATH. UUO). A path specification in the LOOKUP block overrides any default path specification given in the PATH. UUO.

The monitor returns the negative word count (or positive block count for files larger than 2^{17} words) in the LH of E + 3, 0 in RH of E + 3 when the four-word argument block is given. As a result, the monitor treats a negative project-programmer number as if it were 0, however, this will not always be true; therefore, programs must be written to either clear E + 3 before doing a LOOKUP, ENTER, or RENAME or set E + 3 to the desired project-programmer number. In the future, a negative project-programmer number may be used to indicate SIXBIT alphabetic characters for project and programmer initials.

The numbers placed in the RH of E + 1 on an error return have a significance analogous to that described for the ENTER UUO (refer to Appendix E).

If the file is currently being superseded, the old file is used.

c. RENAME UUO - RENAME D, E is used to alter the filename, the filename extension and/or protection of a file, or to delete a file from the disk. This UUO can be used to change the name of an SFD, but an attempt to change the extension or project-programmer number associated with an SFD, the name, extension, or project-programmer number associated with a UFD, or the project-programmer number of a device with an implied project-programmer number (e.g., SYS:, NEW:, OLD:) results in a protection error. To RENAME a file, a LOOKUP or ENTER must first be done to identify the file for the RENAME UUO. Locations E through E + 2 are as described for ENTER. If E + 3 = 0, there is no change in the directory of the file. If E + 3 is the default project-programmer number, the file is renamed in that UFD. If E + 3 has a different project-programmer number than the one in which the file is LOOKUPed or ENTERed (i.e., E + 3 is not the default project-programmer number), the monitor deletes the directory entry from the old directory (UFD or SFD) and inserts the directory entry into the specified UFD, provided the user has the privileges to delete files from the old directory, and to create files in the new UFD. (This is an efficient way to move a file from one UFD to another, since no I/O needs to be done on the data blocks of the file.) If E + 3 = XWD 0, ADR, the file is renamed into a new SFD or UFD according to the path specified by ADR. (Refer to the PATH. UUO.) Therefore, the only way to RENAME a file into a SFD different from the one which it is in to give an explicit path via an argument block.

A CLOSE is optional because RENAME performs a CLOSE. However, a CLOSE should not be issued between a LOOKUP and RENAME if the file is not in the default path or cannot be obtained from the default path by scanning because CLOSE erases all memory of the path of a file. If a CLOSE is performed and the file is not in the default path, the RENAME returns the FILE NOT FOUND error. In addition, disk accesses are minimized if a CLOSE does not precede a RENAME.

RENAME enters the information specified in E through E + 2 into the retrieval information and proper directory. If the contents of E is zero, RENAME has the effect of deleting the file. Although only a privileged job can delete a UFD, any job can delete an SFD. If the directory is not empty or if a job is currently using the directory, the RENAME returns the DIRECTORY NOT EMPTY error. (Refer to Appendix E for the error codes.) Refer to Paragraph 6.2.8.3 for a special note on error recovery.

When issuing a RENAME UUO, the user must ensure that the status at locations E through E + 3 are as he desires. An ENTER or LOOKUP must have preceded the RENAME; therefore, the contents of E through E + 3 will have been altered, or filled if the E is the same for all UUOs.

d. Examples - The sample code below can be used to assemble the 15-bit creation date of a disk (or DECtape) file in register T1 after a successful LOOKUP. The four-word argument block begins at location E.

```
LDB     T1, [POINT 12, E+2, 35]     ;GET LOW-ORDER PART
LDB     T2, [POINT 3, E+1, 20]      ;GET HIGH-ORDER PART
DPB     T2, [POINT 3, T1, 23]       ;MERGE THE TWO PARTS
```

The following sample code illustrates setting the 15-bit creation date in the four-word ENTER argument block from the value in register T1.

```
DPB     T1, [POINT 12, E+2, 35]     ;STORE LOW-ORDER PART
ROT     T1, -↑D12                   ;POSITION HIGH PART
DPB     T1, [POINT 3, E+1, 20]      ;STORE HIGH-ORDER PART
```

6.2.8.2 Extended Arguments for LOOKUP, ENTER, RENAME UUOs - A number of quantities have been added to the existing four-word block. The user program may specify exactly the number of words in the argument block. If the left half of E is 0 and the right half of E is three or greater, the right half of E is interpreted as the count of the number of words which follow. If the right half of E is less than three, a file-not-found return is given because the user program is not supplying enough arguments. Allowed arguments supplied by the user program are returned by the monitor as values. If the user program supplies arguments that are not allowed, the monitor ignores these arguments and supplies values on return. Table 6-6 indicates the arguments that may be supplied by a user program.

Table 6-6
Extended LOOKUP, ENTER, and RENAME Arguments

Rel. Loc	Symbol	Lookup	Create Supers	Update Rename	Arguments and Value
0	.RBCNT	A	A	A	Count of arguments following
1	.RBPPN	A0	A0	A0	Directory name (project-programmer no.) or pointer
2	.RBNAM	A	A	A	Filename in SIXBIT
3	.RBEXT	A	A	A	File extension (LH)
		V	A0	A	High order 3 bits of 15-bit creation date (bits 18-20)
					Access date (bits 21-35)
4	.RBPRV	V	A0	A	Privilege (bits 0-8)
		V	V	A	Mode (bits 9-12)
		V	A0	A	Creation time (bits 13-23)
		V	A0	A	Low order 12 bits of 15-bit creation date (bits 24-35)
5	.RBSIZ	V	V	V	Length of file in data words written (+no. words)
6	.RBVER	V	A	A	Octal version number (36 bits)
7	.RBSPL	V	A	A	Filename to be used in output spooling.
10	.RBEST	V	A	A	Estimated length of file (+no. blocks)
11	.RBALC	V	A	A	Highest relative block number within the file allocated by user or monitor to file.

A = Argument (supplied by privileged or nonprivileged user program) and returned by monitor as a value.

A0 = Argument like A with the addition that a 0 argument causes the monitor to substitute a default value.

V = Value (returned by monitor) cannot be set even by privileged program, monitor ignores argument.

A1 = Argument if privileged program (ignored if nonprivileged).

(continued on next page)

Table 6-6 (Cont)
Extended LOOKUP, ENTER, and RENAME Arguments

Rel. Loc	Symbol	Lookup	Create Supers	Update Rename	Arguments and Value
12	.RBPOS	V	A	A	Logical block no. of first block to allocate within F.S.
13	.RBFT1	V	A	A	Future nonprivileged argument - reserved for DEC
14	.RBNCA	V	A	A	Nonprivileged argument reserved for customer to define
15	.RBMTA	V	A1	A1	Tape label if on backup tape
16	.RBDEV	V	V	V	Logical unit name on which the file is located
17	.RBSTS	V	A1	A1	1) LH=Combined status of all files in UFD 2) RH=Status of this file
20	.RBELB	V	V	V	Bad logical block within error unit
21	.RBEUN	V	V	V	1) LH=Logical unit no. within F.S. of bad unit (0,,,N).
		V	V	V	2) RH=No. of consecutive blocks in bad region
22	.RBQTF	V	A1	A1	(UFD-only) FCFS logged-in quota in blocks
23	.RBQTO	V	A1	A1	(UFD-only) logged-out quota in blocks
24	.RBQTR	V	A1	A1	(UFD-only) reserved logged-in quota
25	.RBUSD	V	A1	A1	(UFD-only) no. of blocks used at last logout
26	.RBAUT	V	A1	A1	Author project-programmer number (creator or superseder)
27	.RBNXT	V	A1	A1	Next file structure name if file continued
30	.RBPRD	V	A1	A1	Predecessor file structure name if file continued
31	.RBPCA	V	A1	A1	Privileged argument word reserved for each customer to define as he wishes
32	.RBUFD	V	V	V	Logical block number within F.S. (not cluster no.) of the RIB of the UFD in which the name of this file appears
33	.RBFLR	V	V	V	Relative block number in file of first block in RIB
34	.RBXRA	V	V	V	Extended RIB address
35	.RBTIM	V	V	V	Creation date in universal date-time standard (refer to Paragraph 3.6)

A = Argument (supplied by privileged or nonprivileged user program) and returned by monitor as a value.

A0 = Argument like A with the addition that a 0 argument causes the monitor to substitute a default value.

V = Value (returned by monitor) cannot be set even by privileged program, monitor ignores argument.

A1 = Argument if privileged program (ignored if nonprivileged).

The following explanation is a more complete description of the terms used in Table 6-6.

.RBPPN
LH=octal project number (right-justified).
RH=octal programmer number.
The project-programmer number is of the UFD in which the file is to be LOOKedUP, ENTERed, or RENAMEd. To LOOKUP the MFD, .RBPPN must contain a 1 in the left half and a 1 in the right half indicating that the filename (.RBNAM) is to be LOOKedUP in project 1, programmer 1's UFD (the MFD).

.RBNAM
SIXBIT filename, left justified with trailing nulls. If the MFD or UFD is being LOOKedUP, ENTERed, or RENAMEd, this location contains the project-programmer number. If a SFD is being LOOKedUP, ENTERed, or RENAMEd, this location contains the directory name. The argument can be 0 only on a RENAME, in which case the file is deleted. If the filename is not left justified on ENTER, most programs are unsuccessful on a subsequent LOOKUP. The monitor cannot left-justify the argument because it may be an octal project-programmer number.

.RBEXT
LH=SIXBIT filename extension, left justified with trailing nulls. Null extensions are discouraged because they convey no information. If the extension is not left justified on ENTER, most programs are unsuccessful on a subsequent LOOKUP. RH, bits 18-20 = high order 3 bits of 15-bit creation date, bits 21-35=access date in standard format. If an error return is given, bits 18-35 are set to an error code by the monitor before the error (no skip) return is taken. Refer to Paragraph 6.2.8.3 for a special note on error recovery.

.RBPRV
Bits 0-8=protection codes. (RB.PRV)
Bits 9-12=data mode in which file is created. (RB.MOD)
Bits 13-23=creation time in minutes since midnight. (RB.CRT)
Bits 24-35=low order 12 bits of 15-bit creation date in standard format. (RB.CRD)

.RBSIZ
Written length of file. The word is the positive number of words written in the file. For extended arguments, this word is never used for project-programmer numbers. (The four-word block remains compatible so that LH=-number of words in file, RH=0.) This argument is ignored, and a value is always returned.

.RBVER
Octal version number like the contents of location 137 in the job data area.

LH=patch level (A=1, B=2, etc.)
Set by monitor except in the case of privileged programs.

RH=octal version number, never converted to decimal. This argument is accepted, except on a LOOKUP. If a user program wishes to increase the version number by 1 on each UPDATE, it should add 1 to location E + 6 between the LOOKUP and the ENTER.

.RBSPL
Filename to be used to label the output on a device which is being spooled. The filename is taken from the ENTER to the device, or is 0 if an ENTER was not done.

.RBEST
Estimated length of file in positive number of blocks. On an ENTER, FILSER uses this value as the number of blocks to allocate for the file. If the estimated number of blocks is too low, incremental allocation is done.

.RBALC — Number of 128-word blocks, N, to be allocated to the file after completion of ENTER or RENAME. This number includes the RIBs of the file. N is equivalent to last relative block of the file.

A 0 means do not change allocation rather than deallocate all the blocks of the file. All of the data blocks can be deallocated by superseding the file and doing no outputs before the CLOSE. This argument can be used to allocate additional space onto the end of the file, deallocate previously allocated but unwritten space, or truncate written data blocks.

The smallest unit of disk space that the monitor can allocate is a cluster of 128-word blocks. Typically small devices use a cluster size of 1 block. If N is not the last block of a cluster, the monitor rounds up, thereby adding a few more blocks than the user requested.

.RBPOS — Logical block number, L, of the first block to be allocated for a new group of clusters appended to the file. A logical block number is specified with respect to the entire file structure. Logical block numbers begin with logical block number 0. This feature combined with DSKCHR UUO allows a user program to allocate a file with respect to tracks and cylinders for maximum efficiency when the program runs alone. Because SAT blocks, swapping space, and bad blocks are scattered throughout a file structure, programs using this feature must be prepared to handle such contingencies. It is discouraged for any programs to depend on blocks actually used for allocation to operate without errors.

.RBFT1 — Future nonprivileged argument reserved for DEC.

.RBNCA — Nonprivileged argument reserved for customer definition.

.RBMTA — A 36-bit tape label if file has been put on magnetic tape. If allocated space is 0, then file was deleted from disk when it was copied on magnetic tape. Argument is accepted only from privileged programs; otherwise, it is ignored.

.RBDEV — The logical name of the unit on which the file is located. Ignored as an argument, returned as a value.

.RBSTS — File status word

LH=status of UFD. Bit 0=1 (RP.LOG) if the user is logged in and is set by LOGIN. LOGOUT clears this bit.

RH=status of file.

Bit 18=1 (RP.DIR) if file is a directory file; needed to protect the system from a user who might try to modify a directory file. The protection error is given if extension UFD is given on an ENTER or RENAME and this bit is not set.

Bit 19=1 (RP.NDL) if file cannot be deleted, renamed, or superseded, even by a privileged program or by a user logged in under [1,2] .

Bit 21=1 (RP.NFS) if file should not be dumped by FAILSAFE because certain files are needed before FAILSAFE can run.

Bit 22=1 (RP.ABC) if file always has bad checksum (because the monitor never recomputes the checksum) e.g., SWAP.SYS, SAT.SYS.

Number of 128-word blocks, N, to be allocated to the file after completion of ENTER or RENAME. This number includes the RIBs of the file. N is equivalent to last relative block of the file.

(continued on next page)

.RBSTS (Cont) Bit 26=1 (RP.CMP) if UFD compressing.

Bit 32=1 (RP.BFA) if file is bad because of a FAILSAFE restore.

Bit 33=1 (RP.CRH) if file was closed after a crash.

Bit 35=1 (RP.BDA) if file is bad because of damage assessment.

The following bits appear in both the LH and RH of this location:

Bit 11 (RP.URE) and bit 29 (RP.FRE) = 1 if any file in this UFD (or this file) has had a hard data error while reading. (The IO.DTE bit has been set.) An entry is made in the BAT block so that the bad region is not reused.

Bit 10 (RP.UWE) and bit 28 (RP.FWE) = 1 if any file in this UFD (or this file) has had a hard data error while writing. (The IO.DTE bit has been set.) An entry is made in the BAT block so that the bad region is not reused.

Bit 9 (RP.UCE) and bit 27 (RP.FCE) = 1 if any file in this UFD (or this file) has had a software checksum error or redundancy check error. (The IO.IMP bit has been set.)

NOTE

Device errors (IO.DER) are not flagged in the file status word because they refer to a device and disappear when a device is fixed.

.RBELB Logical block number within the unit on which last date error (IO.DTE) occurred, as opposed to block within file structure. Set by the monitor in the RIB on a CLOSE when the hardware detects either a hard bad parity error or two search errors while reading or writing the file. Device errors, checksum, and redundancy errors are not stored here. This argument is ignored, and a value is returned.

.RBEUN LH=logical unit number within file structure on which last bad region was detected.

RH=number of bad blocks in the last-detected bad region. The bad region may extend beyond the file. This argument is ignored, and a value is returned.

.RBQTF Meaningful for UFD only. Contains first-come-first-served logged-in quota. This quota is the maximum number of data and RIB blocks that can be in this directory in this structure while the user is logged in. The UFD and its RIB are not counted. Argument is ignored unless it is from a privileged program.

.RBQTO Meaningful for UFD only. Contains logged-out quota. This quota is the maximum number of data and RIB blocks that can be left in this directory in this file structure after the user logs off. LOGOUT requires the user to be below this quota to log off. LOGIN stores these quotas in the RIB of the UFD, so that LOGOUT does not have to scan ACCT.SYS at LOGOUT time to find the quota. Argument is ignored unless it is from a privileged program.

.RBQTR Meaningful for UFD only. (Reserved for the future.) Contains reserved logged-in quota. This quota is the guaranteed number of blocks the user has when he logs in. Argument is ignored unless it is from a privileged program.

.RBUSD Meaningful for UFD only. Contains number of data and RIB blocks used in this directory in this file structure by the user when he last logged off. LOGIN reads this word so that it does not have to LOOKUP all files in order to set up the number of blocks the user has written. LOGIN sets bit 0 of the file status word (.RBSTS) and LOGOUT clears it in order to indicate whether LOGOUT has stored the quantity. Argument is ignored unless it is from a privileged program.

.RBAUT Contains project-programmer number of the creator or superseder of the file, as opposed to owner of file. Usually the author and the owner are the same. Only when a file is created in a different directory are these different. This argument is used by Batch for validating queue entries in other directories. Argument is ignored unless it is from a privileged program.

.RBNXT Reserved for future.

.RBPRD Reserved for future.

.RBPCA Privileged argument reserved for customer definition.

.RBUFD The logical block number (not cluster number) in the file structure of the RIB of the UFD in which the name of this file appears.

.RBFLR The relative block number of the file to which the first pointer of this RIB points. It is used for multiple RIBs (i.e., 0 for prime RIB).

.RBXRA The extended RIB address (logical unit number and cluster address of next RIB in a multiple-RIB file).

.RBTIM The date and time of creation of the file in the universal date-time standard (refer to Paragraph 3.6). That is, the LH contains the date and the RH contains the time as a fraction of a day.

6.2.8.3 Error Recovery for ENTER and RENAME UUOs - Error codes for the LOOKUP, ENTER, and RENAME UUOs are returned in the right half of location E + 1 of the four-word argument block and in the right half of location E + 3 (.RBEXT) in the extended argument block. This means that the error code overwrites the high order 3 bits of the creation date and the entire access date. Since the vast majority of programs recover from these errors either by aborting or by reinitializing the entire argument block, this overwriting of data does not cause any problems. However, a small number of programs may attempt recovery by fixing just the incorrect part of the argument block and then retrying the UUO. These programs should always restore the right half of location E + 1 before retrying an ENTER or a RENAME UUO. (In order to eliminate problems for programs recovering from errors for files with zero creation dates, which is the most common case, error codes are restricted to a maximum of 15 bits even though the entire right half of E + 1 is used. In addition, the 5.06B and later monitors force access dates to be greater than or equal to the creation date, but never greater than the current date.)

6.2.9 Special Programmed Operator Service

The following are special programmed operator service UUOs.

6.2.9.1 PATH. AC, or CALLI AC, 110[1] - This UUO sets or reads the default directory path, or reads the current directory path on a channel. The call is:

```
          MOVE AC, [XWD n, ADR]
          PATH. AC,
          error return
          normal return
ADR:      arg
          scan switch
          ppn
          SFD1 name
          SFD2 name
            :
ADR+n-1:  0
```

The first word of the argument block contains one of the following:

C (ADR) = SIXBIT device name, or XWD 0, D[2]
 Return the current path for the specified device or channel D.

C (ADR) = XWD JOB, -1
 Return the default directory path.

C (ADR) = -2
 Define the default directory path.

C (ADR) = -3
 Define the additional path to be searched when a file is not found in the user's directory path.

C (ADR) = XWD JOB, -4
 Return the additional path to be searched when a file is not found in the user's directory path.

[1] This UUO depends on FTSFD which is normally off in the DECsystem-1040.

[2] Note that this function of the PATH. UUO is available even if FTSFD is turned off.

If the left half of ADR is a job number N and the right half of ADR is -1 or -4, the returned values are for either

1) job N if 0<N≤ the highest legal job number, or
2) the current job if N is outside the above range (i.e., N≤ 0 or N > the highest legal job number).

When defining a path within a UFD (C(ADR) = -2), ADR+1 is the scan switch, ADR+2 is the default project-programmer number, and the remainder of the argument block up to the first zero word defines the default path. The scan switch determines whether or not the monitor scans for the file on a LOOKUP. If the switch is 1, the monitor examines the specified directory only; higher level directories are not searched. If the switch is 2, the following occurs:

1) The monitor searches the UFD or SFD specified by the path (either explicit or default path). If the file is found, the scan is terminated.
2) If the file is not found, the monitor backs up one directory along the path and continues the scan (i.e., it scans the directory in which the current SFD appears). The scan is terminated when the UFD is searched or when the file is found.

Scanning allows directories to be nested since any file not found in the current SFD is obtained automatically from a higher level directory. This is useful when a user has a default directory in use containing files he is currently working on and a higher level directory containing checked-out routines. Since SFDs are continued across file structures but the depth of the nesting of directories is not necessarily the same on each file structure, each scan searches the file structures that are 1) in the job's search list and 2) have SFDs to the depth specified in the path. The file structures are searched in the same order as they appear in the search list.

On an ENTER, the scan switch is ignored; if the file is found in the specified directory, it will be superseded. If the file is not found, it will be created at the end of the path in the specified directory whether or not a file with the same name appears in a higher level directory.

When defining the additional path to be used after the user's directory path is searched (C(ADR) = -3), ADR+1 indicates if SYS (bit 35 = 1) or experimental SYS (bit 34 = 1) is to be scanned, and ADR+2 is the project-programmer number to be used for a user library. These locations are used as follows. If the file is not found in the user's directory path on a LOOKUP DSK:, the directory specified in ADR+2 is searched for the file. This directory must be a UFD and allows users with different directory paths to share a common directory of files. If the file is not found in the library and if bit 35 of ADR+1 is set, the system library (SYS:[1,4]) is searched. In addition on a LOOKUP SYS:, if bit 34 of ADR+1 is set, the directory area [1,5] is searched before the system library area [1,4]. The [1,5] area is called the experimental SYS area (NEW:) and can be used to separate software that is near the end of the development and testing stages from the standard system software on the system library [1,4].

When returning a path, ADR+1 contains the following:

bits 34 and 35	the scan switch
bit 33=1	if experimental SYS (NEW:) is searched
bit 32=1	if SYS is searched
bit 31=1	if there is a user library

(continued on next page)

bit 30=1 — if the user-supplied project-programmer number is to be ignored on a LOOKUP or ENTER UUO and the implied project-programmer number of the device is to be used (e.g., [1,4] if SYS; [1,5] if NEW). The implied project-programmer number is returned in ADR +2.

bits 27-29 — the type of search list:
0 a non-standard search list (e.g., DSKA)
1 job search list
2 ALL search list
3 SYS search list

and ADR+2 through ADR+n-1 is the path. If the path is less than n-1 words, a zero word is stored at the end. If ADR contains a device name or channel number when the UUO is called, the file structure name or ersatz device name is returned in ADR depending on the name specified (e.g., SYS is returned only if C(ADR) = SYS and the job does not have a device with the logical name SYS). If a LOOKUP or ENTER has been done on the specified device or channel number, the following is returned in the argument block:

ADR: the SIXBIT name of the file structure or ersatz device
ADR+1: the scan switch.
ADR+2: the actual project-programmer number associated with the file.
ADR+3: the actual path of the file.
:
ADR+m: 0 the end of the path if m <n-1.

If no LOOKUP or ENTER has been done, the following is returned:

ADR: SIXBIT DSK or ersatz device name.
ADR+1: the scan switch.
ADR+2: the job's default project-programmer number (or the project-programmer number of the ersatz device).
ADR+3: the default path to the file.
:
ADR+m: 0 the end of the path if m <n-1

On an error return,

AC is unchanged if the UUO is not implemented. (SFD remains a reserved extension, but all SFD code disappears.) The GETTAB which returns the maximum number of SFDs allowed returns 0 or fails. The default path is the user's project-programmer number.

AC is 0 if the device or channel number does not represent a disk.

AC is -1 if a SFD in the path specification is not found.

(continued on next page)

Examples

1) This example sets the default path to [27,235,SUB] with no scanning in effect.

```
   MOVE 1, [XWD 5, A]
   PATH. 1,
   error
   normal
A: -2
   1
   27,235
   SUB
   0
```

2) Refer to Figure 6-6. The path plus filename for file A is X.MAC [10,63]. The path plus filename for file B is Y.CBL [14,5]. The path plus filename for file C is Z.ALG [14,5,M].

3) Refer to Figure 6-7. The job's search list is DSKA/N, DSKB, DSKC and the default path is [PPN, A, B, C, D].

   A) LOOKUP DSK: with no matches scans in order: DSKA:D (.SFD), DSKA:C, DSKB:C, DSKA:B, DSKB:B, DSKA:A, DSKB:A, DSKA:PPN (.UFD), DSKB:PPN DSKC:PPN.

   B) LOOKUP DSK: FILE2 finds DSKA: FILE2 [PPN, A, B, C].

(continued on next page)

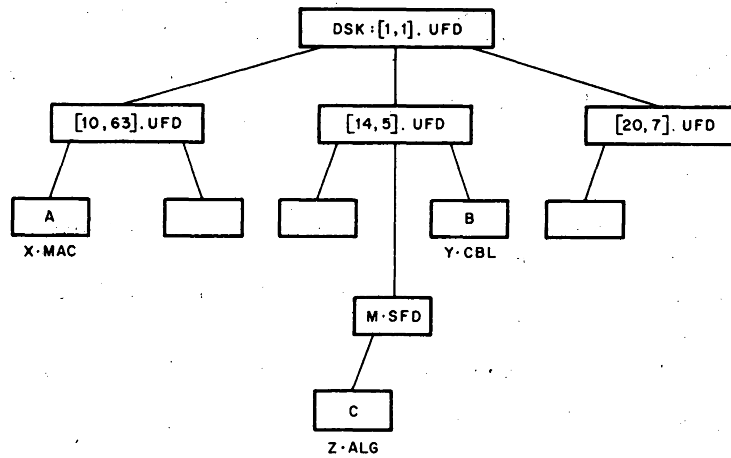


Figure 6-6 Directory Paths on a Single File Structure

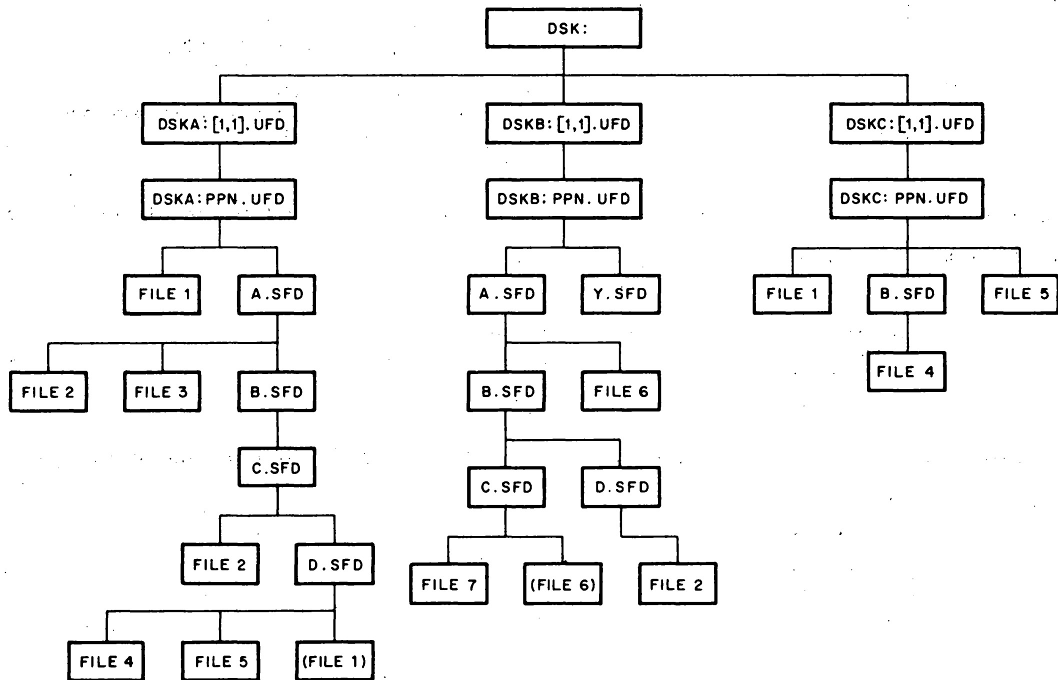


Figure 6-7 Directory Paths on Multiple File Structures

C) LOOKUP DSKB: FILE2, or LOOKUP DSKC: FILE2 fails.

D) ENTER DSK: FILE9 receives an error since no file structure has both the no-create bit off and the directory structure [PPN, A, B, C, D].

E) ENTER DSKA: FILE1 creates the file at the end of the path on DSKA (the file designated by (FILE1) in diagram).

If the default path is [PPN, A, B, C]:

A) ENTER DSK: FILE6 creates DSKB: FILE6 [PPN, A, B, C] (the file designated by (FILE6) in diagram).

B) ENTER DSK: FILE2 supersedes FILE2 in DSKA: [PPN, A, B, C]

C) LOOKUP DSK: FILE4 fails.

D) ENTER DSK: FILE7 supersedes FILE7 in DSKB: [PPN, A, B, C].

4) The user defines the following path.

```
        MOVE 1, [XWD 5,A]
        PATH. 1,
        error
        MOVE 1, [XWD 3,B]
        PATH. 1,
        error

A:  -2                    Define the default directory path
     2                    Scanning is in effect
    10,63                 The UFD [10,63]
    NAME                  The SFD [NAME]
     0                    The default path is [10,63,NAME]

B:  -3                    Define an additional path.
     3                    Both experimental SYS and SYS are searched.
    10,7                  The user library is [10,7]
```

If the user is logged in as [10,10] and does a LOOKUP DSK: FILTST, the following directories are searched in order:

```
        [NAME.SFD]    }
        [10,63. UFD]  }    job's search list.
        [10,7. UFD]   }
        [1,5. UFD]    }    system's search list.
        [1,4. UFD]    }
```

If the user is logged in as [10,10] and does a LOOKUP DSK: PRJFIL [10,155], the following directories are searched:

```
        [10,155. UFD] }
        [10,7. UFD]   }    job's search list
        [1,5. UFD]    }
        [1,4. UFD]    }    system's search list
```

6.2.9.2 USETI and USETO UUOs - The function of these UUOs is to notify the disk service routines that a particular relative block (instead of the next block in sequence) is to be used on the following INPUT or OUTPUT on the specified channel. USETI and USETO do not perform I/O; they simply change the current position of the file. Note that each INPUT or OUTPUT also logically advances the file; therefore, to reread or rewrite the same block a USETI (or USETO) must be given before each INPUT (or OUTPUT).

Since the monitor reads (writes) as many buffers as it can on INPUT (OUTPUT), it is difficult to determine which buffer the monitor is processing when the USETI (USETO) is given. Thus, the INPUT (OUTPUT) following the USETI (USETO) may not read (write) the buffer containing the block specified with the USETI (USETO). However, a single buffer ring reads (writes) the desired block since the device must stop after each INPUT (OUTPUT). Alternatively, if bit 30 of the status word (IO.SYN) is set via an INIT, OPEN, or SETSTS UUO, the device stops after each bufferful of data on an INPUT (OUTPUT) so that the USETI (USETO) will apply to the buffer supplied on the next INPUT (OUTPUT). The calls are:

USETI D,N and USETO D,N

where D is the channel number, and N designates a block relative to the beginning of the file. N can be in the following ranges:

N	Block Represented
$1\text{-}777777_8$	Blocks of the file
0	Prime (1st) RIB
$-2,\ldots,-10_8$	Extended (2nd to the 8th) RIB[1]
-1	Last block accessed (USETO) or end of file (USETI).

Note that the 18-bit effective address used for N is interpreted as both an unsigned positive integer and a signed (2's complement) integer. This is required since, with extended RIBs, there can be more than 377777_8 (largest positive signed integer) blocks in a file. The exact interpretation of N depends upon the context of the USETI/USETO (i.e., reading, writing, updating).

When reading or writing a file, USETI precedes an INPUT and USETO precedes an OUTPUT (i.e., USETI is illegal for a non-privileged program unless a LOOKUP has been done, and USETO is illegal for a non-privileged program unless an ENTER has been done). However, there are special cases when updating a file (both a LOOKUP and an ENTER have been done) when USETI may be followed by an OUTPUT and USETO may be followed by an INPUT. The action performed on a USETI or USETO depends on the value of N.

When N is a block number less than or equal to the current size of the file in blocks (i.e., N is a block that has been previously written), USETI or USETO points to block N in order to read or write that block on the next INPUT or OUTPUT.

---

[1] The number of extended RIBs allowed on the system can be changed with MONGEN and can be obtained from a GETTAB table (.GTLVD, item 23). Extended RIBs depend on FTMRIB which is normally off in the DECsystem-1040.

When N is a block number greater than the current size of the file in blocks, USETI followed by an INPUT receives the end-of-file return (e.g., if the file is 5 blocks long, USETI with n=7 receives the end of file return). On a USETO followed by an OUTPUT, the monitor allocates the intervening blocks, writes zeroes in the first new block up to block N-1, and then writes block N. For example, if the file is 2 blocks long, USETO with n=4 writes zeroes in block 3 and the data on the OUTPUT in block 4. If the number of blocks requested cause the disk to be filled or the user's quota to be exceeded, as many blocks as allowed will be allocated and the IO.BKT bit will be set in the status word. In addition, in update mode, USETI followed by an OUTPUT appends the data to the end of the file (i.e., makes the file larger). USETO followed by an INPUT allocates and zeroes the first new block up to block N-1 and then receives the end-of-file return.

When N=0 on reading, writing, and updating, USETI and INPUT read the prime RIB, and USETO and OUTPUT receive the IO.BKT error. In addition, in update mode, USETO and INPUT read the prime RIB, and USETI and OUTPUT receive the IO.BKT error.

When N=-2 to -10_8, a USETI and INPUT read the indicated extended RIB (-2 is the 2nd RIB,..., -10_8 is the 8th RIB). USETO followed by an OUTPUT attempts to allocate a large number of blocks (since N is interpreted as an unsigned integer) and therefore is not recommended because the user's disk quota will probably be exceeded.

When N=-1, USETO and OUTPUT rewrite the last block in which I/O was performed. USETI and INPUT receive the end of file return. In addition, in update mode, USETI followed by OUTPUT appends the data to the end of the file, and USETO and INPUT read the last block in which I/O was performed.

The user can append data to the last block of an append-only file by specifying a USETO followed by an OUTPUT to the last block.[1] The monitor then reads the block (of N words) into a monitor buffer, copies words N+1 through 200 from the user's buffer into the monitor buffer, and rewrites the block. The current length of the block can be obtained from the LOOKUP/ENTER block. It is not necessary to read the last block of the file before appending to it because the data already existing in the block is not changed.

When appending data to the last block of a file, the IO.BKT bit is set and no output is done if

1) Any block before the last block is written.
2) The last block already contains 200 words.
3) Fewer blocks are written than the current size of the block.

If the last block is written with a buffer-mode OUTPUT, the size of the last block becomes 200 words, and therefore, cannot be appended to.

Append-only files can be read only if FTAIR is on. Note that BASIC stores data at the beginning of files that it must read and therefore, to run BASIC, FTAIR must be turned on.

---

[1] This feature depends on FTAPLB, which is normally off in the DECsystem-1040. Therefore a new block must be written in order to append to a file.

If no previous LOOKUP or ENTER has been done, these UUOs are considered to be super-USETI and super-USETO which are available only to privileged programs. If the program is non-privileged, super-USETI and super-USETO cause the IO.BKT bit to be set in the status word. These privileged UUOs are documented in UUOPRV.RNO in the DECsystem-10 Software Notebooks.

6.2.9.3 SEEK UUO[1] - This UUO, when used in conjunction with USETI and USETO, allows user programs control over the time at which positioning operations occur. Following a regular USETI or USETO, positioning is to the cylinder containing the requested relative block within a file. Following a super-USETI or super-USETO, positioning is to the cylinder containing the specified disk block.

The call is:

```
        SEEK AC,        ; or CALLI D, 56
        return
```

D specifies a software channel number. The SEEK UUOs are honored by the monitor only if the unit for which they are issued is idle. If the unit is in any other state, the SEEK UUO is a no-operation.

SEEK UUOs issued for public file structures are treated in the same way as private file structures. This allows users to debug programs using a public disk pack and later run the same programs using a private disk pack.

The following is proper UUO sequence for issuing a SEEK.

For output

a. USETO to select a block (relative or actual)
b. SEEK to request positioning
c. computations
d. OUTPUT to request actual output

For input

a. USETI to select a block (relative or actual)
b. SEEK to request positioning
c. computations
d. INPUT to request actual input.

6.2.9.4 RESET UUO - This UUO causes files that are in the process of being written, but have not been CLOSEd or RELEASed, to be deleted; the space is reclaimed. If a previous version of the file with the same name and extension existed, it remains unchanged on the disk (and in the UFD). If the programmer wishes to retain the newly created file and to delete the older version, he must CLOSE or RELEASe the file before doing a RESET UUO.

---

[1]This UUO depends on FTDSEK which is normally off in the DECsystem-1040.

6.2.9.5 DEVSTS UUO - After each interrupt, FILSER stores the results of a CONI in the DEVSTS word of the device data block. The DEVSTS UUO is used to return the contents of the DEVSTS word to the user (refer to Paragraph 4.10.1).

6.2.9.6 CHKACC UUO[1] - This UUO allows programs to check the user's access to a particular file. The call is:

```
MOVE AC, [EXP LOC]
CHKACC AC,                  ;or CALLI AC, 100
error return
normal return
```

The LH of LOC contains the code for the type of access to be checked and the RH of LOC contains a 9-bit protection field. If the access code contained in the LH of LOC is greater or equal to 7, then the RH of LOC is interpreted as UFD privilege bits. LOC+1 contains the project-programmer number of the directory, and LOC+2 contains the project-programmer number of the user.

The type of access to be checked is represented by one of the following codes:

0	.ACCPR	Change protection.
1	.ACREN	Rename.
2	.ACWRI	Write.
3	.ACUPD	Update.
4.	.ACAPP	Append.
5	.ACRED	Read.
6	.ACEXO	Execute only.
7	.ACCRE	Create in UFD.
10	.ACSRC	Read directory as a file.

The error return is given if the UUO is not implemented. On a normal return, AC contains 0 if access is allowed or -1 if access if not allowed.

6.2.9.7 STRUUO AC, or CALLI AC, 50 - This UUO manipulates file structures and is intended primarily for monitor support programs.

The call is:

```
MOVE AC, [XWD N, LOC]
STRUUO AC,                  ;or CALLI AC, 50
error return                ;AC contains an error code
normal return               ;AC contains status information
```

N is the number of words in the argument list starting at location LOC. For the functions with a fixed length argument list, N may be 0.

The first word of the argument list specifies the function to be performed. Function 0 (.FSSRC) is the only unprivileged function; the remaining functions are available only to jobs logged-in under [1,2]

---

[1] This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

or to programs running with the JACCT bit set. Refer to the Specifications section of the DECsystem-10 Software Notebooks for a complete description of the privileged functions and their appropriate error codes.

The present functions are as follows:

Function	Name	
0	.FSSRC	Define a new search list for this job. This is the only unprivileged function.
1	.FSDSL	Define a new search list for any job or for the system. Privileged function.
2	.FSDEF	Define a new file structure. Privileged function.
3	.FSRDF	Redefine an existing file structure. Privileged function.
4	.FSLOK	Prevent any further new INITs, ENTERs, or LOOKUPs. Privileged function.
5	.FSREM	Remove file structure from system. Privileged function.
6	.FSULK	Test and set UFD interlock. Privileged function.
7	.FSUCL	Clear UFD interlock. Privileged function.
10	.FSETS	Simulate disk hardware errors. Privileged function.

6.2.9.7.1 Function 0 .FSSRC - This function allows a new file structure search list to be specified for the job issuing the UUO. The call is:

```
        MOVE AC, [XWD N, LOC]
        STRUUO AC,
        error return
        normal return
LOC:    0                              ;.FSSRC
LOC+1:  first file structure name
LOC+2:  0
LOC+3:  status bits
LOC+4:  second file structure name
LOC+5:  0
LOC+6:  status bits
  .
  .
  .
```

The argument list consists of word triplets, which specify the new search list order to replace the current search list. The current search list may be determined with the JOBSTR UUO. The first word contains a left-justified file structure name in SIXBIT. The second word is not used at present. The third word contains the following status bits:

Bit 0 = 1 if software write-protection is requested for this file structure.

Bit 1 = 1 if files are not to be created on this file structure unless the specific file structure is specified in an ASSIGN command or in an INIT or OPEN UUO.

The user may use the MOUNT command to add a new file structure name to his search list. The MOUNT program

a. Requests the file structure to be mounted (if it is not already mounted).

b. Creates a UFD for the user if he has a logged-in quota in file SYS: QUOTA.SYS on that file structure.

A user cannot create files on a file structure unless he or the project-programmer number specified has a UFD on that file structure. However, by using the .FSSRC function, the user may add a file structure name to his search list if the file structure is mounted and either the user has a UFD for that file structure or he does not want to write on that file structure. If the user attempts to delete a file structure name from his search list by the .FSSRC function, the monitor moves the file structure name from the active search list to the passive search list. The DISMOUNT command must be used to remove the file structure from the active or passive search list. The DISMOUNT command causes the mount count to be decremented, signifying that the user is finished with the file structure, and checks that the user has not exceeded his logged-out quota on the file structure.

Table 6-7
.FSSRC Error Codes

Symbol	Code	Explanation
FSILF%	0	Illegal function code.
FSSNF%	1	One or more file structures not found.
FSSSA%	2	One or more file structures single access only.
FSTME%	4	Too many entries in search list.
FSRSL%	17	File structure is repeated in a search list definition.

6.2.9.8 JOBSTR AC, or CALLI AC, 47[1] - This UUO returns the next file structure name in the job's search list along with other information about the file structure. Programs like DIRECT use this UUO to list a user's directory correctly and specify in which file structures the files occur, as well as the order in which they are scanned.

The call is:

```
MOVE AC, [XWD N,LOC]
JOBSTR AC,                    ;or CALLI AC, 47
error return
normal return
```

LOC is the address of the N-word argument block. When the UUO is called, the first word should be one of the following:

a. -1 to return the first file structure name in the search list.

b. a file structure name to return the next file structure following the specified name.

c. 0 to return the file structure name immediately following the FENCE. Refer to Paragraph 6.2.7.

---

[1] In the DECsystem-1040, FTSTR is normally off so that there is only one file structure on the system. However, this UUO is implemented and returns the file structure name or -1.

On return, the first word contains:

a. the first file structure name in the search list if -1 was specified.

b. the next file structure name appearing after the specified name or after the FENCE (if 0 was specified).

c. 0 if the item after the specified name is the FENCE.

d. -1 if there are no more file structure names in the search list, or the search list is empty.

The second word contains 0 (reserved for a future argument), and the third word contains status bits. Current status bits are:

Bit 0 = 1 if software write protection is in effect for this job.

Bit 1 = 1 if files are not to be created on this file structure, when a multiple file structure name is specified in an INIT or OPEN UUO. Files can be created if a specific file structure or physical unit is specified.

The following is an example of reading a job's search list.

```
       SETOM LOC              ;place -1 in LOC to get 1st
                              ;name in search list.
LOOP:MOVEI AC, LOC            ;set up AC.
       JOBSTR AC,             ;do the UUO.
       JRST ERROR             ;error return.
       MOVE AC, LOC           ;get file structure name returned.
       JUMPE AC, FENCE        ;jump if it is the FENCE.
       AOJE AC, END           ;jump if end of search list (-1).
       .                      ;LOC has next file structure name.
       .
       .
       JRST LOOP              ;repeat with next file structure name.
LOC:   -1                     ;file structure name.
       0                      ;reserved for future use.
       0                      ;status bits.
```

6.2.9.9 GOBSTR AC, or CALLI AC, 66 - This UUO returns successive file structure names in the search list of either an arbitrary job or the system. The GOBSTR UUO is a generalization of the JOBSTR UUO (see Paragraph 6.2.9.8). It is a privileged UUO unless information being requested is either about the system search list or the jobs logged-in under the same project-programmer number as the calling job's number. For example, the KJOB program needs information about the search lists of jobs logged in under the same project-programmer number as the job logging out. The privilege bits required are either JP.SPA (bit 16) or JP.SPM (bit 17) of the privilege word (.GTPRV).

The call is:

```
       MOVE AC, [XWD N,LOC]
       GOBSTR AC,             ;or CALLI AC, 66
       error return           ;AC contains an error code
       normal return
```

When the UUO is called, AC specifies the length (N) and address (LOC) of an argument list. N may be 0, 3, 4, or 5 where N = 0 has the same effect as N = 3. Only the arguments included by N(LOC, LOC+1,...,LOC+N-1) are used or returned. The argument list is as follows:

```
LOC:  job number                     ;job whose search
                                     ;list is desired.
      XWD proj, prog                 ;project-programmer
                                     ;number of above job.
      SIXBIT /file structure name/   ;or -1 or 0.
      0                              ;currently unused.
      Status                         ;status bits are the same
                                     ;as in JOBSTR UUO.
```

If the job number = -1, the number of the job issuing the UUO is used. If the job number = 0, the given project-programmer number is ignored and the system search list is used. When the given project-programmer number is -1, the project-programmer number of the job issuing the UUO is used.

On an error return, AC contains one of the following error codes:

Code	Meaning
3	If LOC+2 is not 0, -1, or a file structure name in jobs search list.
6	If job number (LOC) and project-programmer number (LOC +1) do not correspond.
10	If job issuing the UUO is not privileged.
12	If the length specified for the argument list is not valid.

6.2.9.10 SYSSTR AC, or CALLI AC, 46 - This UUO provides a simple mechanism to obtain all the file structure names in the system. The proper technique to access all files in all UFDs is to access the MFD on each file structure separately. Monitor support programs use this UUO to access all the files in the system.

The call is:

```
MOVEI AC, 0 or the last value returned by previous SYSSTR
SYSSTR AC,                          ;or CALLI AC, 46
error return
normal return
```

An error return is given if either

a. The UUO is not implemented
b. The argument is not a file structure name

On a normal return, the next public or private file structure name in the system is returned in AC. A return of 0 in AC on a normal return means that the list of file structure names has been exhausted. If

0 is specified as an argument, the first file structure name is returned in AC. The argument cannot be a physical disk unit name or a logical name.

6.2.9.11 SYSPHY AC, or CALLI AC, 51[1] - This UUO returns all physical disk units in the system. The SYSPHY UUO is similar to the SYSSTR UUO (see Paragraph 6.2.9.10).

The call is:

```
        MOVEI AC, 0 or the last unit name returned by previous SYSPHY
        SYSPHY AC,                    ; or CALLI AC, 51
        error return                  ;not implemented or not a physical disk
        normal return                 ;unit name
```

On the first call AC should be 0 to request the return of the first physical unit name. On subsequent calls, AC should contain the previously returned unit name.

An error return is given if AC does not contain a physical disk unit name as zero. On a normal return, the next physical unit name in the system is returned in AC. A return of 0 in AC indicates that the list of physical units has been exhausted.

6.2.9.12 DEVPPN AC, or CALLI AC, 55[1] - This UUO allows a user program to obtain the project-programmer number associated with a disk device. The device argument given can be a logical device name, a physical device name, or one of the special devices called ersatz devices. (Refer to DECsystem-10 Operating System Commands for the list of system devices.)

When the UUO is called, AC must contain either the device name as a left-justified SIXBIT quantity, or the channel number of the device as a right-justified quantity.

The call is:

```
        MOVE AC, [SIXBIT /DEV/]       ;or MOVEI AC, channel number
        DEVPPN AC,                    ;or CALLI AC, 55
        error return
        normal return
```

The error return is taken if:

[1]This UUO depends on FT5UUO which is normally off in the DECsystem-1040.

a. The UUO is not implemented; therefore, the contents of AC remain the same on return. In this case, obtain the appropriate project-programmer number as follows:

1. For the user's area, use the GETPPN UUO (refer to Paragraph 3.6.2.3).
2. For the special ersatz devices, use the default project-programmer numbers appearing in the following list.

Device	Project-programmer Number
ALL:	User's project-programmer number
BAS:	[5,1]
COB:	[5,2]
DSK:	User's project-programmer number
HLP:	[2,5]
LIB:	Set by each user
MXI:	[5,3]
NEW:	[1,5]
OLD:	[1,3]
SYS:	[1,4]

b. The device does not exist or the channel is not INITed; therefore, zero is returned in AC.

c. The device is not a disk; the user's project-programmer number is returned in AC.

If a legal device is specified, the normal return is given and the project-programmer number associated with the device is returned in AC. However, if the user has device NEW: enabled in his search list and device SYS: is given as the argument to the DEVPPN UUO, the project-programmer number returned is [1,5].

The following is an example for reading a UFD if either device SYS or the user's area is specified.

```
        MOVEI   A,16            ;GET MFD PROJECT-PROGRAMMER NUMBER
        GETTAB  A,              ;NO CHANGE IF NO GETTAB
          MOVE  A,[1,,1]        ;  IN CASE OF LEVEL C
        MOVEM   A,MFDPPN        ;STORE MFD DIRECTORY NUMBER

        MOVE    A,DEVNAM        ;GET DEVICE NAME TYPED BY USER
        MOVEM   A,MODE+1        ;STORE FOR OPEN
        DEVPPN  A,              ;GET PROJECT PROGRAMMER NUMBER
          JRST  GETPPX          ;  NOT IMPLEMENTED OR NO SUCH DEVICE

;BACK HERE WITH IMPLIED PPN IN A

GOTPPN: MOVEM   A,PPN           ;STORE PPN IMPLIED BY DEVICE NAME

        OPEN    I,MODE          ;TRY TO OPEN DEVICE
          JRST  ERROR           ;NOT AVAILABLE
        LOOKUP  I,PPN           ;TRY TO LOOKUP UFD
          JRST  ERROR           ;NOT THERE
        IN      I,              ;READ FIRST BLOCK
          JRST  USEIT           ;GO DO USEFUL WORK
        JRST    ERROR           ;ERROR OR END OF FILE

;HERE IF DEVPPN FAILS

GETPPX: CAMN    A,[SIXBIT /SYS/]  ;SEE IF DEVICE NAME SYS:
        JRST    GETPPS          ;YES--GO HANDLE SYS:
        GETPPN  A,              ;NO--GET OWN PPN
          JFCL                  ;(IN CASE OF JACCT)
        JRST    GOTPPN          ;OK--PROCEED ABOVE

GETPPS: MOVE    A,[1,,16]       ;FIND SYS: PPN
        GETTAB  A,              ;FROM MONITOR TABLES
          MOVE  A,[1,,1]        ;(IN CASE OF LEV. C)
        JRST    GOTPPN          ;OK--PROCEED ABOVE

MODE:   14              ;BINARY READ
        0               ;DEVICE NAME
        0,,INBUFH       ;BUFFER HEADER

PPN:    0               ;DIRECTORY NAME
        SIXBIT /UFD/    ;EXTENSION
        0
MFDPPN: 1,,1            ;LOOKUP UFD IN MFD
```

6.2.9.13 DSKCHR AC, or CALLI AC, 45 - The disk characteristics UUO provides necessary information for allocating storage efficiently on different types of disks. Most programs are able to use the generic device name DSK rather than special disk names; however, this UUO is needed by special monitor support programs.

This UUO accepts, as arguments, names of file structures (e.g., DSKA), types of controllers (e.g., DP), controllers (e.g.,DPA), logical units (e.g., DSKA3), physical disk units (e.g., DPA3), or logical device names (e.g., ALPHA). If the argument in LOC specifies more than one unit, the values returned in AC are for the first unit of the specified set. If the argument specifies more than one file structure (i.e., DSK or logical device name for disk), the first unit of the first file structure is returned.

The call is:

```
        MOVE AC, [XWD+N,LOC]      ;N is the number of locations
                                  ;of arguments and values starting
                                  ;at location LOC
        DSKCHR AC,                ;or CALLI AC, 45
        error return              ;not a disk
        normal return
```

On a normal return, AC contains status information in the left half and configuration information in the right half. The left half bits have been chosen so that the normal state is 0.

Name	Bit	Explanation
DC.RHB	Bit 0 = 1	The monitor must reread the home block before the next operation to ensure that the pack ID is correct. The monitor sets this bit when a disk pack goes off-line.
DC.OFL	Bit 1 = 1	The unit is off-line.
DC.HWP	Bit 2 = 1	The unit is hardware write-protected.
DC.SWP	Bit 3 = 1	The unit belongs to a file structure that is write-protected by software for this job.
DC.SAF	Bit 4 = 1	The unit belongs to a single-access file structure.
DC.ZMT	Bit 5 = 1	The unit belongs to a file structure with a mount count that has gone to zero (i.e., no one is using the file structure). Available in 5.02 monitors and later.
	Bit 6 = 1	Reserved for the future.
DC.STS	Bits 7 and 8	Unit status
.DCSTD	= 11	The unit is down.
.DCSTN	= 10	No pack is mounted.
	= 01	Reserved for the future.
.DCSTP	= 00	A pack is mounted.
DC.MSB	Bit 9 = 1	The unit has more than one SAT block.
DC.NNA	Bit 10 = 1	The unit belongs to a file structure for which the operator has requested no new INITs, LOOKUPs, or ENTERs; set by privileged STRUUO function.
DC.AWL	Bit 11 = 1	The file structure is write-locked for all jobs.
	Bits 12 - 14	Reserved for future expansion.

Name	Bit	Explanation
DC.TYP	Bits 15 - 17	The code identifies which type of argument was passed to the monitor in location LOC.
DC.DCN	Bits 18 - 20	Data channel number that software believes hardware is connected to; first data channel is 0.
DC.CNT .DCCFH .DCCDP	Bits 21 - 26 =1 =2	Controller type: FH (Burroughs disk, Bryant drum) controller RC 10 DP (Memorex disk packs) controller RP10
DC.CNN	Bits 27 - 29	Controller number; first controller of each type starts at 0 (e.g., DPA = 0, DPB = 1)
DC.UNT	Bits 30 - 32	Unit type; a controller-dependent field used to distinguish various options of a unit on its controller. If bits 21 - 26 and bits 30 - 32 then type is 1 0 RD10 Burroughs disk (.DCUFD) 1 1 RM10B Bryant drum (.DCUFM) 2 1 RP02 disk pack (.DCUD2) 2 2 RP03 disk pack (.DCUD3)
DC.UNN	Bits 33 - 35	Physical unit number within controller; first unit is 0

The user program supplies in location LOC a left-justified, SIXBIT disk name which may be one of the following:

.DCTDS	0	generic disk name
.DCTAB	1	subset of file structures because of file structure abbreviation
.DCTFS	2	file structure name
.DCTUF	3	unit within a file structure
.DCTCN	4	controller type
.DCTCC	5	controller
.DCTPU	6	physical disk unit name

or a logical name for one of the above assigned by the ASSIGN or MOUNT monitor command.

On a normal return, the monitor returns values in the following locations:

LOC+1 (.DCUFT) The number of blocks left of the logged-in job quota before the UFD of the job is exhausted on the unit specified in LOC. If negative, the UFD is overdrawn. If the negative number is 400000 000000, the UFD has not been accessed since LOGIN; therefore, the monitor does not know the quota.

LOC+2 (.DCFCT) The number of blocks on a first-come first-served basis left for all users in the file structure.

LOC+3 (.DCUNT) The number of blocks left for all users on the specified unit.

LOC+4 (.DCSNM) The file structure name to which this unit belongs.

LOC+5 (.DCUCH) Characteristic sizes

a. Bits 0-8 are the number of blocks/cluster (DC.UCC)
b. Bits 9-17 are the number of blocks/track (DC.UCT)
c. Bits 18-35 are the number of blocks/cylinder (DC.UCY) (see Appendix F).

LOC+6 (.DCUSZ)	The number of 128-word blocks on the specified unit.
LOC+7 (.DCSMT)	The mount count for the file structure. The mount count is the number of jobs that have done a MOUNT command for this file structure without executing a DISMOUNT command; it is a use count.
LOC+10 (.DCWPS)	The number of words containing data bits per SAT block on this unit.
LOC+11 (.DCSPU)	Number of SAT blocks per unit.
LOC+12 (.DCK4S)	Number of K allocated for swapping.
LOC+13 (.DCSAJ)	Zero if none or more than one job has this file structure mounted. XWD -1,,n if only job n has file structure mounted but it is not single access. XWD 0,,n if job has file structure mounted and it is single access.
LOC+14 (.DCULN)	The unit's logical name (e.g., DSKB0).
LOC+15 (.DCUPN)	The unit's physical name (e.g., DPA0).
LOC+16 (.DCUID)	The unit's ID (e.g., 2RP003).
LOC+17 (.DCUFS)	The first logical block used for swapping on this unit.

6.2.9.14 DISK. AC, or CALLI AC, 121 - The DISK. UUO is a general-purpose call designed for setting and examining parameters of the disk and file systems. Its present function allows the user to assign a priority for disk operations (data transfers and head positionings) either for a user I/O channel or for his job (all I/O channels). Therefore, when a disk operation is initiated, the request with the highest priority is selected. If there is more than one request with the same priority, the one most satisfying disk optimization is chosen (refer to Chapter 7).

The range of permissible disk priorities is -3 to +3 with 0 being the normal timesharing priority. Thus, a job can request a lower than normal priority (e.g., when the job is a background job). The maximum priority (greater than 0) that the user is allowed to assign is set by bits 1 and 2 (JP.PRI) of the privilege word .GTPRV.

The call is:

```
MOVE AC, [XWD function, ADR]
DISK. AC,                          ;or CALLI AC, 121
error return
normal return
```

The present function is

Function	Name	Description
0	.DUPRI	Set the disk priority

ADR contains, in the right half, the desired priority (-3 to +3) and in the left half, one of the following:

LH (ADR) = n	Sets the priority for channel n (n is from 0 to 17).
LH (ADR) = -1	Sets the priority for all channels OPENed by the job.
LH (ADR) = -2	Sets the priority for the entire job.

When a priority is set for a channel, it overrides any priority set for the job and remains in effect until the channel is RELEASed. When set for the job, the priority remains in effect until RESET by another DISK. UUO or the SET DSK PRI command (refer to DECsystem-10 Operating System Commands).

6.2.9.15 Simultaneous Supersede and Update - Files that may be simultaneously superseded or updated by several different users should be treated with care. The problem arises when one user has a copy of information to be superseded by another user. For example, file F contains a count of the number of occurrences of a certain event. The count is 10 at a given time. When two users observe separate instances of the event, each tries to increment the count.

Supersede - Incorrectly

```
        Job 1                       Job 2
      LOOKUP A,F
                                  LOOKUP C, F
      READ COUNT (=10)            READ COUNT (=10)
      ADD 1 (=11)                 ADD 1 (=11)
                                       .
      ENTER B, F                       .
      WRITE OUT (=11)             ENTER D, F          (Fail)
      CLOSE B,                         .
                                  ENTER D, F          (Succeed)
                                  WRITE OUT (=11)
                                  CLOSE D,
```

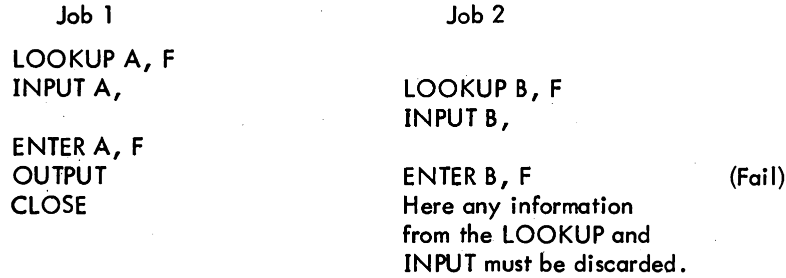
In this example, job 2 ignored job 1's increment.

Supersede - Correctly

```
        Job 1                       Job 2
      ENTER B, F
      LOOKUP A, F
          .                       ENTER D, F          (Fail)
      INPUT A, (=10)
      ADD1 (=11)                       .
      OUTPUT B, (=11)                  .
      CLOSE B,                         .
                                  ENTER D, F          (Succeed)
                                  LOOKUP C, F
                                  INPUT C, (=11)
                                  ADD1 (=12)
                                  OUTPUT D, (=12)
                                  CLOSE D,
```

In this example, both jobs performed the ENTER FIRST; therefore, incorrect copies were not made and the increment of each job was recorded properly.

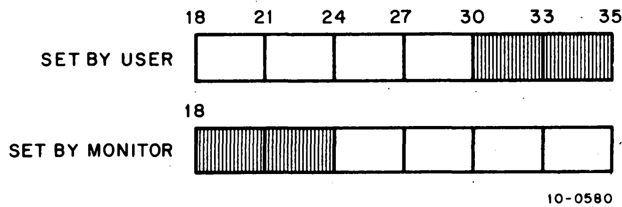
The similar problem with a update can be avoided by never using the information returned by the LOOKUP:



Job 1	Job 2	
LOOKUP A, F		
INPUT A,	LOOKUP B, F	
	INPUT B,	
ENTER A, F		
OUTPUT	ENTER B, F	(Fail)
CLOSE	Here any information from the LOOKUP and INPUT must be discarded.	

6.2.10 File Status (refer to Appendix D)

The file status of the disk is shown below.



Bit 18 - IO.IMP

a. INPUT UUO attempted on a read-protected file

b. INPUT UUO when no LOOKUP was done (or super-USETI/USETO previously attempted by nonprivileged user)

c. OUTPUT UUO when no ENTER was done (or super-USETI/USETO previously attempted by nonprivileged user)

d. Software-detected checksum error

e. Software-detected redundancy error in SAT block or RIB, or

f. Buffered mode I/O attempted after super-USETI/USETO.

g. OUTPUT UUO attempted on a write-locked unit.

Bit 19 - IO.DER Search error, power supply failure

Bit 20 - IO.DTE Disk or data channel parity error. Checksum failure on INPUT.

Bit 21 - IO.BKT

a. Quota is exhausted (past overdrawn)

b. File structure is exhausted

c. RIB is full

d. Super-USETI/USETO block is too large for the file structure

(continued on next page)

Bit 21 - IO.BKT (cont)

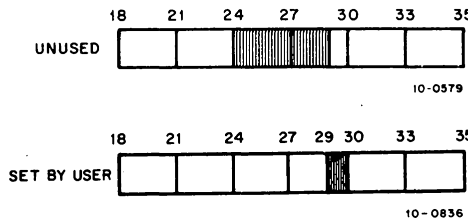
e. More than 777777 blocks were read with one super-USETI/USETO

f. Block number specified is too low for writing in a file that has an append protection code (4). The block number must be greater than the current highest block number of the file. Not set on a USETI or USETO.

g. A super-USETI or USETO was issued by a non-privileged program.

Bit 22 - IO.EOF — EOF encountered on INPUT. No special character appears in the buffer.

Bit 23 - IO.ACT — Device is active



Bit 29 - IO.WHD — Write disk pack headers

6.2.11 Disk Packs

A disk pack system combines disk and the DECtape features. Some packs (similar to individual DEC-tapes) are designed to be private, assignable, and removable. The other packs make up part or all of the public disk storage area where system programs and user files are stored. These disk packs belong to file structures in the storage pool and cannot be assigned to any single user. The system library and shared on-line storage is maintained and swapping storage is assigned within the public disk pack area.

The only distinction between public and private packs is that private packs are intended to be removed from the system during regular operation. Public packs usually stay on-line all the time. However, the file structure format for public and private disk packs is identical.

User programs can exercise much greater control over private packs. For example, a program may attempt to position the arms of disk packs in anticipation of future I/O (refer to Paragraph 6.2.9.3). This capability is useful to a program that is aware of the contents of a disk and is able to use this information to optimize positioning. The program may also specify the position of files on the disk by using the allocate arguments of the extended LOOKUP, ENTER, and RENAME UUOs.

Private packs may be accessed by more than one job (multi-access) or restricted to only one job (single access). To access a private file structure, the user must type the MOUNT monitor command. If the private file structure is already mounted, on-line, and multi-access, the user receives an immediate

response and may start using the private pack. When the user is finished using the private file structure, he should type the DISMOUNT monitor command. If no other job is using the file structure, a message is typed to the operator informing him that the drives belonging to the file structure are free.

6.2.11.1 Removable File Structures - All file structures are designed as if they could be removed from the system; therefore, disk packs are handled the same as other types of disks.

6.2.11.2 Identification - Disk packs have identifying information written on the home block, a block on every unit identifying the file structure to which the unit belongs and its position within the file structure. Part of this information is the pack ID, a one- to six-character SIXBIT name uniquely identifying the disk pack. The MOUNT and OMOUNT programs check that the operator has mounted the proper packs by comparing the pack ID in the home block with the information stored in the system administration file STRLST.SYS.

6.2.11.3 IBM Disk Pack Compatibility - The data format of IBM disk packs has variable-length sectors and no sector headers. DEC format has fixed-length sectors (128 words) and specially written sector headers. Latency optimization is employed to improve system throughput (refer to Paragraph 7.3). DEC's significantly simpler hardware controller is used without reducing user capabilities.

To transfer data from a IBM pack system to a DEC pack system, a simple program in a higher-level language should be written for both machines. The program then reads the IBM disk pack on the IBM computer and writes the files onto magnetic tape. The magnetic tape is then transferred to a DEC computer and read by another program, which writes the files onto the DEC RP01 or RP02 packs.

6.3 SPOOLING OF UNIT RECORD I/O ON DISK

Devices capable of spooling (card reader, line printer, card punch, paper-tape punch, and plotter) have an associated bit in the job's .GTSPL word. If this bit is on when the device is ASSIGNed or INITed, the device is said to be in spool mode. While in this mode, all I/O for this device is intercepted and written onto the disk rather than onto the device. System spooling programs later do the actual I/O transfer to the device.

Spooling allows more efficient use of the device because users cannot tie it up indefinitely. In addition, since the spooling devices are generally slow and the jobs that are to be spooled are usually large, the jobs do not spend unnecessary time in core.

6.3.1 Input Spooling

If a LOOKUP is given after the INIT of the card reader, it is ignored and an automatic LOOKUP is done, using the filename given in the last SET CDR command and the filename extension of .CDR. After every automatic LOOKUP, the name in the input-name counter .GTSPL is incremented by 1 so that the next automatic LOOKUP will use the correct filename.

6.3.2 Output Spooling

If an ENTER is done, the filename specified is stored in the RIB in location .RBSPL so that the output spooler can label the output. Therefore, programs should specify a filename if possible.

If an ENTER is not done, an automatic ENTER is given, using a filename in the general form

xxxyyy.zzz

where xxx is a three-character name manufactured by the monitor to make the 9-character name unique.

yyy is (1) an appropriate station number Snn if a generic device name is INITed or (2) a unit number if a specific unit is INITed.

zzz is the generic name of the device-type (LPT, CDP, PTP, or PLT).

Output spooling should not concern the user because all requests are queued when the user logs off the system. The files are moved to the output queues before the logged-out quota is computed.

CHAPTER 7
MONITOR ALGORITHMS

7.1 JOB SCHEDULING

The number of jobs that may be run simultaneously must be specified in creating the DECsystem-10 Monitor. Up to 127 jobs may be specified. Each user accessing the system is assigned a job number.

In a multiprogramming system all jobs reside in core, and the scheduler decides what jobs should run. In a swapping system, jobs exist on an external storage device (usually disk or drum) as well as in core. The scheduler decides not only what job is to run but also when a job is to be swapped out onto the disk (drum) or brought back into core.

In a swapping system, jobs are retained in queues of varying priorities that reflect the status of the jobs at any given moment. Each job number possible in the system resides in only one queue at any time. A job may be in one of the following queues:

a. Run queues - for runnable jobs waiting to execute. (There are three run queues of different levels of priorities.)
b. I/O wait queue - for jobs waiting while doing I/O.
c. I/O wait satisfied queue - for jobs waiting to run after finishing I/O.
d. Sharable device wait queue - for jobs waiting to use sharable devices.
e. TTY wait queue - for jobs waiting for input or output on the user's console.
f. TTY wait satisfied queue - for jobs that completed a TTY operation and are awaiting action.
g. Stop queue - for processes that have been completed or aborted by an error and are awaiting a new command for further action.
h. Null queue - for all job numbers that are inactive (unassigned).

Each queue is addressed through a table. The position of a queue address in a table represents the priority of the queue with respect to the other queues. Within certain queues, the position of a job determines its priority with respect to the other jobs in the same queue. For example, if a job is first in the queue for a sharable device, it has the highest priority for the device when it becomes available. However, if a job is in an I/O wait queue, it remains in the queue until the I/O is completed. Therefore, in an I/O wait queue, the job's position has no significance. The status of a job changes each time it is placed into a different queue.

Each job, when it is assigned to run, is given a quantum time. When the quantum time expires, the job ceases to run and moves to a lower priority run queue. The activities of the job currently running may cause it to move out of the run queue and enter one of the wait queues. For example: when a currently running job begins input from a DECtape, it is placed in the I/O wait queue, and the input is begun. A second job is set to run while the input of the first job proceeds. If the second job then decides to access a DECtape for an I/O operation, it is stopped because the DECtape control is busy, and it is put in the queue for jobs waiting to access the DECtape control. A third job is set to run. The input operation of the first job finishes, making the DECtape control available to the second job. The I/O operation of the second job is initiated, and the job is transferred from the device wait queue to the I/O wait queue. The first job is transferred from the I/O wait queue to the highest priority run queue. This permits the first job to preempt the running of the third job. When the quantum time of the first job becomes zero, it is moved into the second run queue, and the third job runs again until the second job completes its I/O operations.

Data transfers also use the scheduler to permit the user to overlap computation with data transmission. In unbuffered modes, the user supplies an address of a command list containing pointers to relative locations in the user area to and from which data is to be transferred. When the transfer is initiated, the job is scheduled into an I/O wait queue where it remains until the device signals the scheduler that the entire transfer has been completed.

In buffered modes, each buffer contains a use bit to prevent the user and the device from using the same buffer at the same time (refer to Paragraph 4.3). If the user overtakes the device and requires the buffer currently being used by the device as his next buffer, the user's job is scheduled into an I/O wait queue. When the device finishes using the buffer, the device calls the scheduler to reactivate the job. If the device overtakes the user, the device is stopped at the end of the buffer and is restarted when the user finishes with the buffer.

Scheduling occurs at each clock tick (1/60th or 1/50th of a second) or may be forced at monitor level between clock ticks if the current job becomes blocked (unrunnable). The asynchronous swapping algorithm is also called at each clock tick and has the task of bringing a job from disk into core. This function depends on

a. The core shuffling routine, which consolidates unused areas in core to make sufficient room for the incoming job,
b. The swapper, which creates additional room in core by transferring jobs from core to disk.

Therefore, when the scheduler is selecting the next job to be run, the swapper is bringing the next job to be run into core. The transfer from disk to core takes place while the central processor continues computation for the previous job.

7.2 PROGRAM SWAPPING

Program swapping is performed by the monitor on one or more units of the system independent of the file structures that may also use the units. Swapping space is allocated and deallocated in clusters of 1K words (exactly); this size is the increment size of the memory relocation and protection mechanism. Directories are not maintained, and retrieval information is retained in core. Most user segments are written onto the swapping units as contiguous units. Swapping time and retrieval information is, therefore, minimized. Segments are always read completely from the swapping unit into core with one I/O operation. The swapping space on all units appears as a single system file, SWAP.SYS, in directory SYS in each file structure. This file is protected from all but privileged programs by the standard file protection mechanism (refer to Paragraph 6.2.3).

The reentrant capability reduces the demands on core memory, swapping space, swapping channel, and storage channel; however, to reduce the use of the storage channel, copies of sharable segments are kept on the swapping device. This increases the demand for swapping space. To prevent the swapping space from being filled by user's files and to keep swapped segments from being fragmented, swapping space is preallocated when the file structure is refreshed. The monitor dynamically achieves the space-time balance by assuming that there is no shortage of swapping space. Swapping space is never used for anything except swapped segments, and the monitor keeps a single copy of as many segments as possible in this space. (The maximum number of segments that may be kept may be increased by individual installations but is always at least as great as the number of jobs plus one.) If a sharable segment on the swapping space is currently unused, it is called a dormant segment. An idle segment is a sharable segment that is not used by users in core; however, at least one swapped-out user must be using the segment or it would be a dormant segment.

Swapping disregards the grouping of similar units into file structures; therefore, swapping is done on a unit basis rather than a file structure basis. The units for swapping are grouped in a sorted order, referred to as the active swapping list. The total virtual core, which the system can allocate to users, is equal to the total swapping space preallocated on all units in the active swapping list. In computing virtual core, sharable segments count only once, and dormant segments do not count at all. The monitor does not allow more virtual core to be granted than the system has capacity to handle.

When the system is started, the monitor reads the home blocks on all the units that it was generated to handle. The monitor determines from the home blocks which units are members of the active swapping list. This list may be changed at once-only time. The change does not require refreshing of the file structures, as long as swapping space was preallocated on the units when they were refreshed. All of the units with swapping space allocated need not appear in the active swapping list. For example: a drum and disk pack system should have swapping space allocated on both drum and disk packs. Then, if the drum becomes inoperable, the disk packs may be used for swapping without refreshing.

Users cannot proceed when virtual core is exhausted; therefore, FILSER is designed to handle a variety of disks as swapping media. The system administrator allocates additional swapping space on slower disks and virtually eliminates the possibility of exhausting virtual core; therefore, in periods of heavy demand, swapping is slower for segments that must be swapped on the slower devices. It is also undesirable to allow dormant segments to take up space on high-speed units. This forces either fragmentation on fast units or swapping on slow units; therefore, the allocation of swapping space is important to overall system efficiency.

The swapping allocator is responsible for assigning space for the segment the swapper wants to swap out. It must decide

a. Onto which unit to swap the segment.
b. Whether to fragment the unit if not enough contiguous space is available.
c. Whether to make room by deleting a dormant segment.
d. Whether to use a slower unit.

The units in the active swapping list are divided into swapping classes, usually according to device speed. For simplicity, the monitor assumes that all the units of class 0 are first followed by all the units of class 1. Swapping classes are defined when the file structures are refreshed and may be changed at once-only time.

When attempting to allocate space to swap out a low or high segment, the monitor performs the following:

Step	Procedure
1	The monitor looks for contiguous space on one of the units of the first swapping class.
2	The monitor looks for noncontiguous space on one of the units in the same class.
3	The monitor checks whether deleting one or more dormant segments would yield enough contiguous or noncontiguous space.

If all of these measures fail, the monitor repeats the process on the next swapping class in the active swapping list. If none of the classes yield enough space, the swapper begins again and deletes enough dormant segments to fragment the segment across units and classes. When a deleted segment is needed again, it is retrived from the storage device.

7.3 DEVICE OPTIMIZATION

7.3.1 Concepts

Each I/O operation on a unit consists of two steps: positioning and data transferring. To perform I/O, the unit must be positioned, unless it is already on a cylinder or is a non-positioning device. To position a unit, the controller cannot be performing a data transfer. If the controller is engaged in a data transfer, the positioning operation of moving the arm to the desired cylinder cannot begin until the data transfer is complete.

The controller ensures that the arms have actually moved to the correct cylinder. This check is called verification, and the time required is fixed by hardware. If verification fails, the controller interrupts the processor, and the software recalibrates the positioner by moving it to a fixed place and beginning again. When verification is complete, the controller reads the sector headers to find the proper sector on which to perform the operation. This operation is called searching. Finally, the data is transferred to or from the desired sectors. To understand the optimization, the transfer operation includes verification, searching, and the actual transfer. The time from the initiation of the transfer operation to the actual beginning of the transfer is called the latency time. The channel is busy with the controller for the entire transfer time; therefore, it is important for the software to minimize the latency time

The FILSER code, the routines that queue disk requests and make optimization decisions, handles any number of channels and controllers and up to eight units for each controller.[1] Optimization is designed to keep:

a. As many channels as possible performing data transfers at the same time.
b. As many units positioning on all controllers, which are not already in position for a data transfer.

Several constraints are imposed by the hardware. A channel can handle only one data transfer on one control at a time. Furthermore, the control can handle a data transfer on only one of its units at a time. However, the other units on the control can be positioning while a data transfer is taking place provided the positioning commands were issued prior to the data transfer. Positioning requests for a unit on a controller that is busy doing a data transfer for another of its units must be queued until the data transfer is finished. When a positioning command is given to a unit through a controller, the controller is busy for only a few microseconds; therefore, the software can issue a number of positioning commands to different units as soon as a data transfer is complete. All units have only positioning mechanism that reaches each point; therefore, only one positioning operation can be performed on a unit at the same time. All other positioning requests for a unit must be queued.

---

[1] Disk latency optimization depends on FTDOPT which is normally off in the DECsystem-1040. If this switch is off, all requests are handled on a first-come first-served basis.

The software keeps a state code in memory for each active file, unit, controller, and channel, to remember the status of the hardware. Reliability is increased because the software does not depend on the status information of the hardware. The state of a unit is as follows:

I	Idle; No positions or transfers waiting or being performed.
SW	Seek Wait; Unit is waiting for control to become idle so that it can initiate positioning (refer to Paragraph 6.2).
S	Seek; Unit is positioning in response to a SEEK UUO; no transfer of data follows.
PW	Position Wait; Unit is waiting for control to become idle so that it can initiate positioning.
P	Position; Unit is positioning; transfer of data follows although not necessarily on this controller.
TW	Transfer Wait; Unit is in position and is waiting for the controller/channel to become idle so that it can transfer data.
T	Transfer; Unit is transferring; the controller and channel are busy performing the operation.

Table 7-1 lists the possible states for files, units, controllers, and channels.

Table 7-1
Software States

File†	Unit	Controller	Channel
I	I	I	I
	SW		
	S		
PW	PW		
P	P		
TW	TW		
T	T	T	T

†Cannot be in S or SW state because SEEKs are ignored if the unit is not idle.

7.3.2 Queuing Strategy

When an I/O request for a unit is made by a user program because of an INPUT or OUTPUT UUO, one of several things can happen at UUO level before control is returned to the buffer-strategy module in UUOCON, which may, in turn, pass control back to the user without rescheduling. If an I/O request requires positioning of the unit, either the request is added to the end of the position-wait queue for

the unit if the control or unit is busy, or the positioning is initiated immediately. If the request does not require positioning, the data is transferred immediately. If the channel is busy, the request is added to the end of the transfer-wait queue for the channel. The control gives the processor an interrupt after each phase is completed. Optimization occurs at interrupt level when a position-done or transfer-done interrupt occurs.

7.3.2.1 Position-Done Interrupt Optimization - The following action occurs only if a transfer-done interrupt does not occur first. Data transfer is started on the unit unless the channel is busy transferring data for some other unit or control. If the channel is busy, the request goes to the end of the transfer-wait queue for that channel.

7.3.2.2 Transfer-Done Interrupt Optimization - When a transfer-done interrupt occurs, all the position-done interrupts inhibited during the data transfer are processed for the controller, and the requests are placed at the end of the transfer-wait queue for the channel. All units on the controller are then scanned. The requests in the position-wait queues on each unit are scanned to see the request nearest the current cyclinder. Positioning is begun on the unit of the selected request. All requests in the transfer-wait queue for all units on the channel that caused the interrupt are then scanned and the latency time is measured. The request with the shortest latency time is selected, and the new transfer begins.

7.3.3 Fairness Considerations

When the system selects the best task to run, users making requests to distant parts of the disk may not be serviced for a long time. The disk software is designed to make a fair decision for a fixed percentage of time. Every n decisions the disk software selects the request at the front of the position-wait or transfer-wait queue and processes it, because that request has been waiting the longest. The value of n is set to 10 (decimal) and may be changed by redefining symbols with MONGEN.

7.3.4 Channel Command Chaining

7.3.4.1 Buffered Mode - Disk accesses are reduced by using the chaining feature of the data channel. Prior to reading a block in buffered mode, the device independent routine checks to see if there is another empty buffer, and if the next relative block within the file is a consecutive logical block within the unit. If both checks are true, FILSER creates a command list to read two or more consecutive blocks into scattered core buffers. Corresponding decisions are made when writing data in buffered mode, and, if possible, two or more separate buffers are written in one operation. The command chaining decision is not made when a request is put into a position-wait or transfer-wait queue;

instead, it is postponed until the operation is performed, thus increasing the chances that the user program will have more buffers available for input or output. The default size of the channel command list is 20 decimal words, and can be changed by redefining CCWMAX with MONGEN.

7.3.4.2 Unbuffered Mode - Unbuffered modes do not use channel chaining, and therefore, read or write one command word at a time. Each command word begins at the beginning of a 128-word block. If a command word does not contain an even multiple of 128 words, the remaining words of the last block are not read, if reading, and are written with zeroes, if writing.

7.4 MONITOR ERROR HANDLING

The monitor detects a number of errors. If a hardware error is detected, the monitor repeats the operation ten times. If the failure occurs eleven times in a row, it is classified as a hard error. If the operation succeeds after failing one to ten times, it is a soft error.

7.4.1 Hardware Detected Errors

Hardware detected errors are classified either as device errors or as data errors. A device error indicates a malfunction of the controller or channel. A data error indicates that the hardware parity did not check or a search for a sector header either did not succeed or had bad parity (the user's data is probably bad).

A device error sets the IO.DER bit in the channel status word, and a data error sets the IO.DTE bit.

Disk units may have imperfect surfaces; therefore, a special non-timesharing diagnostic program, MAP, is provided to initially find all the bad blocks on a specified unit. The logical disk addresses of any bad regions of one or more bad blocks are recorded in the bad allocation table (BAT) block on the unit. The monitor allocates all storage for files; therefore, it uses the BAT block to avoid allocating blocks that have previously proven bad. The MAP program writes two copies of the BAT block because the BAT block might be destroyed. If the MAP program is not used, the monitor discovers the bad regions when it tries to use them and adds this information to the BAT block. However, the first user of the bad region loses that part of his data.

A hard data error usually indicates a bad surface; therefore, the monitor never returns the bad region to free storage. This results in the bad region causing an error only once. The bad unit and the logical disk address are stored in the retrieval information block (RIB) of the file when the file is CLOSEd or RESET and the extent of the bad region is determined. The origin and length of the bad region is stored in the bad allocation table (BAT) block.

7.4.2 Software Detected Errors

The monitor makes a number of software checks on itself. It checks the folded checksum (refer to Appendix H) computed for the first word of every group and stored in the retrieval pointer. The monitor also checks for inconsistencies when comparing locations in the retrieval information block with expected values (filename, filename extension, project-programmer number, special code, logical block number). The monitor checks for inconsistencies in the storage allocation table block when comparing the number of free clusters expected with the number of zeroes. A checksum error or an inconsistency error in the SAT block or RIB normally indicates that the monitor is reading the wrong block. When these errors occur, the monitor sets the improper mode error bit (IO.IMP) in the user channel status word and returns control to the user program.

7.5 DIRECTORIES

7.5.1 Order of Filenames

In 5.02 and earlier monitors, the names of newly created files are appended to the directory if the directory does not contain more than 64 filenames. If the directory contains more than 64 filenames, a second block is used for the new filenames. When filenames are deleted from the first block, entries from the second block are not moved into the first. When additional new files are created, their names are added to the end of the first block of the directory instead of the end of the directory. Thus, the order of the filenames in the directory may not be according to the date of creation.

In 5.03 and later monitors, if FTDUFD = 1, files are always entered in the directory in the order in which they are created. In the DECsystem-1040, FTDUFD is normally off indicating that the order of filenames is the same as in the 5.02 and earlier monitors.

7.5.2 Directory Searches

Table space in core memory is used to reduce directory searching times. The JBTPPB table contains pointers to a list of four-word blocks for the user's project-programmer number, one block for each file structure on which the user has a UFD.

Four-word name and access blocks contain copies of LOOKUP information for recently-accessed files and may reduce disk accesses to one directory read for a LOOKUP on a recently-active file. Recent LOOKUP failures are also kept in core, but are deleted when space is needed.

7.6 PRIORITY INTERRUPT ROUTINES

7.6.1 Channel Interrupt Routines

Each of the seven PI channels has two abolute locations associated with it in memory: 40+2n and 41+2n, where n is a channel number (1-7). When an interrupt occurs on a channel, control is immediately transferred to the first of the two associated locations (unless an interrupt on a higher priority

channel is being processed). For fast service of a single device, the first location contains either a BLKI or BLKO instruction. For service of more than one device on the same channel, the first location contains a JSR to location CHn in the appropriate channel interrupt routine. The JSR ensures that the current state of the program counter is saved.

Each channel interrupt routine (mnemonic name, CHANn, where n is the channel number) consists of three separate routines:

CHn:	The contents of the program counter is saved in location CHn. CHn+1 contains a JRST to the first device service routine in the interrupt chain.
SAVCHn:	The routine to save the contents of a specified number of accumulators. It is called from the device service routines with a JSR.
XITCHn:	The routine to restore saved accumulators. Device service routines exit to XITCHn with a POPJ PDP, if SAVCHn was previously called.

7.6.2 Interrupt Chains

Each device routine contains a device interrupt routine DEVINT where DEV is the three-letter mnemonic for the device concerned. This routine checks to determine whether an interrupt was caused by device DEV. The interrupt chain of a given channel is a designation for the logical positioning of each device interrupt routine associated with that channel.

The monitor flow of control on the interrupt level through a chain is illustrated below. Channel 5 is used in the example.

```
Monitor Routine   Relevant Code                          Explanation

Absolute          52/JSR CH5                             ;control transferred here
Locations         53/                                    ;on interrupt
                     ↓
CHAN5             CH5: 0                                 ;contents of PC saved here
                     JRST PTPINT                         ;control transfers to first
                     ↓                                   ;link in interrupt chain

                  PTPINT: CONSO PTP,PTPDON               ;if PDP done bit is
PTPSER               JRST LPTINT                         ;on, PTP was cause
                     .                                   ;of interrupt -
                     :                                   ;otherwise, go to
                     ↓                                   ;next device.

                  LPTINT: CONSO LPT,LPTLOV+LPTERR+LPTDON
LPTSER               JEN @ CH5                           ;three possible bits
                     .                                   ;may indicate that
                     :                                   ;LPT caused interrupt
```

When a real-time device is added to the interrupt chain (CONSO skip chain) by a RTTRP UUO (refer to Paragraph 3.8.1), the device is added to the front of the chain. After putting a real-time device on Channel 5 in single mode (refer to Paragraph 3.8.1), the chain is as follows:

```
Monitor Routine     Relevant Code                        Explanation

Absolute            52/JSR CH5                           ;control transferred here
Locations           53/                                  ;on interrupt
                        ↓
CHAN5               CH5: 0                               ;contents of PC saved here
                        JRST RDTINT                      ;control transfers to first
                        ↓                                ;link in interrupt chain
RTDEV               RTDINT: CONSO RTD,BITS
                        JRST PTPINT
                        JRST <context switcher and
                              dispatch for real-time
                              interrupts >
                    PTPINT: CONSO PTP,PTPDON             ;if PTP done bit is
PTPSER                  JRST LPTINT                      ;on, PTP was cause
                        .                                ;of interrupt -
                        .                                ;otherwise, go to
                        ↓                                ;next device.
                    LPTINT:CONSO LPT, LPTLOV+LPTERR+LPTDON
LPTSER                  JEN @ CH5                        ;three possible bits
                        .                                ;may indicate that
                        .                                ;LPT caused interrupt
```

After putting a real-time device on channel 5 in normal block mode (refer to Paragraph 3.8.1), the chain is as follows:

```
Monitor Routine     Relevant Code                        Explanation

Absolute            52/JSR CH5                           ;control transferred here
Locations           53/                                  ;on interrupt
                        ↓
CHAN5               CH5: 0                               ;contents of PC saved here
                        JRST RTDINT                      ;control transfers to first
                        ↓                                ;link in interrupt chain
RTDEV               RTDINT:CONSO RTD,BITS
                        JRST PTPINT
                        BLKI RTD,POINTR
                        JRST <context switcher >
                        JEN @ CH5
```

(continued on next page)

Monitor Routine	Relevant Code	Explanation
	PTPINT: CONSO PTP,PTPDON	;if PTP done bit is
PTPSER	JRST LPTINT	;on, PTP was cause
	⋮	;of interrupt -
		;otherwise, go to
	↓	;next device.
	LPTINT:CONSO LPT, LPTLOV+LPTERR+LPTDON	
LPTSER	JEN @ CH5	;three possible bits
	⋮	;may indicate that
		;LPT caused interrupt.

After putting a real-time device on channel 6 in fast block mode (refer to Paragraph 3.8.1), the chain is as follows:

Monitor Routine	Relevant Code	Explanation
Absolute	54/BLKO RTD,POINTR	;control transferred
Locations	55/JSR CH6	;here on interrupt
CHAN6	CH6: 0	;contents of PC saved
	JRST <context switcher >	;control transfers to
		;context switcher.

The exec mode trapping feature can be used with any of the standard forms of the RTTRP UUO: single mode, normal block mode, and fast block mode. The following examples illustrate the chain when used with each of the three modes.

Single Mode (Exec Mode)

Monitor Routine	Relevant Code	Explanation
Absolute	52/JSR CH5	;control transferred here
Locations	53/	;on interrupt
	↓	
CHAN5	CH5: 0	;contents of PC saved here
	JRST RDTINT	;control transfers to first
	↓	;link in interrupt chain
RTDEV	RTDINT: CONSO RTD,BITS	
	JRST PTPINT	
	JSR TRPADR	
	JEN @ CH5	

(continued on next page)

Single Mode (Exec Mode) (Cont)

Monitor Routine	Relevant Code	Explanation
	PTPINT:CONSO PTP,PTPDON	;if PTP done bit is
PTPSER	JRST LPTINT	;on, PTP was cause
	⋮	;of interrupt -
		;otherwise, go to
	↓	;next device.
	LPTINT:CONSO LPT, LPTLOV+LPTERR+LPTDON	
LPTSER	JEN @ CH5	;three possible bits
	⋮	;may indicate that
		;LPT caused interrupt

Normal Block Mode (Exec Mode)

Monitor Routine	Relevant Code	Explanation
Absolute	52/JSR CH5	;control transferred here
Locations	53/	;on interrupt
	↓	
CHAN5	CH5: 0	;contents of PC saved here
	JRST RTDINT	;control transfers to first
	↓	;link in interrupt chain
RTDEV	RTDINT:CONSO RTD,BITS	
	JRST PTPINT	
	BLKI RTD,POINTR	
	JSR TRPADR	
	JEN @ CH5	
	PDPINT: CONSO PTP,PTPDON	;if PTP done bit is
PTPSER	JRST LPTINT	;on, PDP was cause
	⋮	;of interrupt -
		;otherwise, go to
	↓	;next device.
	LPTINT:CONSO LPT,LPTLOV+LPTERR+LPTDON	
LPTSER	JEN @ CH5	;three possible bits
	⋮	;may indicate that
		;LPT caused interrupt.

Fast Block Mode (Exec Mode)

Monitor Routine	Relevant Code	Explanation
Absolute	54/BLKO RTD,POINTR	;control transferred here
Locations	55/JSR CH6	;on interrupt
	↓	

(continued on next page)

Fast Block Mode (Exec Mode) (Cont)

Monitor Routine	Relevent Code	Explanation
CHAN6	CH6: 0 JRST RDTINT ↓	;contents of PC saved here ;control transfers to first ;link in interrupt chain
RTDEV	RTDINT: JSR TRPADR JEN @ CH6	

7.7 MEMORY PARITY ERROR ANALYSIS, REPORTING, AND RECOVERY[1]

The memory parity error analysis and recovery software allows the machine to run with PARITY STOP off, thereby gaining increased CPU speed (10% more on the KA10 processor and 100% more on the KI10 processor), better error reporting, and improved failsoft recovery. The analysis software considers its goals to be

1) Never jeopardize the system or the user program by allowing it to continue with bad data from memory.
2) Always maintain the running of the system with the maximum number of users as possible as long as there is no possibility of violating the integrity of the system or the user program.

In either case, complete information is printed for the operator so that he can reconfigure the memories and reload the system when necessary. Additional information is recorded on the disk by DAEMON for field service in order that the cause of the error can be located and fixed.

7.7.1 Description of Analysis

The error analysis software differentiates between user mode and executive mode when a parity error occurs. If the processor is executing in user mode and the user is enabled for parity trapping (refer to Paragraph 3.1.3.1), control is transferred to the user's routine. Otherwise, the execution of the user's job is stopped and the user receives the error messages

```
?ERROR IN JOB n
?MEM PAR ERR AT USER PC nnnnnn
```

Simultaneously, a request is made for the lowest priority channel routine to sweep through core in order to locate all words with bad memory parity, in case there is more than one word. During the sweep, all locations with bad parity are rewritten, so that subsequent references usually will not receive a parity error. After the sweep of core is completed, all jobs (including the current job) with

[1]This feature depends on FTMEMPAR which is normally off in the DECsystem-1040.

parity errors in their low segments receive the above ERROR IN JOB message. All jobs with errors in their high segments are swapped out if the high segment has the hardware user-mode write protect bit set, since a copy exists on the swapping space. In this case recovery occurs for all jobs sharing the high segment except for the currently running job. If the high segment is not write protected for a job (so that there is no copy on the disk), if the high segment is locked, or if one of the sharing job's low segment is locked, all jobs sharing the high segment are stopped and receive an error message since no recovery is possible. In addition, the segment name is cleared so that new users will receive a new copy from the file system on a R, RUN, or GET command or a RUN or GETSEG UUO.

If the processor is in executive mode when the error occurs, the analysis procedure depends on the value of the PC. Two conditions are recognized as not being harmful:

1) a parity error during the PI 7 sweep of memory.
2) a parity error during the storing of data words around the location of a channel-detected memory parity error.

If the PC is at the BLT instruction which moves user core to facilitate core allocation, the bad word is determined from the BLT pointer. If the pointer is in the protected part of the job data area, this area is cleared so the monitor will not attempt to use the bad words, since they contain executive mode addresses. In either case, the user's job is stopped and an error message is output to the user. In addition, the memory sweep procedure is invoked to find additional words with bad parity.

If the PC is an executive mode location and there are no PIs in progress, the UUO is run to completion, the current user receives an error message, and the memory sweep procedure is invoked. If the sweep routine detects bad parity in an address within the monitor or detects no words with bad parity (because they have been rewritten on a read-pause-write instruction), the routine prints on the CTY (instead of OPR),

```
?EXEC PARITY HALT
?n MEM PAR ERRS FROM aaaaaa TO bbbbbb ON CPUn FOR JOBx [program]
```

and then HALTs. This message is printed without using the interrupt system in order to maximize the chances of the message being output. Although the operator can attempt to continue the system by pushing the CONT console switch, this is not a recommended operator procedure (e.g., the monitor may have incorrect data thereby causing more damage). (Refer to MEMPAR in Notebook 8 of the DECsystem-10 Software Notebooks for complete operator instructions on memory parity error recovery.)

If a PI is in progress when the parity error is detected, a sweep of core is made at the high priority APR PI level. If a word with bad parity is discovered in the monitor area or no parity errors are found, the monitor prints the above message to the operator and halts. The finding of words without bad parity is considered serious because the read-pause-write class instructions rewrite memory before the parity interrupt occurs so that the parity error is usually corrected. In this case, the operator receives the message

```
?0 MEM PAR ERRORS
```

On all recoverable or non-recoverable parity errors, the operator receives on either OPR or CTY a message similar to the following:

?n MEM PAR ERRS FROM aaaaaa TO bbbbbb ON CPUn for JOBx [program]

preceded by 5 bells. This alerts him to potential problems and gives him the necessary information for reconfiguring the memories. In addition, the operator is notified of the jobs that have been stopped in case they are crucial to the operation of the system. If they are, he can take appropriate action to restart them.

If the DF10 channel detects a memory parity error while reading for file I/O from memory, the user's job is not stopped and the user does not receive an error message. Instead the error is treated as a device error and the IO.DER error bit is set. However, the operator receives the message

?n MEM PAR ERRS FROM aaaaaa TO bbbbbb ON CHANNEL n

where n is the logical channel number starting with the fastest device as defined by MONGEN. For example, the fastest disk unit is on the first channel and the magnetic tape TM10B control is on the last channel.

If the DF10 channel detects a memory parity error while swapping a job out of core, the user's job is stopped and the user receives the following error message:

?ERROR IN JOB n
?SWAP OUT CHN MEM PAR ERR

The operator receives the message

?m MEM PAR ERRS FROM aaaaaa TO bbbbbb ON CHANNEL n FOR JOB x [prog]

If the error is detected in a high segment on the swap out, all jobs using the high segment receive the error message. The high segment name is cleared so that new users will receive a new copy of the segment from the file system.

On all parity errors detected by the processors or the channels, DAEMON is awakened to correct the information stored by the monitor's analysis routine. DAEMON writes this information in the hardware log file on the disk for the use of field service in diagnosing and solving the problem.

APPENDIX A
DECTAPE COMPATIBILITY BETWEEN DEC COMPUTERS

The following chart illustrates the ability to read the indicated tapes with a suitable program. In general, the standard software of machines of one family will not read tapes written by the standard software of machines of a different family.

The standard tapes of the PDP-1, PDP-4, PDP-6, PDP-7, PDP-9, PDP-10, PDP-11, and PDP-15 consist of 578 blocks of 128 36-bit words (256 18-bit words). The standard tapes of the PDP-15 and the PDP-8 family consist of 4096 blocks of 129 12-bit words (43 36-bit words).

Written by \ Read by	PDP-1 550,550-A And 555, TU55, TU56	PDP-4 550 And 555, TU55, TU56	PDP-5 552 And 555, TU55, TU56	PDP-6 551 And 555, TU55 TU56	PDP-7 550-A And 555, TU55, TU56	PDP-8 552, TC01 And 555, TU55, TU56	PDP-8/I,L TC01, TC08 And TU55, TU56	PDP-8/E TC08-P, TD8E And TU55, TU56	LINC-8 And Optional Converter And LINCtape Drive	PDP-9 TC02 And TU55, TU56	PDP-10 TD10 And TU55, TU56	PDP-11 TC11 And TU56	PDP-12 TC12 And TC12-F TU55, TU56	PDP-15 TC02, TC15 And TU55, TU56
PDP-1	A	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
PDP-4	Z	A	D	D	A	D	D	D	E	D	D	D	E	D
PDP-5	Z	D	A	B	C	A	A	A	E	A	A	A	E	A
PDP-6	Z	D	A	A	C	A	A	A	E	A	A	A	E	A
PDP-7	Z	A	C	C	A	C	C	C	E	C	C	C	E	C
PDP-8	Z	D	A	B	C	A	A	A	E	A	A	A	E	A
PDP-8/I,L	Z	D	A	B	C	A	A	A	E	A	A	A	E	A
PDP-8/E	Z	D	A	B	C	A	A	A	E	A	A	A	E	A
LINC-8 & Converter	Z	E	E	B	E	E	E	E	A	E	E	E	A	E
PDP-9	Z	D	A	A	C	A	A	A	E	A	A	A	E	A
PDP-10	Z	D	A	A	C	A	A	A	E	A	A	A	E	A
PDP-11	Z	D	A	B	C	A	A	A	E	A	A	A	E	A
PDP-12 & TC12-F	Z	E	E	B	E	E	E	E	A	E	E	E	A	E
PDP-15	Z	D	A	A	C	A	A	A	E	A	A	A	E	A

KEY:

A = Can be done.

B = Can be done only by ignoring indicated checksum errors.

C = Can be done with programmed checksum.

D = Can probably be done as in (C) except that PDP-4 is too slow for calculating the exclusive-or checksum in line; calculations must be done before writing and after reading.

E = Program and optional hardware exist to convert to and from LINCtape format. Standard PDP-12 and LINC-8 tapes are in LINCtape format which is incompatible. DECtapes must be formatted on another machine before writing on PDP-12 or LINC-8.

Z = No information available.

NOTES:

1. PDP-8/S cannot use DECtape. Classic LINC only uses LINCtape which is incompatible with DECtape.
2. The PDP-6 and 10 (and probably other machines) cannot find the first or last block when searching from the end zone.
3. The PDP-9 and -15 software writes data in reverse order in blocks which are written while moving in the reverse direction.

APPENDIX B
WRITING REENTRANT USER PROGRAMS

B.1 DEFINING VARIABLES AND ARRAYS

The LOADER simplification makes it somewhat more difficult to define variables and arrays. The easiest way to define variables and arrays, so the resulting relocatable binary can be loaded on a one- or two-segment machine, is to put them all in a separate subprogram as internal global symbols using Block 1 and Block N pseudo-ops. All other subprograms refer to this data as external global locations. Most reentrant programs have at least two subprograms, one for the definition of low segment locations and one for instructions and constants for the high segment. (This last subprogram must have either a HISEG pseudo-op or a TWOSEG pseudo-op followed by RELOC 400000.) Programs are self-initializing; therefore, they clear the low segment when they are started although the monitor clears core when it assigns it to a user.

Block 1 and Block N pseudo-ops cause the LOADER to leave indications in the job data area (LH of JOBCOR) so a monitor SAVE command will not write the low segment. This is advantageous in sharable programs for two reasons. It reduces the number of files in small DECtape directories (the maximum is 22 files). Also, I/O is accomplished only on the first user's GET that initializes the high segment, but not on any subsequent user's GETs for either the high or low segment.

B.2 EXAMPLE OF TWO-SEGMENT REENTRANT PROGRAM

```
LOW SEGMENT SUBPROGRAM:

TITLE LOW - EXAMPLE OF LOW SEGMENT SUB-PROGRAM
        JOBVER=137
        LOC     JOBVER
        3                       ;VERSION3
        RELOC   0
        INTERNAL LOWBEG,DATA,DATA1,DATA2,TABLE,TABLE1

LOWBEG:
DATA:     BLOCK  1
DATA1:    BLOCK  1
DATA2:    BLOCK  1

TABLE:    BLOCK  10
TABLE1:   BLOCK  10
LOWEND=.-1                      ;LAST LOCATION TO BE CLEARED
          END
```

```
HIGH SEGMENT SUBPROGRAM:

TITLE HIGH - EXAMPLE OF HIGH SEGMENT SUB-PROGRAM
        HISEG                       ;OR TWOSEG
                                    ;RELOC 400000
        EXTERN  LOWBEG,LOWEND
        T=1
BEGIN:  SETZM   LOWBEG              ;CLEAR DATA AREA
        MOVEI   T,LOWBEG+1
        HRLI    T,LOWBEG
        BLT     T,LOWEND
        MOVE    T,DATA1             ;COMPUTE
        ADDI    1,1
        MOVEM T, DATA2
          .
          .
          .
        END     BEGIN               ;STARTING ADDRESS
```

B.3 CONSTANT DATA

Some reentrant programs require certain locations in the low segment to contain constant data, which does not change during execution. The initialization of this data happens only once after each GET, instead of after each START; therefore, programmers are tempted to place these constants in the sub-program that contains the definition of the variable data locations. This action requires the SAVE command to write the constants out and the GET command to load the constants again; therefore, the constant data should be moved by the programs from the high segment to the low segment when the rest of the low segment is being initialized. The exception is when the amount of code and constants in the high segment needed to initialize the low segment constants take up too much room in the high segment. In this case, it is best to have I/O in the low segment on each GET. A rule to follow in deciding between this high segment core space and the low segment GET I/O time is: put the code in the high segment if it does not put the high segment over the next 1K boundary.

B.4 SINGLE SOURCE FILE

A second way of writing single save file reentrant programs is to have a single source file instead of two separate ones. This is more convenient, although it involves conditional assembly and, therefore, produces two different relocatable binaries. A number of system programs have been written this way. The idea is to have a conditional switch which is 1 if a reentrant assembly and 0 if a non-reentrant assembly.

```
DEMO - DEMO ONE SOURCE REENTRANT PROGRAM -V002  MACRO 44.0.4 14:30  2-APR-71 PAGE 1
DEMO

     1                              TITLE DEMO - DEMO ONE SOURCE REENTRANT PROGRAM -V002
     2
     3  000137                                LOC <JOBVER=137>
     4  000137  000000  000002                EXP     002              ;VERSION NUMBER
     5  000000'                               RELOC
     6
     7                                        INTERN  JOBVER,PURE
     8
     9                              IFNDEF PURE,<PURE==1>              ;ASSUME REENTRANT IF PURE UNDEFINED
    10  400000'                               IFN PURE,<TWOSEG>        ;TELL LOADER TO EXPECT TWO SEGMENTS
    11  400000'                               IFN PURE,<RELOC 400000>  ;START OF HIGH SEGMENT RELOCATION
    12
    13  400000' 047000  000000  BEG:          RESET                    ;RESET ALL I/O
    14  400001' 200000  400007'               MOVE    0,[DATAB,,DATAB+1]
    15  400002' 402000  000000'               SETZM   DATAB            ;NOW CLEAR DATA REGION
    16  400003' 251000  000203'               BLT     0,DATAE-1        ;UP TO LAST LOCATION
    17  400004' 000000  400004'               .
    18  400005' 000000  400005'               .
    19  400006' 000000  400006'               .
    20
    21  000000'                     IFN PURE,<RELOC>                   ;SET RELOCATION COUNTER TO LOW SEGMENT
    22
    23  000000'                     DATAB:                             ;FIRST LOCATION CLEARED ON STARTUP
    24  000000'                     DATA:     BLOCK   1
    25  000001'                     TABLE:    BLOCK   ↑D128
    26  000201' 000000  000201'               .
    27  000202' 000000  000202'               .
    28  000203' 000000  000203'               .
    29  000204'                     DATAE:                             ;END OF DATA AREA
    30
    31  400007'                     IFN PURE,<RELOC>                   ;BACK TO HIGH SEGMENT
    32  400007'                               LIT                      ;PUT LITERALS IN HIGH SEGMENT
    33  400007' 000000' 000001'
    34                  400000'               END     BEG

NO ERRORS DETECTED

HI-SEG. BREAK IS 400010
PROGRAM BREAK IS 000204

2K CORE USED
```

APPENDIX C
CARD CODES

Table C-1
ASCII Card Codes

ASCII Character	Octal Code	Card Punches	ASCII Character	Octal Code	Card Punches
NULL	00	12-0-9-8-1	@	100	8-4
CTRL-A	01	12-9-1	A	101	12-1
CTRL-B	02	12-9-2	B	102	12-2
CTRL-C	03	12-9-3	C	103	12-3
CTRL-D	04	9-7	D	104	12-4
CTRL-E	05	0-9-8-5	E	105	12-5
CTRL-F	06	0-9-8-6	F	106	12-6
CTRL-G	07	0-9-8-7	G	107	12-7
CTRL-H	10	11-9-6	H	110	12-8
TAB	11	12-9-5	I	111	12-9
LF	12	0-9-5	J	112	11-1
VT	13	12-9-8-3	K	113	11-2
FF	14	12-9-8-4	L	114	11-3
CR	15	12-9-8-5	M	115	11-4
CTRL-N	16	12-9-8-6	N	116	11-5
CTRL-O	17	12-9-8-7	O	117	11-6
CTRL-P	20	12-11-9-8-1	P	120	11-7
CTRL-Q	21	11-9-1	Q	121	11-8
CTRL-R	22	11-9-2	R	122	11-9
CTRL-S	23	11-9-3	S	123	0-2
CTRL-T	24	9-8-4	T	124	0-3
CTRL-U	25	9-8-5	U	125	0-4
CTRL-V	26	9-2	V	126	0-5
CTRL-W	27	0-9-6	W	127	0-6
CTRL-X	30	11-9-8	X	130	0-7
CTRL-Y	31	11-9-8-1	Y	131	0-8
CTRL-Z	32	9-8-7	Z	132	0-9
ESCAPE	33	0-9-7	[	133	12-8-2
CTRL-\	34	11-9-8-4	\	134	0-8-2
CTRL-]	35	11-9-8-5	]	135	11-8-2
CTRL-↑	36	11-9-8-6	↑ ∧	136	11-8-7
CTRL-←	37	11-9-8-7	← _	137	0-8-5
SPACE	40		\ ‾	140	8-1

NOTE: The ASCII character ESCAPE (octal 33) is also CTRL-[on a terminal.

Table C-1 (Cont)
ASCII Card Codes

ASCII Character	Octal Code	Card Punches	ASCII Character	Octal Code	Card Punches
!	41	12-8-7	a	141	12-0-1
"	42	8-7	b	142	12-0-2
#	43	8-3	c	143	12-0-3
$	44	11-8-3	d	144	12-0-4
%	45	0-8-4	e	145	12-0-5
&	46	12	f	146	12-0-6
'	47	8-5	g	147	12-0-7
(	50	12-8-5	h	150	12-0-8
)	51	11-8-5	i	151	12-0-9
*	52	11-8-4	j	152	12-11-1
+	53	12-8-6	k	153	12-11-2
,	54	0-8-3	l	154	12-11-3
-	55	11	m	155	12-11-4
.	56	12-8-3	n	156	12-11-5
/	57	0-1	o	157	12-11-6
0	60	0	p	160	12-11-7
1	61	1	q	161	12-11-8
2	62	2	r	162	12-11-9
3	63	3	s	163	11-0-2
4	64	4	t	164	11-0-3
5	65	5	u	165	11-0-4
6	66	6	v	166	11-0-5
7	67	7	w	167	11-0-6
8	70	8	x	170	11-0-7
9	71	9	y	171	11-0-8
:	72	8-2	z	172	11-0-9
;	73	11-8-6	{	173	12-0
<	74	12-8-4	|	174	12-11
=	75	8-6	}	175	11-0
>	76	0-8-6	~	176	11-0-1
?	77	0-8-7	DEL	177	12-9-7

NOTE: The ASCII characters } and ~(octal 175 and 176) are treated by the monitor as ALT-MODE and are often considered the same as ESCAPE.

Table C-2
DEC-029 Card Codes

Character	Octal Code	Card Punches	Character	Octal Code	Card Punches
SPACE	40		@	100	8-4
!	41	11-8-2	A	101	12-1
"	42	8-7	B	102	12-2
#	43	8-3	C	103	12-3
$	44	11-8-3	D	104	12-4
%	45	0-8-4	E	105	12-5
&	46	12	F	106	12-6
'	47	8-5	G	107	12-7
(	50	12-8-5	H	110	12-8
)	51	11-8-5	I	111	12-9
*	52	11-8-4	J	112	11-1
+	53	12-8-6	K	113	11-2
,	54	0-8-3	L	114	11-3
-	55	11	M	115	11-4
.	56	12-8-3	N	116	11-5
/	57	0-1	O	117	11-6
0	60	0	P	120	11-7
1	61	1	Q	121	11-8
2	62	2	R	122	11-9
3	63	3	S	123	0-2
4	64	4	T	124	0-3
5	65	5	U	125	0-4
6	66	6	V	126	0-5
7	67	7	W	127	0-6
8	70	8	X	130	0-7
9	71	9	Y	131	0-8
:	72	8-2	Z	132	0-9
;	73	11-8-6	[	133	12-8-2
<	74	12-8-4	\	134	11-8-7
=	75	8-6	]	135	0-8-2
>	76	0-8-6	↑ ∧	136	12-8-7
?	77	0-8-7	← _	137	0-8-5

NOTE: Octal codes 0-37 and 140-177 are the same punches as ASCII.

Table C-3
DEC-026 Card Codes

Character	Octal Code	Card Punches	Character	Octal Code	Card Punches
SPACE	40		@	100	8-4
!	41	12-8-7	A	101	12-1
"	42	0-8-5	B	102	12-2
#	43	0-8-6	C	103	12-3
$	44	11-8-3	D	104	12-4
%	45	0-8-7	E	105	12-5
&	46	11-8-7	F	106	12-6
'	47	8-6	G	107	12-7
(	50	0-8-4	H	110	12-8
)	51	12-8-4	I	111	12-9
*	52	11-8-4	J	112	11-1
+	53	12	K	113	11-2
,	54	0-8-3	L	114	11-3
-	55	11	M	115	11-4
.	56	12-8-3	N	116	11-5
/	57	0-1	O	117	11-6
0	60	0	P	120	11-7
1	61	1	Q	121	11-8
2	62	2	R	122	11-9
3	63	3	S	123	0-2
4	64	4	T	124	0-3
5	65	5	U	125	0-4
6	66	6	V	126	0-5
7	67	7	W	127	0-6
8	70	8	X	130	0-7
9	71	9	Y	131	0-8
:	72	11-8-2/11-0	Z	132	0-9
;	73	0-8-2	[	133	11-8-5
<	74	12-8-6	\	134	8-7
=	75	8-3	]	135	12-8-5
>	76	11-8-6	↑ ∧	136	8-5
?	77	12-8-2/12-0	← _	137	8-2

NOTE: Octal codes 0-37 and 140-177 are the same punches as ASCII.

APPENDIX D
DEVICE STATUS BITS

Table D-1
Device Status Bits

Device	Function	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
CDP	SETSTS												DEC 029 Card Codes		User Word Count	Data Mode			
	GETSTS		Punch Error		Data when EOC reached		I/O Active												
CDR	SETSTS												Super Image Mode	Sync Input		Data Mode			
	GETSTS	No 7-9 Punch	Data Missed	Binary Check-sum Error		EOF card EOF button	I/O Active												
DIS	SETSTS															Data Mode			
	GETSTS						I/O Active												
DSK	SETSTS												Write Head-ers	Sync Input	User Word Count	Data Mode			
	GETSTS	Write Lock	Search Error	Disk Parity Error	Block No. Too Large	End of File	I/O Active												

Table D-1 (cont)
Device Status Bits

Device	Function	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
DTA	SETSTS											Semi-Standard I/O Mode	Non-structured Dump Mode	Sync Input	User Word Count	Data Mode			
	GETSTS	Write Lock	Data Missed	Parity Error	Block No. Too Large	End of File	I/O Active												
LPT	SETSTS												Suppress Form Feeds		User Word Count	Data Mode			
	GETSTS						I/O Active												
MTA	SETSTS									Write Even Parity	Tape Density		No Retry	Sync Input	User Word Count	Data Mode			
	GETSTS	Write Lock Illegal Operation	Data Missed	Parity Error	Record Too Long	End of File	I/O Active	Load Point Rewinding	End Point										
PLT	SETSTS														User Word Count	Data Mode			
	GETSTS						I/O Active												
PTP	SETSTS														User Word Count	Data Mode			
	GETSTS						I/O Active												
PTR	SETSTS													Sync Input		Data Mode			
	GETSTS	Block Incomplete		Checksum Error		End of Tape	I/O Active												

Table D-1 (Cont)
Device Status Bits

Device Function		18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
PTY	SETSTS GETSTS				Block No. Too Large		I/O Active	PTY Wait	TTY Re-sponse	Monitor Mode						Data Mode			
TTY	SETSTS										Echo of $ Sup-press	Echo Sup-press	Full Char-acter Set	Sync Input	User Word Count	Data Mode			
	GETSTS	TTY Not Assig-ned for image mode input	Ignore Inter-rupt	Echo Fail-ure	Char-acter Lost		I/O Active												

Note 1: SETSTS UUO may set all bits except Bit 23 and GETSTS UUO may return all bits (18-35); however, the two are separated to show those bits normally set by the user program on INIT, OPEN, or SETSTS as distinct from those normally set by the monitor (GETSTS).

Note 2: Unused bits should always have the value 0.

Note 3: Refer to the appropriate device sections in Chapters 5 and 6 for the complete description of each status bit.

APPENDIX E
ERROR CODES

The error codes in Table E-1 are returned in AC on RUN and GETSEG UUOs, in location E + 1 on 4-word argument blocks of LOOKUP, ENTER, and RENAME UUOs, and in the right half of location E + 3 on extended LOOKUP, ENTER, and RENAME UUOs. The codes are defined in the S.MAC monitor file.

Table E-1
Error Codes

Symbol	Code	Explanation
ERFNF%	0	File not found, illegal filename (0,*), filenames do not match (UPDATE), or RENAME after a LOOKUP failed.
ERIPP%	1	UFD does not exist on specified file structures. (Incorrect project-programmer number.)
ERPRT%	2	Protection failure or directory full on DTA.
ERFBM%	3	File being modified (ENTER, RENAME).
ERAEF%	4	Already existing filename (RENAME) or different filename (ENTER after LOOKUP).
ERISU%	5	Illegal sequence of UUOs (RENAME with neither LOOKUP nor ENTER, or LOOKUP after ENTER).
ERTRN%	6	a. Transmission, device, or data error (RUN, GETSEG only).
		b. Hardware-detected device or data error detected while reading the UFD RIB or UFD data block.
		c. Software-detected data inconsistency error detected while reading the UFD RIB or file RIB.
ERNSF%	7	Not a saved file (RUN, GETSEG only).
ERNEC%	10	Not enough core (RUN, GETSEG only).
ERDNA%	11	Device not available (RUN, GETSEG only).
ERNSD%	12	No such device (RUN, GETSEG only).
ERILU%	13	Illegal UUO (GETSEG only). No two-register relocation capability.

(continued on next page)

Table E-1 (Cont)
Error Codes

Symbol	Code	Explanation
ERNRM%	14	No room on this file structure or quota exceeded (over-drawn quota not considered).
ERWLK%	15	Write-lock error. Cannot write on file structure.
ERNET%	16	Not enough table space in free core of monitor.
ERPOA%	17	Partial allocation only.
ERBNF%	20	Block not free on allocated position.
ERCSD%	21	Cannot supersede an existing directory (ENTER).
ERDNE%	22	Cannot delete a non-empty directory (RENAME).
ERSNF%	23	Sub-directory not found (some SFD in the specified path was not found).
ERSLE%	24	Search list empty (LOOKUP or ENTER was performed on generic device DSK and the search list is empty).
ERLVL%	25	Cannot create a SFD nested deeper than the maximum allowed level of nesting.
ERNCE%	26	No file structure in the job's search list has both the no-create bit and the write-lock bit equal to zero and has the UFD or SFD specified by the default or explicit path (ENTER on generic device DSK only).
ERSNS%	27	GETSEG from a locked low segment to a high segment which is not a dormant, active, or idle segment. (Segment not on the swapping space.)

APPENDIX F
COMPARISON OF DISK-LIKE DEVICES

Table F-1
Disk Devices

Device Name	Fixed-Head Disk	Drum	Removable Disk Pack(s)	
Manufacturer	Burroughs	Bryant	Memorex, ISS	
Device Type Controller	RD10 RC10	RM10B RC10	RP02 RP10	RP03 RP10
Maximum Disks per Controller	4	4	8	8
Maximum Controllers per System	2	2	3	3
Hardware Mnemonic	DSK	DSK	DPC	DPC
Software Mnemonic	FHA, FHB	FHA, FHB	DPA, DPB, DPC	DPA, DPB, DPC
Capacity Minimum (X10**6 words)	.5	.345	5.2	10.4
Maximum (1 control) (X10**6 words)	2	1.38	41.4	82.8
Blocks/Track	20	30	10	10
Blocks/Cylinder	4000	2700	200	400
Blocks/Unit	4000	2700	40000	80000
Rotational Speed (rpm)	1800	3600	2400	2400
Revolution Time (msec)	33	17	25	25
128-Word Blocks/ Revolution	20	30	10	10
Transfer Rate μs word	13	4.3	15	15

Table F-1 (Cont)
Disk Devices

Device Name	Fixed-Head Disk	Drum	Removable Disk Pack(s)	
Manufacturer	Burroughs	Bryant	Memorex, ISS	
Device Type Controller	RD10 RC10	RM10B RC10	RP02 RP10	RP03 RP10
Seek Time				
Average (msec)	0	0	50	50
Minimum (msec)	0	0	20	20
Maximum (msec)	0	0	80	80
Swapping Times (msec)			(includes 30 ms verify)	
1K	25	13	84	84
4K	73	27	144	144
10K	154	54	256	264
25K	358	120	589	589

NOTE

Although the Bryant drum is a drum in every sense, its software mnemonic is still FHA because it is connected to the system through the fixed head disk control.

APPENDIX G
MAGNETIC TAPE CODES

Table G-1
ASCII Codes and BCD Equivalents

ASCII	Character Symbol	BCD	ASCII	Character Symbol	BCD
040	blank	20	100	@	57
041	!	52	101	A	61
042	"	17	102	B	62
043	#	32	103	C	63
044	$	53	104	D	64
045	%	77	105	E	65
046	&	35	106	F	66
047	'	14	107	G	67
050	(	34	110	H	70
051	)	74	111	I	71
052	*	54	112	J	41
053	+	60	113	K	42
054	,	33	114	L	43
055	-	40	115	M	44
056	.	73	116	N	45
057	/	21	117	O	46
060	Ø	12	120	P	47
061	1	01	121	Q	50
062	2	02	122	R	51
063	3	03	123	S	22
064	4	04	124	T	23
065	5	05	125	U	24
066	6	06	126	V	25
067	7	07	127	W	26
070	8	10	130	X	27
071	9	11	131	Y	30
072	:	15	132	Z	31
073	;	56	133	[	75
074	<	76	134	\	36†
075	5	13	135	]	55
076	>	16	136	↑	illegal
077	?	72	137	←	37

† Code used for all illegal codes.

When converting from ASCII to BCD, the following is done for ASCII codes 000-037 and 140-177:

000	ignored.
001-010	same as ASCII 134.
011	same as ASCII 040.
012-014	constitutes end of line.
015	ignored.
016-031	same as ASCII 134.
032	end of file.
033-037	same as ASCII 134.
140	same as ASCII 134.
141-172	same as ASCII 101-132.
173-176	same as ASCII 134.
177	ignored.

APPENDIX H
FILE RETRIEVAL POINTERS

Sequential and random file access are handled more efficiently by the monitor if all the information describing the file can be kept in core at once. To understand this effect, it is necessary to know how the monitor accesses files.

With each named file, UFD, and MFD, the monitor writes a special block containing necessary information needed to retrieve the data blocks that constitute the file. This block is called a retrieval information block, or RIB.

Retrieval pointers in the RIB describe contiguous blocks of file storage space called groups. Each pointer occupies one word and has one of three forms:

a. A group pointer
b. An EOF pointer
c. A change of unit pointer.

H.1 A GROUP POINTER

A group pointer has three fields:

a. A cluster count
b. A folded checksum
c. A cluster address within a unit. The width of each field may be specified at refresh time; therefore, the same code can handle a wider variety of sizes of devices.

The cluster count determines the number of consecutive clusters that can be described by one pointer. The folded checksum is computed for the first word of the first block of the group. Its main purpose is to catch hardware or software errors when the wrong block is read. The folded checksum is not a check on the hardware parity circuitry. The size of the cluster address field depends on the largest unit size in the file structure and on the cluster size. A cluster address is converted to a logical block address by multiplying by the number of blocks per cluster.

H.1.1 Folded Checksum Algorithm

This algorithm takes the low order n-bit byte, repeatedly adds it to the upper part of the word, and then shifts. The code is:

```
LOOP:   ADD     T1,T
        LDB     T,LOW ORDER N BITS OF T1
        LSH     T1,-N                      ;RIGHT SHIFT BY N BITS
        JUMPN   T1,LOOP
        DONE                               ;ANSWER IN T
```

This scheme elminated the usual overflow problem associated with folded checksums and terminates as soon as there are no more bits to add.

H.2 END-OF-FILE POINTER

The EOF is indicated by a zero word.

H.3 CHANGE OF UNIT POINTER

A file structure may comprise more than one unit; therefore, the retrieval information block must indicate which unit the logical block is on. Because a file can start on one device and move to another, a method of indicating a change from one unit to another in the middle of the file is necessary. To show this movement, a zero count field indicates that the right half of the word specifies a change in unit. A zero count field contains a unit number with respect to the file structure. The first retrieval pointer, with respect to the RIB, always specifies a unit number. Bit 18 is 1 to guarantee that the word is non-zero; otherwise it might be confused with an EOF pointer.

H.4 DEVICE DATA BLOCK

The monitor keeps a copy of up to 10 retrieval pointers in core at once. Therefore, if a file is allocated in 10 or less contiguous blocks (i.e., described in 10 or less pointers), all of the retrieval information can be kept in core and no additional accesses to the RIB are necessary.

H.5 ACCESS BLOCK

For each active file, the monitor keeps eight words of storage called an access block. These access blocks remain dormant in monitor core after a file is closed and are reclaimed only when the core space is necessary. Therefore, if a 4-word LOOKUP is done after a file has been active, access to the UFD and RIB blocks will not require I/O.

DEC-10-ULKMA-A-D

DECsystem-10

LINK-10

PROGRAMMER'S REFERENCE MANUAL

This document reflects the software as of Version 1.

1st Printing May 1973

COPYRIGHT © 1973 by Digital Equipment Corporation

The material in this document is for informational purposes and is subject to change without notice.

Actual distribution of the software described in this specification will be subject to terms and conditions to be announced at some future date by Digital Equipment Corporation.

DEC assumes no responsibility for the use or reliability of its software on equipment which is not supplied by DEC.

The software described in this manual is furnished to purchaser under a license for use on a single computer system and can be copied (with inclusion of DEC's copyright notice) only for use in such system, except as may otherwise be provided in writing by DEC.

TABLE OF CONTENTS

Chapter 1 INTRODUCTION TO LINK-10

1.1	Input to LINK-10	651
1.1.1	Relocatable Code	651
1.1.2	Symbols and Libraries	652
1.2	Output From LINK-10	653
1.3	Overlay Facility	654
1.4	Miscellaneous Features	655
1.5	Initialization of LINK-10	656
1.5.1	Using LINK-10 Automatically	656
1.5.2	Using LINK-10 Directly	657

Chapter 2 AUTOMATIC USE OF LINK-10

2.1	General Command Format	661
2.2	COMPIL Switches	662
2.3	Specifying Disk Areas Other Than SYS	667
2.4	SAVE and SSAVE System Commands	668
2.5	COMPIL Examples	669
2.6	Summary	673

Chapter 3 USING LINK-10

3.1	LINK-10 Command Strings	676
3.2	Changing Defaults	678
3.3	LINK-10 Switch Algorithms	679
3.3.1	Device Switches	680
3.3.2	File Dependent Switches	681
3.3.3	Output Switches	682
3.3.4	Immediate Action Switces	683
3.3.5	Delayed Action Switches	684
3.3.6	Switches that Create Implicit File Specifications	684
3.4	LINK-10 Switches	685

Chapter 4 LINK-1Ø SWITCHES

/BACKSPACE	691
/COMMON	692
/CONTENTS	694
/CORE	697
/COUNTER	698
/CPU	700
/DEBUG	701
/DEFAULT	703
/DEFINE	705
/ENTRY	707

LINK-10

/ERRORLEVEL 708
/ESTIMATE 709
/EXCLUDE 711
/EXECUTE 713
/FOROTS 714
/FORSE 715
/FRECOR 716
/GO 718
/HASHSIZE 719
/INCLUDE 721
/LOCALS 722
/LOG 723
/LOGLEVEL 725
/MAP 727
/MAXCOR 729
/MPSORT 731
/MTAPE 732
/NOINITIAL 734
/NOLOCAL 736
/NOSEARCH 737
/NOSTART 738
/NOSYMBOL 739
/NOSYSLIB 740
/OTS 742
/PATCHSIZE 744
/REQUIRE 746
/REWIND 747
/RUNCOR 748
/RUNAME 749
/SAVE 750
/SEARCH 752
/SEGMENT 753
/SET 754
/SEVERITY 755
/SKIP 756
/SSAVE 757
/START 758
/SYMBOL 759
/SYMSEG 761
/SYSLIB 763
/SYSORT 765
/TEST 766
/UNDEFINED 767
/UNLOAD 768
/VALUE 769
/VERBOSITY 770
/XPN 772
/ZERO 774

Chapter 5	LINK-10 MESSAGES	775
Chapter 6	LINK-10 EXAMPLES	803
Appendix A	LINK ITEM TYPES	
A.1	Link Item Types 0-37	816
A.2	Link Item Type 400 FORTRAN	828
A.3	Link Item Type 401 FORTRAN	829
A.4	Link Item Types 1000-1777	829
Appendix B	LOADER AND LINK-10 DIFFERENCES	843
GLOSSARY		849

FOREWORD

This manual is the reference document on the DECsystem-10 Linking Loader, LINK-10. It is aimed at the intermediate-level applications programmer and contains complete documentation on LINK-10, including descriptions of the LINK Item Types generated by the DECsystem-10 Translators.

Chapter 1 is an introduction to LINK-10 and describes the two methods of initializing the Linking-Loader. Chapter 2 discusses the automatic use of LINK-10 through the COMPIL-class commands, and Chapter 3 discusses the direct use of LINK-10 through the R LINK system command. LINK-10 switches are described in alphabetical order in Chapter 4. LINK-10 messages and examples appear in Chapters 5 and 6, respectively. The Appendices and Glossary contain supplementary information.

A beginning user of LINK-10 can benefit from this manual by reading Chapters 1 and 2, whereas an advanced user would be more interested in Chapters 3 and 4. A user who has been employing the LOADER program will find Appendix B a valuable aid in the transition to the LINK-10 program.

CHAPTER 1

INTRODUCTION TO LINK-10

LINK-10, the DECsystem-10 Linking Loader, is the system utility program that merges independently-translated modules of a user's program into a single module. Its main function is to prepare and link this input with other modules required by the user into a form that can be executed by the operating system.

1.1 INPUT TO LINK-10

LINK-10 accepts as its primary input the output from the DECsystem-10 translators in order to produce an executable version of the user's program. This output, known as object modules, is in the form of binary files which contain the user's programs and additional information generated by the translators. This additional information is necessary for linking separately-translated modules, for debugging, and for generating auxilary output such as map, log, and save files.

1.1.1 Relocatable Code

Most object modules contain relocatable code so that the module's position in core can be determined by LINK-10. Relocatable code is a benefit to both the user and the system. The user benefits because he is able to code all of his modules without regard to where they will be located in core. He need not be concerned with the location where one module ends and another one begins. The system benefits because a module written in relocatable code can be placed anywhere in core memory. When moving the relocatable object modules into the areas of

core memory at which they will be executed, LINK-10 adjusts all relocatable addresses in the modules into actual machine locations. In reality, LINK-10 places the modules in a user virtual address space (refer to the Glossary) and the operating system, as it schedules the usage of the system, transfers the modules to and from core memory. However, for simplicity, the user virtual address space is referred to as core memory in the remainder of the manual.

1.1.2 Symbols and Libraries

In addition to relocating and loading the user's object modules, LINK-10 is also responsible for linking these modules with other modules required for execution. Linkages among modules are provided through the use of symbols. By including symbols in his programs, the user is delaying the assignment of actual values until load time. This method of assigning values is advantageous because:

- It allows the user to change only the definition of the symbol instead of changing every occurrence of the value, and
- Only the module containing the definition of the symbol must be retranslated when a change occurs. Since other modules using the symbol are bound to it at load time, they do not have to be retranslated.

Although a user can define and use a symbol entirely within a single module, he usually refers to additional symbols that are defined in other modules. It is these modules that must be linked to the user's program for execution. In most cases, these required modules are contained in a library of relocatable binary programs. Modules within a library can either be created and translated previously by the user

or be part of the system's repertoire of programs. For instance, most higher-level languages have associated with them a library containing commonly-used mathematical, input/output, and data conversion routines. The user refers to modules in the library via symbols in his program and these symbols are then linked to the proper location in the library modules themselves. By linking these symbols and loading the required modules, LINK-10 provides communication between independently-translated modules and library routines.

In order to satisfy any undefined symbols, the required system libraries are usually searched after all loading specified by the user has been performed. However, the user can indicate that libraries be searched at a particular point in the loading procedure by specifying the appropriate switch to LINK-10 (refer to /SEARCH and /SYSLIB in Chapter 4). When LINK-10 processes the switch, the indicated libraries are searched and the required modules are loaded. The user also has the option of specifying by name which modules he wants (or does not want) loaded from a library or of inhibiting the search of the library altogether.

1.2 OUTPUT FROM LINK-10

When LINK-10 has performed the tasks of loading the user's object modules in core, bringing in and linking any other modules required for execution, and adjusting all the addresses, there is in core an executable version of the user's program. This executable version is the primary output of LINK-10. Since the loaded program at this point reflects the state of the user's core memory, it is usually referred

to as his core image. Having arrived at this state, the user can request LINK-10 to either:

- Transfer control to the core image for immediate execution (using the EXECUTE or START system commands, or the /DEBUG, /EXECUTE, or /TEST switches in LINK-10), or
- Output the core image to a device for storage (using the SAVE or SSAVE system commands, or the /SAVE or /SSAVE switches in LINK-10) in order to avoid the loading procedure in the future.

If the complete, loaded program is saved on a device in core image form, it can be brought into core and executed at a later time (using the GET and RUN system commands). The loading process does not have to be repeated since the results of all of LINK-10's actions are contained in the core image. However, if the user wishes to revise the modules that made up his core image, he must once again use LINK-10.

While the primary output of LINK-10 is the executable version of the user's program, the user can request auxilary output from LINK-10 in the form of map, log, save, symbol, and expanded core image files (XPN files). This additional output is not automatically generated by LINK-10 and the user must include the appropriate switches to obtain this output (refer to Chapter 4 for a description of the switches). This output is for the user's convenience when debugging his program.

1.3 OVERLAY FACILITY

LINK-10 will have an overlay facility to be used when the total core required by a program is more than the core available to the user. The user then organizes his program so that only some parts of the

program are required in core at any one time and the remaining parts are transferred in and out of core. During execution, these transferred parts are brought into core as required. The part brought into core overlays the part currently in that area. Because these parts of the program reside in the same area of core at different times, the amount of core required for the entire program is reduced.

1.4 MISCELLANEOUS FEATURES

LINK-10 has a large number of options in order that the user can gain precise control over the loading process. The user can set various loading parameters and can control the loading of symbols and modules. By setting switches in his input command strings to LINK-10, he can specify the core size of LINK-10 modules, the start address of modules, the size of the symbol table, the messages that he will see on his terminal or in his log file, and the severity level and verbosity of the messages. He can control the loading of modules by specifying the modules that should be loaded and the files that should be searched for symbol definitions. He has control over the number of segments to be allowed and the segment into which the symbol table will be placed.

The user has control over file specifications that LINK-10 examines to determine device names and filenames. He can accept the LINK-10 defaults for components in a file specification or he can set his own defaults which will be used automatically when he omits a component from his command string. He can also position devices, allocate space and assign protections to output files, and clear directories of DECtapes.

Some options available to the user are interactive. In the process of producing a core image, LINK-10 attempts to satisfy all requests for symbols defined in other modules and allows the user to interactively ask for a list of undefined symbols during the loading procedure. The user then has the opportunity to define them without reloading.

1.5 INITIALIZATION OF LINK-10

LINK-10 is initialized by the user in one of two ways:

- Automatically through the use of the LOAD, EXECUTE, or DEBUG system commands. This is the most common usage of LINK-10.
- Directly through the use of the R LINK system command. This is recommended for very large and relatively complex loading procedures.

1.5.1 Using LINK-10 Automatically

LINK-10 is automatically initiated when the user issues one of the system commands LOAD, EXECUTE, or DEBUG. These commands are known as COMPIL-class commands because they use the COMPIL program to control the actions of DECsystem-10 translators and LINK-10. COMPIL's job is to accept the command string typed by the user, interpret it, and construct and pass new command strings to various system programs, including the translators and LINK-10. This action taken by COMPIL is a convenience to the user since it saves him from typing the command strings to LINK-10. Once the command string to COMPIL is processed, the user does not interactively communicate with the translators or LINK-10. LINK-10 processes the appropriate command strings passed to

it by COMPIL and supplies intelligent defaults for any parameters not specified by the user. If LINK-10 obtains an error condition, it terminates the load and returns control to the operating system for further instructions. Otherwise, it loads the program and, depending on the COMPIL-class command used, either exits or starts the loaded program. Refer to Chapter 2 for the descriptions and use of the COMPIL-class commands.

In general, the extremely fine control of the loading process that is provided by manually running LINK-10 is not required for the average user because the COMPIL program supplies reasonable defaults to LINK-10.

1.5.2 Using LINK-10 Directly

Direct use of LINK-10 is useful for those who are developing large and complex programs, loading from devices other than disk, manipulating symbol tables for complex debugging situations, and performing segment manipulations.

The user runs LINK-10 directly by using the system command R LINK. LINK-10 responds with an asterisk which indicates that the user can type his input as a series of specifications which are to be used in the loading process. LINK-10 accepts input until the user specifies the exit condition; at which point it finishes all of its tasks and exits or begins the program, as specified by the user.

This method of running LINK-10 gives the user access to its full capability. The user does not have to accept LINK-10's default

conditions, but can supply his own set of defaults. He can interactively monitor the loading process by setting internal parameters, requesting values of particular items, specifying modules and files to be loaded, and controlling the format and contents of output files. Refer to Chapter 3 for the description of the LINK-10 command string, and Chapter 4 for the switches used when directly running LINK-10.

CHAPTER 2

AUTOMATIC USE OF LINK-10

The user causes LINK-10 to be run automatically whenever he types the LOAD, EXECUTE, and DEBUG system commands. These commands accept a simple command string format and are converted internally to a series of more complex command strings that are directly processed by various system programs, including language translators and LINK-10. The aforementioned commands are used to compile, load, and execute programs, to obtain output in the form of maps, to search files in library search mode, and to invoke the various debugging aids. The following paragraphs describe each of these system commands.

NOTE

> The information in this chapter is a subset of the material available on the LOAD, EXECUTE, and DEBUG commands. The subset presented here assumes that the source files have previously been translated, and thus only the switches directly applicable to loading the binary files are listed. Complete reference documentation on the COMPIL-class commands, their valid command formats, and all available switches can be obtained from the appropriate command descriptions in DECsystem-10 OPERATING SYSTEM COMMANDS, DEC-10-MRDC-D, located in the DECsystem-10 SOFTWARE NOTEBOOKS and in the DECsystem-10 USERS HANDBOOK, DEC-10-NGZB-D.

The LOAD command translates the user-specified source files into relocatable object modules (if necessary) and loads these object modules to form a core image. This command does not cause execution of the resulting core image. After completion of this command, the user can either execute his program (START system command) or save the core image (SAVE or SSAVE system command) for future execution.

LINK-10

Automatic Use of LINK-1Ø

The EXECUTE commmand translates the user-specified source files (if necessary), loads the object modules into a core image, and, in addition, begins execution of the program. The action of this command is the same as that of the LOAD command followed by the START system command.

The DEBUG command translates the user-specified source files (if necessary), loads the object modules into a core image, and prepares for debugging by additionally loading a system debugging program. Usually this debugging program is loaded first, followed by the user's program and other information required by the debugging program (e.g., the symbol table). However, when COBOL programs are being loaded, COBDDT (the COBOL debugging program) is loaded after the user's program. Upon completion of loading, control is transferred to the debugging program, rather than the user's program, so that the user can check out his program by examining and modifying the contents of locations. This examination and modification can occur both before program execution begins and during execution if the user specifies breakpoints in the program at which execution is to be suspended.

The debugging program can be COBDDT, MANTIS, or DDT, depending on the first source file in the command string. If the first file is a COBOL file, COBDDT (the COBOL debugging program) is loaded. If the first file is a FORTRAN source file, MANTIS (the FORTRAN debugging program) is loaded. (Note that MANTIS is under the control of an assembly conditional switch which is normally off. Therefore, the installation must turn this assembly switch on for the loading of MANTIS.) If the

first file is any other file, DDT (the Dynamic Debugging Technique) is loaded. When the first file has previously been compiled (i.e., the file has an extension of .REL, meaning relocatable binary object module), COMPIL does not determine the type of source file from which it came so DDT is loaded with the binary files. In this case, if the user desires COBDDT or MANTIS, he must explicitly specify this debugging program via the appropriate switch (refer to the /COBOL and /MANTIS switches in DECsystem-10 OPERATING SYSTEM COMMANDS).

2.1 GENERAL COMMAND FORMAT

The LOAD, EXECUTE, and DEBUG system commands have the same general command format. They all accept a list of file specifications.

```
LOAD output file spec = concatenated input file specs
EXECUTE output file spec = concatenated input file specs
DEBUG output file spec = concatenated input file specs
```

An input or output file specification consists of a device name, a filename with or without a filename extension, and a directory enclosed in square brackets. Only one output file specification can be given on the left of each equals sign, but any number of input file specifications can occur on the right. Input file specifications are separated from each other by commas or plus signs. If commas are used, the translator produces separate relocatable object modules for each output file. If plus signs are used, the input files separated by plus signs will be translated into a single relocatable object module. Plus signs must be used when a collection of files must be

concatenated to produce an acceptable module as input to a translator. The sequence of "output file spec = concatenated input file specs" can be given repeatedly in a command string by separating each sequence with a comma.

The output file specification and the equals sign can be omitted, in which case the object module is placed in the user's default directory on the disk with a name derived from the source file and the extension .REL. The filename given to the output file depends upon the form of the user's input file specifications. If the user has only one input file, the output file is given the name of the input file. If the user has more than one input file and the files are separated by commas, the name of each output file is the name of the corresponding input file. If the user has plus signs separating the file specifications, the name given to the output file is the name of the last input file in the series of files separated by plus signs.

2.2 COMPIL SWITCHES

Switches can be included on the LOAD, EXECUTE, and DEBUG command strings to direct LINK-10 in its processing. These switches are used to generate listings, to create libraries, to search user libraries, and to obtain loader maps. Each switch is preceded by a slash and can be either temporary or permanent. A temporary switch applies only to the file immediately preceding it. Characters (including spaces or commas) cannot separate the filename and the switch. A permanent switch applies to all files following it until modified by a subsequent switch. It is separated from the file it precedes by a space or a comma.

LINK-10

COMPIL Switches

LINK-10 switches described in Chapter 3 can be passed on the COMPIL-class command strings by preceding the switch specification with a % character instead of a / character. Following the % character is the LINK-10 switch specification preceded and followed by a delimiter. The delimiter can be any character; however, the user must be careful that the character he uses does not have a specific meaning to the COMPIL program. For example, the @ character indicates an indirect command file, and the semicolon causes the remainder of the line to be treated as a comment and thus ignored. The recommended delimiter is a single or double quote character. The beginning and ending delimiter must be the same character. A LINK-10 switch specification consists of the switch name and optionally a keyword and a value. The items in the specification are separated by colons. (Refer to Chapter 4 for the formats of the individual LINK-10 switches.) Note that LOADER switches (those beginning with a % but without enclosing delimiters) are illegal when passed to LINK-10. As an aid to users, a warning message is printed if the LINK-10 switch delimiter is one that could be interpreted as a LOADER switch (e.g.,A-Z,a-z,0-9,&, and -).

Since the first function of each of these three commands is to determine if the source files need translating (i.e., compiling or assembling), there are many switches that pertain to the translating process. The purpose of this manual is to describe the use of LINK-10 and switches pertaining to the translation of the source file are not

included. All switches that can be placed on the command string are described in DECsystem-10 OPERATING SYSTEM COMMANDS.

NOTE

Since currently there are two linking-loaders on the DECsystem-10, the user must indicate the desired loader when using the LOAD, EXECUTE, or DEBUG command. At the present time, the LOADER program is the default case, and the user must include the /LINK switch to indicate that he wishes to use the LINK-10 program. (The setting of the LOADER program as the default is a system parameter that can be changed by individual installations.) In the future, LINK-10 will become the standard default.

Table 2-1

COMPIL Switches Pertaining to Loading

/DDT	Loads DDT regardless of the extension of the first file in the command string. This is a permanent switch in that it applies to all subsequent files regardless of its position in the command string.
/FOROTS	Loads the file with FOROTS (the new FORTRAN object time system) instead of FORSE. This switch affects FORTRAN files only.
/FORSE	Loads the file with FORSE (the old FORTRAN object time system) instead of FOROTS. This switch affects FORTRAN files only.

Table 2-1 (Cont)

COMPIL Switches Pertaining to Loading

/LIBRARY	The action is identical to that of the /SEARCH switch. The use of the /SEARCH switch is recommended since it is the complement of /NOSEARCH.
/LINK	Causes the files to be loaded by the LINK-10 program instead of the LOADER program. If used, this switch must be placed before any file specifications (either implied or explicit) since the COMPIL program may have to generate load-control switches.
/LMAP	Produces a loader map during the loading process (same action as /MAP) containing the local symbols.
/LOADER	Causes the file to be loaded by the LOADER program instead of the LINK-10 program. Since this is the current default action, this switch is needed only if the installation has specified LINK-10 as the default linking-loader. In a future release, LINK-10 will become the standard default.
/MAP	Produces a load map during the loading process. The map does not contain local symbols. When this switch is encountered, a loader map is requested from LINK-10. After

Table 2-1 (Cont)

COMPIL Switches Pertaining to Loading

	the library search of the default system libraries, the map is written in the user's disk area with the filename specified by the user (e.g., /MAP:dev:file.ext[directory]) or the default filename (e.g., the name of the last program seen with a start address or nnnLNK.MAP (where nnn is the user's job number) if there is no such program). This switch is an exception to the permanent switch rule in that it causes only one map to be produced even though it may appear as a permanent switch.
/NOSEARCH	Loads all routines of the file whether the routines are referenced or not. Since this is the default action, this switch is used only to turn off library search mode (/LIBRARY or /SEARCH). This switch is not equivalent to the /NOSYSLIB switch of LINK-10, which does not search any libraries, including the default system libraries. The /NOSEARCH default is to search the default system libraries.
/SEARCH	Loads the files in library search mode. This mode causes a module in a special library file to be loaded only if one or more of its

Table 2-1 (Cont)
COMPIL Switches Pertaining to Loading

	declared entry symbols satisfies an undefined global request. The default system libraries are always searched regardless of the state of this switch.

2.3 SPECIFYING DISK AREAS OTHER THAN SYS

When translating his source files, the user has the option of selecting the disk area from which the language translator is obtained. The disk areas are [1,3] for OLD, [1,4] for SYS, [1,5] for NEW, and the user's area for DSK and are specified by the switches /OLD, /SYS, /NEW, and /SELF, respectively. (These four switches are described in DECsystem-10 OPERATING SYSTEM COMMANDS.) For example, if the user is translating his source files with a FORTRAN compiler that is on the OLD disk area of [1,3], he gives the following command string:

```
COMPILE/OLD FILEA.F4,FILEB.F4,FILEC.F4
```

The FORTRAN compiler is then obtained from area [1,3].

The first disk area seen in the command string is also the area from which LINK-10 is obtained. Thus, in the command string:

```
LOAD /LINK /OLD FILEA.F4,FILEB.F4,FILEC.F4
```

not only is the FORTRAN compiler obtained from OLD, but also the LINK-10 linking-loader. If LINK-10 is not found on the specified area, then the SYS disk area of [1,4] is searched. However, if the first disk area seen is the user's area (as indicated by the /SELF switch), only the areas specified in the user's job search list, which may include a user library (LIB), are searched. The searching does

not continue onto the NEW, OLD, and SYS areas. Thus, a user who is using a copy of a translator in his disk area but who does not have a copy of LINK-10 in that area must use two disk area specifications. For example,

LOAD /LINK /SYS /SELF FILEA.FOR,FILEB.FOR,FILEC.FOR

LINK-10 is obtained from the SYS disk area and the FORTRAN compiler from the user's disk area. Since SYS will be searched for LINK-10 on all disk specifications other than SELF, the user needs to specify two disk areas only when he is using a translator from his area.

2.4 SAVE AND SSAVE SYSTEM COMMANDS

After loading is completed, the loaded program may be written onto an output device so that it can be executed at some future date without rerunning LINK-10. The SAVE and SSAVE system commands output the core image onto the specified device as one or two files. If the SAVE command is used, the program will be nonsharable when it is later loaded into core. When the SSAVE command is used, the high segment (if any) of the program will be sharable when the program is loaded. The general command format of the two commands is the same:

```
SAVE dev:file.ext[directory]core

SSAVE dev:file.ext[directory]core
```

where

dev: is the name of the device on which to write the saved file. If omitted, DSK: is assumed.

file is the name of the saved file. If omitted, the job's current name is used. This name is set by the last R, RUN, GET, SAVE, or SSAVE system command, the last command which ran a

program (e.g., DIRECT), or the last SETNAM UUO.

.ext is the extension of the low segment file. If omitted, the following extensions are assigned:

If the program has one segment, the extension .SAV is assigned.

If the program has two segments, the low segment file has the extension .LOW, and the high segment file has the extension .HGH when a SAVE command is used and the extension .SHR when a SSAVE command is used.

[directory] is the area in which to save the file. If omitted, the user's default directory is used.

core is the amount of core in which to save the program. If omitted, the minimum required is assigned.

Refer to DECsystem-10 OPERATING SYSTEM COMMANDS for complete descriptions on the SAVE and SSAVE commands.

2.5 COMPIL Examples

In the following example, the user is translating, loading, and executing a MACRO program. The /LINK switch requests that the LINK-10 linking loader be used instead of the LOADER.

```
.EXECUTE /LINK SIMPLE.MAC )
MACRO: SIMPLE
LINK: LOADING
[EXECUTION]
THIS IS A VERY SIMPLE TWO-SEGMENT MACRO PROGRAM.

EXIT
```

LINK-10

COMPIL Examples

In the example below, the user is compiling, loading, and executing three COBOL programs. The /MAP:PROGMP.MAP switch requests the generation of a map file with the name PROGMP.MAP.

```
.EXECUTE /LINK /MAP:PROGMP FILA,FILB,FILC
COBOL: CBS08A  [FILA.CBL]
COBOL: CBS08B  [FILB.CBL]
COBOL: CBS08C  [FILC.CBL]
LINK:  LOADING
[EXECUTION]
RUNNING CBS08A
RUNNING CBS08B
RUNNING CBS08C

EXIT
```

The map file is now on the user's disk area. He can print the file with the following command:

```
.PRINT PROGMP.MAP
TOTAL OF 3 BLOCKS IN LPT REQUEST
```

The following is a listing of the map file generated.

```
                LINK-10 SYMBOL MAP OF   PROGMP          PAGE 1
        PRODUCED BY LINK-10 VERSION (32) ON  2-APR-73 AT  8:25:10

LOW  SEGMENT STARTS AT       0 ENDS AT    2416 LENGTH     2416 = 2K
        STARTING ADDRESS IS   1250, LOCATED IN PROGRAM CBS08A

                *************

JOBDAT-INITIAL-SYMBOLS

        ZERO LENGTH MODULE

                *************

LIBOL-STATIC-AREA
        LOW  SEGMENT STARTS AT     140 ENDS AT    1200 LENGTH     1040 (OCTAL), 544 (DECIMAL)

        .COMM.          140    COMMON         LENGTH         544     (DECIMAL)

                *************

CBS08A  FROM DSK:FILA.REL[27,235]       CREATED BY COBOL ON  2-APR-73 AT  8:24:00
        LOW  SEGMENT STARTS AT    1200 ENDS AT    1355 LENGTH      155 (OCTAL), 109 (DECIMAL)

        CBS08A          1270    ENTRY  POINT    RELOCATABLE

                *************

CBS08B  FROM DSK:FILB.REL[27,235]       CREATED BY COBOL ON  2-APR-73 AT  8:24:00
        LOW  SEGMENT STARTS AT    1355 ENDS AT    2040 LENGTH      463 (OCTAL), 307 (DECIMAL)

        CBS08B          1464    ENTRY  POINT    RELOCATABLE

                *************
```

```
CBSØ8C  FROM DSK:FILC REL[27,235]       CREATED BY COBOL ON  2-APR-73 AT  8:25:ØØ
        LOW  SEGMENT STARTS AT    2Ø4Ø ENDS AT    24Ø7 LENGTH      347 (OCTAL), 231 (DECIMAL)

        CBSØ8C           214Ø    ENTRY  POINT    RELOCATABLE

                **************

TRACED  FROM SYS:LIBOL.REL[1,4] CREATED ON 23-MAR-73 AT 16:4Ø:ØØ
        LOW  SEGMENT STARTS AT    24Ø7 ENDS AT    2416 LENGTH        7 (OCTAL), 7 (DECIMAL)

        BTRAC.           2412    ENTRY  POINT    RELOCATABLE
        CBDDT.           2413    ENTRY  POINT    RELOCATABLE
        PTFLG.           2414    GLOBAL SYMBOL   RELOCATABLE
        TRACE.           24Ø7    ENTRY  POINT    RELOCATABLE
        TRPD.            2412    ENTRY  POINT    RELOCATABLE
        TRPOP.           2412    ENTRY  POINT    RELOCATABLE

                **************

        [END OF LINK-1Ø MAP OF  PROGMP]
```

2.6 SUMMARY

The LOAD, EXECUTE, and DEBUG system commands, along with the switches described in Table 2-1, are sufficient for loading and executing most programs. The user can load separately-compiled programs and debugging programs, obtain maps, search files in a library search mode, and execute the program. To produce a saved file of his core image, the user can employ the system commands SAVE and SSAVE. More complex loading procedures can be performed by directly using LINK-10, as described in Chapter 3.

CHAPTER 3

USING LINK-10

The user runs LINK-10 directly by issuing the system command

R LINK

LINK-10 responds with an asterisk at which point the user types in his command strings. The LINK-10 program interprets all of the input typed by the user up to the end of the command string. A command string is defined as a series of characters terminated by a carriage return-line feed. A carriage return-line feed is generated when the user depresses the RETURN key on his terminal. The RETURN key is represented in this manual by the symbol ↵. If the user needs to continue a command string on another line, he can place a hyphen as the last non-blank, non-comment character before the carriage return-line feed. Continuation lines are considered part of the current command string, and the current string is not considered terminated until a carriage return-line feed is seen without a preceding hyphen. Comments may be added to any line by preceding the comment with a semicolon. Trailing spaces and tabs (including those before comments) are always ignored.

When the command string is terminated, LINK-10 processes the data in the command string by performing the actions specified by the user. This usually entails setting relevant internal conditions and storing information for later use. Each command string is completely scanned and processed before LINK-10 accepts a new one. After scanning and

processing the current command string, LINK-10 returns with another asterisk signifying its readiness to accept more input. The program accepts command string input until the user gives the exit condition switch (/GO) indicating that LINK-10 is to finish all loading tasks. At this point control is either returned to the operating system or given to the loaded program for execution, depending upon the preceding command strings.

3.1 LINK-10 COMMAND STRINGS

Command strings to LINK-10 contain a series of input and/or output file specifications and non-conflicting switches to direct the loading process. The general command string format is as follows:

```
*output specifications=input specifications
```

Any number of specifications can be included in the command string by separating each specification from other specifications with a comma. Although the equals sign is not required, it is recommended that the user include it so that he can distinguish his output specifications from his input ones. If the user does not include an equals sign, he must use a comma to separate the specifications. The input and output specifications are then distinguished by the type of switch associated with the specification, and the specifications can appear in any order (e.g., input specifications can precede output specifications).

An input or output specification consists of a file specification and switches appearing before and/or after the file specification. A file specification is in the form

dev:file.ext[directory]

and the individual switches that can be used in the command string are described in Chapter 4.

When items in a file specification are missing, LINK-10 has a set of initial values to be used as defaults. On input specifications, the default values assumed for missing items in a file specification are as follows:

Device	DSK:
Filename	A blank filename
Extension	.REL
Directory	The user's default directory

On output specifications, the default values are as follows:

Device	DSK:
Filename	Name of the last program containing a start address. If there is no program with a start address, the name nnnLNK, where nnn is the user's job number, is used.
Extension	Dependent on the type of output file requested via switches. Log file .LOG Map file .MAP Saved file .SHR,.HGH,.SAV,.LOW Symbol file .SYM Expanded save file .XPN
Directory	The user's default directory.

These defaults are applied just prior to initializing the device and opening the file, and are used only if the user has not given values for items in a file specification. The initial LINK-10 defaults for items in a file specification are used only when a value for the item

does not appear in the command string or until the value is seen if it is after the beginning of the string.

If a component of a file specification is given before the filename, it remains in effect until changed by a value given subsequently by the user for the same component or until the end of the command string. For example, a user can specify a device name at the beginning of the string and not have to repeat the device name for each specification if he is using the same device for all specifications in the command string. However, once the device name is changed, the new name is used as the default device for the reaminder of the command string.

As another example, the user can specify an extension and a directory to be used by issuing a command string such as

```
*.BIN[10,7]DSKB:FIL1,DSKC:FIL2.REL[10,20],DSKA:FIL3 ⤸
```

The extension .BIN and the directory [10,7] are used for any specifications that do not include an extension or directory. The above command string is equivalent to

```
*DSKB:FIL1.BIN[10,7],DSKC:FIL2.REL[10,20],DSKA:FIL3.REL[10,7] ⤸
```

3.2 CHANGING DEFAULTS

The /DEFAULT switch is used to change the initial values that are assumed when the user does not include a component of a file specification in his command string. The values specified with this

switch remain in effect for the entire load unless changed by another /DEFAULT switch. The form of the /DEFAULT switch is as follows:

components of file specification /DEFAULT:keyword

where

components of file specification are the components which the user wants as his default components.

keyword is either INPUT or OUTPUT to change the default components for the input or output specifications, respectively. If this argument is omitted, INPUT is assumed.

For example, the following specification

DSKB: .BIN[10,20]/DEFAULT

changes the values to be used as defaults for the input specifications to be DSKB: for the device, .BIN for the extension, and [10,20] for the directory.

NOTE

Because the extensions for output files depend upon the types of file being requested, the user cannot change the output extensions. Any attempt to do so is ignored.

3.3 LINK-10 SWITCH ALGORITHMS

LINK-10 allows the user to request various loading parameters via switches in the command string. Switches are used to specify output files, to set defaults, to control the loading of programs, to set values, to format maps and symbol tables, to request values of symbols, and to position devices. Some switches merely change the status of LINK-10 by setting internal values; others request immediate

action to be taken.

LINK-10 has several categories of switches with a specific algorithm for the handling of each category. These categories are:

- Device Switches
- File Dependent Switches
- Output Switches
- Immediate Action Switches
- Delayed Action Switches
- Switches that create implicit file specifications

3.3.1 Device Switches

Switches in this category (e.g., /SKIP, /REWIND) affect the device within an input or output specification. The switch is in effect after the device is initialized and, depending on its position, either before or after the file is read or written. If the switch appears before the filename, the appropriate action is taken before the file is processed, and if it appears after the filename, action is taken after the file is processed. Switches in this category apply only to the current input or output specification and do not carry over to subsequent devices. In other words, once the requested action is performed, it is not performed again unless another device switch is given.

For example, the following specification may be given by the user:

```
/SKIP:2 MTA1:MYFILE/UNLOAD,
```

After the magnetic tape is initialized, LINK-10 skips forward over two files (/SKIP:2), reads the file called MYFILE, and after reading the file, rewinds and unloads the tape (/UNLOAD).

3.3.2 File Dependent Switches

Switches belonging to this category (e.g.,/NOLOCAL, /SEARCH) modify the loading or the contents of a file. These switches are either temporary or permanent in nature. A temporary switch applies only to the file specification immediately preceding it. An intervening comma cannot separate the file specification and the switch. A permanent switch appears before the file specification and applies to all file specifications following it until modified by a subsequent switch or until the end of the current command string is reached. (Remember that continuation lines are considered part of the current command string). This means that permanent file-dependent switches, unlike device switches, continue to apply to following specifications (i.e., the action requested by the switch is not terminated at the comma which separates specifications).

For example, the following specifications may be issued by the user:

```
,/NOLOCAL DTA3:MAIN1,MAIN2,MYLIB/SEARCH,
```

Two files, MAIN1 and MAIN2, are loaded in their entirety from DTA3 without their local symbols. The file MYLIB is searched and parts of it are loaded only if required (i.e., they are required to satisfy any undefined symbol requests); if needed, they are also loaded without local symbols.

LINK-10

Switch Algorithms

3.3.3 Output Switches

Switches in this category (e.g.,/MAP, /LOG, /SAVE) initialize the output devices and create the output files. Each output specification must contain one of these switches because LINK-10 does not create output files unless explicitly requested to do so. Each switch represents a specific type of output file and is used with a file specification to indicate the device and filename of the file. Only one output switch can be used with each output specification. If the switch is the only item appearing in the output specification, the device name and filename are taken from the previous specification or from the LINK-10 defaults for output.

For example, if the user desires a saved file and a map file on DSKB: and both with the name OUTPUT, he can issue the following specifications:

```
DSKB:OUTPUT/SAVE,/MAP=
```

The two files will have the same filename (OUTPUT) but, by default, the extensions will be different (refer to Paragraph 3.1). The comma separating the two switches is required to indicate that two output files are desired. If the user is satisfied with accepting the LINK-10 defaults for output specifications, he can give the following

```
/SAVE,/MAP=
```

NOTE

Although the /LOG switch is considered an output switch, it is handled in a slightly different fashion from the remaining output switches. By assigning a device the logical name LOG before initializing LINK-10, the user receives the log file on the device assigned as LOG, even if he does not include the /LOG switch in his command string. The filename associated with the log file is nnnLNK.LOG, where nnn is the user's job number. The /LOG switch can then be used in the LINK-10 command string to change the filename of the log file. For example,

```
.ASSIGN DSKC:LOG:
.R LINK
*DSKC:MYLOG/LOG
```

renames the log file on DSKC: from nnnLNK.LOG to MYLOG.LOG. If the logical device is not assigned, then the building of the log file begins when the /LOG switch is seen. This results in the initialization timings not being included in the file.

3.3.4 Immediate Action Switches

Switches in this category (e.g., /UNDEF, /VALUE, /NOINITIAL, /NOSYM) are processed by LINK-10 as soon as they are seen. These switches are divided into two types:

- Those that request typeout from LINK-10.
- Those that change the status of the loading procedure.

Type-out switches (e.g., /UNDEF) request information from LINK-10 and are not dependent upon a particular specification. For this reason, they can appear anywhere in the command string but are usually on a command line by themselves because the user is interactively requesting information to determine if he may have forgotten to specify needed parameters. After processing the switch (i.e., at the

end of the command string), LINK-10 returns the requested information immediately. Once the information is returned to the user, the switch is cleared.

Status changing switches (e.g.,/NOINITIAL, /NOSYM) are related to the entire loading procedure and not to an individual specification. They are placed in the command string at the point at which the user wants the action to be performed. Once the action has been taken, it is in effect for the entire loading process and cannot be overridden. For example, once the user gives the /NOSYM switch to notify LINK-10 not to generate a local symbol table, he cannot, in the same load, give a switch to LINK-10 to nullify this action.

3.3.5 Delayed Action Switches

Switches in this category (e.g., /MAXCOR,/HASHSIZE) are used to change operational parameters of LINK-10 to the specified values. When the switch is seen, LINK-10 accepts the value but does not use it until it is needed. For example, there is a preset value for the maximum core LINK-10 can occupy during loading. Use of the /MAXCOR switch changes this value immediately but LINK-10 does not examine the value until it needs to expand its core size.

3.3.6 Switches that Create Implicit File Specifications

Switches in this category (e.g., /DEBUG, /SYSLIB) cause LINK-10 to create one or more input file specifications for programs that must be loaded along with the user's program and to set various other switches related to the implicitly specified file. As an example, the /DEBUG

switch indicates that a debugging program is to be loaded and that subsequent modules are to be loaded with local symbols, unless otherwise specified by the user. If one of these switches appears before the file specification, the program implied by the switch is loaded before the current file. If the switch is after the file specification, the program is loaded after the current file. Once the program implied by the switch is loaded, the switch is cleared.

3.4 LINK-10 SWITCHES

Switches to LINK-10 have one of the following forms:

```
/switch
/switch:arg
/switch:(arg,...,arg)
/switch:value
/switch:arg:value
/switch:(arg:value,...,arg:value)
```

where

/switch	is the name of the desired switch. This name can be truncated to a unique abbreviation. The first six characters of the name are sufficient to ensure uniqueness.
arg	is a keyword or a symbol name. Keywords can be truncated to a unique abbreviation.
value	is either a decimal or octal number. An octal value can be used with a switch that accepts decimal values by preceding the octal value with a number sign (#).
:	is the separator between components in a switch specification and must be present if more than one item is given.
( )	are used to enclose multiple keywords and/or values to a switch. They are required if more than one argument appears with the switch.

NOTE

For the first release of LINK-10, multiple keywords cannot be specified in a single switch specification. This means that the user must issue a switch specification for each desired keyword (e.g., /CONTENTS:LOCAL /CONTENTS: RELOCATABLE). This restriction will be removed in a later release of LINK-10.

Each switch specification must be terminated with a space; however, spaces cannot appear within a switch specification (i.e., between the slash and the end of the value).

Table 3-1 briefly describes the switches that can be used on the LINK-10 command string, and Chapter 4 contains the complete descriptions of the switches in alphabetical order.

Table 3-1

LINK-10 Switches

Switch	Meaning
/BACKSPACE	Spaces backwards over the specified number of files.
/COMMON	Allocates a COMMON area.
/CONTENTS	Specifies the types of symbols to be output in a map.
/CORE	Specifies LINK-10's initial low segment size.
/COUNTER	Lists the relocation counters and their values.
/CPU	Specifies the processor on which the program will run.
/DATA	Loads defined constant data. This switch is not implemented in Version 1.
/DEBUG or /D	Loads and specifies execution of a debugging program.
/DEFAULT	Changes default values for missing components in a file specification.
/DEFINE	Assigns values to undefined global symbols interactively.
/ENTRY	Lists library search symbols.
/ERRORLEVEL	Selectively suppresses messages to the terminal.
/ESTIMATE	Allocates disk space for an output file.
/EXCLUDE	Inhibits the loading of specified modules.
/EXECUTE or /E	Specifies execution of the program upon completion of loading.

Table 3-1 (Cont.)

LINK-10 Switches

Switch	Meaning
/FOROTS	Loads FOROTS, if required, during default system library searching.
/FORSE	Loads FORSE, if required, during default system library searching.
/FRECOR	Specifies the amount of free core guaranteed after each expansion.
/GO or /G	Terminates the loading progress.
/HASHSIZE	Specifies the size of the global symbol table.
/INCLUDE	Forces the loading of specified modules from a library.
/LOCALS or /L	Loads with local symbols.
/LOG	Causes a log file to be generated.
/LOGLEVEL	Suppresses messages to the log file.
/MAP or /M	Causes a map file to be generated.
/MAXCOR	Specifies LINK-10's maximum low segment core size.
/MPSORT	Sorts the symbol table for output to the map file.
/MTAPE	Performs magnetic tape functions.
/NOINITIAL	Clears the initial global symbol tables.
/NOLOCAL or /N	Loads without local symbols.
/NOSEARCH	Turns off user library search mode.

Table 3-1 (Cont.)
LINK-10 Switches

Switch	Meaning
/NOSTART	Ignores starting addresses.
/NOSYMBOL	Inhibits the generation of a symbol table in core.
/NOSYSLIB	Prevents a search of the default system libraries.
/OTS	Indicates the segment for the object time system.
/PATCHSIZE	Allocates patch space.
/REQUIRE	Generates global requests for the specified symbols.
/REWIND	Rewinds the DECtape or magnetic tape.
/RUNCOR	Assigns the initial low segment core size for the program.
/RUNAME	Assigns the program name.
/SAVE	Causes a saved file to be generated.
/SEARCH or /S	Turns on user library search mode.
/SEGMENT	Specifies the segment in which to load the modules.
/SET	Defines the values of a relocation counter.
/SEVERITY	Defines the fatality level of errors.
/SKIP	Spaces forward on a magnetic tape.
/SSAVE	Causes a sharable saved file to be generated.
/START	Specifies the start address of a program.
/SYMBOL	Causes a symbol file to be generated.

Table 3-1 (Cont.)
LINK-10 Switches

Switch	Meaning
/SYMSEG	Moves the symbol table to the specified segment.
/SYSLIB	Performs a search of the default system libraries.
/SYSORT	Sorts the symbol table for output to the symbol file.
/TEST	Loads a debugging program.
/UNDEFINED or /U	Types undefined global symbols on the terminal.
/UNLOAD	Rewinds and unloads the DECtape or magnetic tape.
/VERBOSITY	Specifies the amount of text to be printed for a message.
/VALUE	Lists the current values of the specified global symbols.
/XPN	Creates or saves the expanded core image file.
/ZERO	Clears the specified DECtape directory.

CHAPTER 4

LINK-10 SWITCHES

/BACKSPACE

Function

The /BACKSPACE switch is used to space backwards over the specified number of files. This switch has an effect only on tape devices and is ignored for non-tape devices.

Switch Format

/BACKSPACE:n

n is a decimal number representing the number of files to backspace over. If n is omitted, n=1 is assumed.

Category of Switch

Device Switch (refer to Paragraph 3.3.1)

Examples

,MTA0:/BACK:3,

Backspace MTA0 by three files.

LINK-10

Switches

/COMMON

Function

The /COMMON switch is used to allocate an area of storage of the specified size before loading any more code. An array of storage (a COMMON area) is reserved into which data can be placed in order that it may be shared by several programs and routines. Because the FORTRAN language contains a statement that reserves space for a COMMON area, this switch is used to reserve COMMON arrays when loading non-FORTRAN programs or to allocate a different size area than given via the COMMON statement in a FORTRAN program. However, if this switch is used to allocate a larger size area of the same name as that given in the FORTRAN program, the switch specification must be given before the FORTRAN program is loaded.

The name of each labeled area of COMMON storage is defined as an internal symbol whose value is the address of the first word of the COMMON area. These symbols may be used by other programs as external symbols.

Switch Format

/COMMON:name:n

Name is the symbolic name of up to six SIXBIT characters of the COMMON area. Blank COMMON is designated with the symbolic name ".COMM.".

n is a decimal number representing the size of the area in words.

Restrictions

Although various modules may redefine COMMON areas of the same name, the size of a COMMON area cannot be increased during the loading process. Therefore, the largest definition of a given COMMON area must be loaded first. Any attempt to increase the size of a COMMON area by redefinition will result in a fatal error. This applies to both modules defining COMMON areas and the /COMMON switch.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

/COMMON:.COMM.:1000

Allocate blank COMMON to be 1000 words.

/CONTENTS

Function

The /CONTENTS switch gives the user control over the contents of the map file by allowing him to specify the types of symbols to be included in the file. Each symbol is marked as to its type by the translator that processed the module containing the symbol. Some symbols may be of more than one type. For example, a symbol may be both a global symbol and a relocatable symbol. To insure the inclusion of such a symbol in the map file, the user must specify both the GLOBAL and the RELOCATABLE keywords in the /CONTENTS switch.

Each specification of the /CONTENTS switch is cumulative; keywords set by the first specification are not automatically cleared by the second specification. If the user desires to clear a keyword set in a previous specification, he must explicitly specify its complement.

NOTE

> This switch does not produce a map file. The user must specify the /MAP switch on an output specification in order to obtain the file. Unless the /MAP is given, the /CONTENTS switch has no meaning and is ignored.

Switch Format

/CONTENTS:keyword

/CONTENTS:(keyword,. . ., keyword)

LINK-10

Switches

Keywords are as follows:

ABSOLUTE	include all absolute symbols (usually flags, accumulators, and masks). Complement of NOABSOLUTE.
ALL	include all symbols. Complement of NONE.
COMMON	include all COMMON symbols. Complement of NOCOMMON.
DEFAULT	include the symbols according to LINK-10's default setting, that is: COMMON, GLOBAL, ENTRY, ABSOLUTE, RELOCATABLE, NOLOCAL, and NOZERO. This keyword is used to reset the /CONTENTS switch to the original default setting.
ENTRY	include all entry name symbols. Complement of NOENTRY.
GLOBAL	include all global symbols including COMMON and ENTRY symbols unless these symbols are suppressed with the NOCOMMON and NOENTRY keywords. Complement of NOGLOBAL.
LOCALS	include all local symbols. Complement of NOLOCAL.
NOABSOLUTE	do not include absolute symbols (i.e., turn off the condition corresponding to absolute symbols). Complement of ABSOLUTE.
NOCOMMON	do not include COMMON symbols. Complement of COMMON.
NOENTRY	do not include entry name symbols. Complement of ENTRY.
NOGLOBAL	do not include global symbols including COMMON and ENTRY symbols unless these symbols are requested with the COMMON and ENTRY keywords. Complement of GLOBAL.
NOLOCAL	do not include local symbols. Complement of LOCALS.
NONE	do not include any symbols of any kind. However, header information is still output in the map. Complement of ALL.
NORELOCATABLE	do not include relocatable symbols. Complement of RELOCATABLE.

LINK-10

Switches

NOZERO	do not include symbols from zero length programs. Complement of ZERO.
RELOCATABLE	include symbols that are relocatable (usually addresses). Complement of NORELOCATABLE.
ZERO	include symbols from zero length modules (usually parameter files). A zero length module is one which defines symbols but generates no code. Complement of NOZERO.

If the /CONTENTS switch is not specified, the default setting is COMMON, GLOBAL, ENTRY, RELOCATABLE, ABSOLUTE, NOLOCAL, and NOZERO. When the user specifies a keyword, the keyword is either added to the default setting or deleted from the default setting. For example, if the user issues the /CONTENTS:ZERO switch, the condition for symbols in zero length programs is added to the default setting. However, the keywords ALL, NONE, and DEFAULT reset the default setting to their respective meanings.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/CONTENT:ZERO,/CON:LOCAL,

Include in the map local symbols and symbols from zero length modules, in addition to the types of symbols in LINK-10's default setting.

LINK-10

Switches

/CORE

Function

The /CORE switch is used to specify the initial size of LINK-10's low segment. Generally, this size is less than or equal to MAXCOR (the maximum size of LINK-10's low segment). If the size specified in the /CORE switch is greater than MAXCOR, the core will be assigned. However, the next time LINK-10 needs to expand core, the size will be reduced to MAXCOR.

Switch Format

/CORE:n

n is a decimal number that represents the initial low segment core size for LINK-10. An octal value can be given by preceding it with a number sign (#). N is expressed in units of 1024 words or 512 words (a page) by following the number with K or P respectively. If K or P is omitted, K (1024 words) is assumed.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

/CORE:17K

Specify 17K words as the initial size of LINK-10's low segment.

LINK-10

Switches

/COUNTER

Function

The /COUNTER switch is used to output to the terminal the relocation counters, their initial and current values, and for undefined counters, the length of code depending on them. When a relocation counter is not known, a count of the amount of core used by the counter is kept so that loading can be resolved. Code depending on the counter is stored on the disk until the counter is defined.

Although LINK-10 is designed to handle an indefinite number of relocation counters to provide efficient program construction, the first release of LINK-10 only uses two relocation counters, the low segment counter (.LOW.) and the high segment counter (.HIGH.). These counters are listed in a map file with their initial and final values.

Switch Format

/COUNTER

Category of Switch

Immediate Action Typeout Switch (refer to Paragraph 3.3.4)

LINK-10

Switches

Examples

/COUNTER

RELOCATION COUNTER	INITIAL VALUE	CURRENT VALUE
.LOW.	0	140
.HIGH.	400000	400010

LINK-10

Switches

/CPU

Function

The /CPU switch is used to indicate the central processor on which the program will run once it has been loaded.

Switch Format

/CPU:keyword

Keyword is either KA10 or KI10. If the keyword is omitted, KA10 is assumed. If the /CPU switch is omitted, the machine on which the program is loaded is assumed.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/CPU:KI10

Run the program on the KI10 processor.

LINK-10

Switches

/DEBUG

Function

The /DEBUG switch is used to load a debugging program and to specify that execution of the load will begin at the normal start address of the debugging program instead of the user's program. The debugging programs available are DDT, MANTIS, and COBDDT. This switch does not cause termination of the loading procedure, the /GO switch is needed for termination. The /EXECUTE switch is not used for execution when the /DEBUG switch is given.

The /DEBUG switch turns on the load with local symbols mode and causes it to be in effect for the remainder of the load unless overridden by the /NOLOCALS switch. However, since the /NOLOCALS switch is file dependent, it is cleared at the end of the command string in which it appears and local symbols mode is reinstated. Note that the /LOCALS switch is also file dependent; therefore, the use of the /LOCALS switch and the implicit use of the /LOCALS switch in the /DEBUG switch context have different results (i.e., the /LOCALS switch is cleared at the end of the command string and the load with local symbols mode implied by the /DEBUG switch is not).

The /DEBUG switch does not cause the local symbols of the debugging program to be loaded, regardless of the state of the /LOCALS switch.

LINK-10

Switches

Switch Format

/DEBUG:keyword

Keyword is one of the following: COBDDT, COBOL, DDT, FORTRAN, MACRO, MANTIS. When a compiler or the assembler is specified, the debugging aid associated with that translator is used. For example, if MACRO is specified, the loading of DDT is implied. If the keyword is omitted, DDT is assumed.

Category of Switch

Creates an implicit file specification (refer to Paragraph 3.3.6)

Examples

,/DEBUG:DDT DTA3:FILEA.MAC,

LINK-10

Switches

/DEFAULT

Function

The /DEFAULT switch is used to change LINK-10's initially-assumed values for components missing in a file specification. A file specification is in the form dev:file.ext[directory]. The initial defaults for input specifications are

DSK:.REL [user's default directory]

and for output specifications are

DSK:name of main program.ext dependent on type of output file [user's default directory].

Thus, the user cannot change the extensions of output files, and any attempt to do so is ignored.

Values specified via the /DEFAULT switch are in effect for the entire loading process or until the user issues another /DEFAULT switch.

Switch Format

/DEFAULT:keyword

Keyword is either INPUT or OUTPUT to specify default conditions for input and output specifications, respectively. If the keyword argument is omitted, INPUT is assumed.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

LINK-10

Switches

Examples

DSK:MAIN,/DEFAULT .BIN[10,7],

Load the file MAIN.REL from the user's default directory of the disk and then change the input defaults to load .BIN files from the [10,7] area of the disk.

LINK-10

Switches

/DEFINE

Function

The /DEFINE switch is used interactively by the user to assign values to undefined global symbols in order to satisfy global requests before LINK-10 terminates the load with undefined symbols. The user can employ the /UNDEF switch to obtain a list of the undefined symbols and then use the /DEFINE switch to satisfy the requests for these symbols.

Switch Formats

/DEFINE:symbol:value

/DEFINE:(symbol:value, . . .,symbol:value)

Symbol is the name of the symbol to be defined. If the name given is one of an already-defined symbol, the user receives an error message.

Value is the decimal number to be associated with the symbol. An octal value can be given by preceding it with a number sign (#).

LINK-10

Switches

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

```
*/UNDEF↵
1 UNDEFINED SYMBOL
NOW 400123
*/DEFINE:NOW:897↵
*/DEFINE:OCT:#1234↵
```

LINK-10

Switches

/ENTRY

Function

The /ENTRY switch is used to type out all library search symbols (i.e., entry points) that have been loaded up to the time the switch is given. These symbols are recognized by a specific condition set in the first word of the symbol by the translator that processed the module containing the symbol. The user defines symbols as library search symbols with an ENTRY statement in a MACRO-10 or BLISS-10 module, with a SUBROUTINE, FUNCTION, or ENTRY statement in a FORTRAN module, or with a SUBROUTINE statement in a COBOL module.

This switch is useful for the future overlay facility of LINK-10.

Switch Format

/ENTRY

Category of Switch

Immediate Action Typeout Switch (refer to Paragraph 3.3.4)

Examples

```
*/ENTRY)
Library Search Symbols
SQRT.  3456
```

LINK-10

Switches

/ERRORLEVEL

Function

The /ERRORLEVEL switch is used to selectively suppress LINK-10 messages to the user's terminal. Associated with each message is a decimal number from 0 to 31 called the message level. Via this switch, the user can decide that messages with a message level less than or equal to a specific number are not to be output to his terminal. A user would normally want to suppress informative messages rather than error messages. The higher the message level, the more serious the message. Refer to Chapter 5 for the message level of each LINK-10 message.

Switch Format

/ERRORLEVEL:n

n is a decimal number from 0 to 30. Messages with a message level less than or equal to n will not be output to the terminal. Note that a message with a level of 31 cannot be suppressed. If this switch, or the value of the switch, is omitted, informative messages are suppressed.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/ERRORLEVEL:10

LINK-10

Switches

/ESTIMATE

Function

The /ESTIMATE switch is used to reserve disk space for an output file and must be associated with an output specification. Because each occurrence of the switch allocates space for only one file, the user must issue an /ESTIMATE switch for each file that needs space reserved.

This switch is not required for space allocation for an output file, but its use can both help the user stay within his quota allotment and reduce the number of (RIB) pointers associated with the file.

Switch Format

/ESTIMATE:n

n is a decimal number representing the estimated number of blocks of 128 words of the output file. A warning message is given if LINK-10 fails to allocate the requested size.

If this switch is omitted, or if an insufficient estimate is given, space is allocated automatically as needed.

LINK-10

Switches

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Examples

```
DSKC:OUTPUT/MAP/ESTIMATE:50,/SAVE/ESTIMATE:200,
```

Allocate 50 blocks for the map file and 200 blocks for the save file.

LINK-10

Switches

/EXCLUDE

Function

The /EXCLUDE switch is used to inhibit the loading of certain modules in a file when loading the file in the current mode (either search or nonsearch mode). This switch is useful when the user is searching a library file and definitely knows he does not want certan modules, even though his program may reference the names of these modules. For example, if a library file has several modules with the same library search symbols (e.g., as in dummy routines) and the user wants to load a module other than the first one, he can use this switch to prevent the loading of the modules not desired. Another use of the /EXCLUDE switch is to satisfy global symbol definitions during library searching by excluding the modules that would cause multiply-defined symbols.

Switch Formats

/EXCLUDE:symbol

/EXCLUDE:(symbol, . . ., symbol)

Symbol is the name of the module.

LINK-10

Switches

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

```
,/SEARCH LIBFIL.REL/EXCLUDE:(MOD1,MOD2),
```

Search the file LIBFIL as a library but do not load the modules MOD1 and MOD2 from the file, even if they are referenced.

LINK-10

Switches

/EXECUTE

Function

The /EXECUTE switch is used to specify that the loaded program is to be started at the normal entry point (i.e., the start address) upon completion of loading. This switch does not cause the termination of loading; the /GO switch is needed to terminate loading.

The /EXECUTE and /DEBUG switches cannot be used together because one switch specifies execution of the user's program (/EXECUTE) and the other switch specifies execution of the debugging program (/DEBUG).

Switch Format

/EXECUTE

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/EXE

LINK-10

Switches

/FOROTS

Function

The /FOROTS switch is used to specify the object time system FOROTS, instead of FORSE, for use with FORTRAN programs. FOROTS is then loaded, if required, when LINK-10 searches the default system libraries.

Switch Format

/FOROTS

Category of Switch

Creates an implicit file specification (refer to Paragraph 3.3.6)

Examples

,/FOROTS DSK:MAIN,SUB1,

LINK-10

Switches

/FORSE

Function

The /FORSE switch is used to specify the object time system FORSE, instead of FOROTS, for use with FORTRAN programs. FORSE is then loaded, if required, when LINK-10 searches the default system libraries.

Switch Format

/FORSE

Category of Switch

Creates an implicit file specification (refer to Paragraph 3.3.6)

Examples

,DSK:MAIN.F4/FORSE,

LINK-10

Switches

/FRECOR

Function

The /FRECOR switch guarantees that the specified amount of free core will remain after LINK-10 expands specific areas in its low segment. Since LINK-10's default amount of free core is 2K, users do not need this switch when loading most modules. However, when the modules being loaded are quite large (e.g., monitor modules), a larger amount of FRECOR will result in a faster loading process because LINK-10 will not have to move areas around in core as often.

During the loading procedure, LINK-10 has five areas that can be expanded beyond their initial sizes. These areas are: the user's low segment code area (LC), the user's high segment code area (HC), the local symbol table area (LS), the fixup area (FX), and the global symbol table area (GS). Each area has a lower boundary, a maximum upper boundary, and an actual upper boundary. LINK-10 tries to maintain space between the actual upper boundary and the maximum upper boundary at all times. However, as the loading procedure progresses, LINK-10 may have to expand an area to accomodate the user's input. If the sum of the amount of free core between the actual upper boundary and the maximum upper boundary for all areas minus the size required for the expansion is less than FRECOR, core is expanded to an amount large enough to maintain FRECOR. If the required size of the low segment becomes greater than MAXCOR (the user specified limit) or CORMAX

(the system limit) allows, no further expansion is attempted and core is obtained from the free space recovered by shuffling areas. When all of the free space has been obtained, some or all of the above-mentioned areas must overflow to the disk. Note that free core is not maintained when areas overflow to the disk.

Switch Format

/FRECOR:n

n is a decimal number representing the number of words of free core rounded to the next 128-word multiple. If this switch, or the value of this switch, is omitted, 2K words is assumed.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/FRECOR:3K

/GO

Function

The /GO switch is used to terminate the loading process and is the only termination switch available. When LINK-10 executes the /GO switch, it finishes loading the current specification, searches default libraries (if this action has not been suppressed with the /NOSYSLIB switch), produces the requested output files, and either exits to the monitor or runs the core image produced depending upon the switches appearing in the input command strings. If the /DEBUG switch has been specified, execution begins at the normal start address of the appropriate debugging program. If the /EXECUTE or /TEST switch has been specified, execution begins at the normal start address of the user's program. If one of these switches has not been specified, LINK-10 exits to the monitor.

Switch Format

/GO

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Example

```
/GO
EXIT
.
```

/HASHSIZE

Function

The /HASHSIZE switch is used to specify the initial size of the global symbol table. LINK-10 uses the lowest prime number in its internal list that is greater than or equal to the given value as the hashsize for the symbol table. This switch can be employed by a user who knows before loading that the number of global symbols used by his program is going to be quite large. By setting the hashsize of the symbol table to a larger number, the user can save LINK-10 time and space that would be used in expanding the hash table. When the user receives the message REHASHING GLOBAL SYMBOL TABLE on a load, it serves as an indication that he should use the /HASHSIZE switch at the beginning of subsequent loads of the same programs. Refer to the LINK-10 Design Specification for the hashing technique used in symbol tables.

Switch Format

/HASHSIZE:n

n is a decimal number representing the estimated hashsize of the global symbol table. A recommended hashsize is a number 1/3 larger than the total number of global symbols in the load. The default size (initially 127) is an assembly parameter.

LINK-10

Switches

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/HAS:1000

LINK-10 uses the prime number 1021.

LINK-10

Switches

/INCLUDE

Function

The /INCLUDE switch is used, when loading a file in search mode, to force the loading of specified modules in that file whether or not the user's program references them. For example, if the user does not have a global request for a desired module, he can use this switch to cause that module to be loaded.

Although the /INCLUDE switch is implemented in Version 1, its primary use is for the overlay facility in order to call a module.

Switch Format

/INCLUDE:symbol

/INCLUDE:(symbol, . . ., symbol)

Symbol is the module name of the desired module.

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

,SYS:LIB40/INCLUDE:(SIN,COS,TAN),

Search the library LIB40, but always load the modules SIN, COS, and TAN.

/LOCALS

Function

The /LOCALS switch is used to load local symbols with the specified programs. Local symbols are not processed by LINK-10, but are useful to the user when debugging.

This switch does not cause local symbols to be saved as part of the core image requested by the /SAVE or /SSAVE switch. The /SYMSEG switch or an entry in the JOBDAT location .JBDDT is required if local symbols are to remain in core.

Switch Format

/LOCALS

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

,MYFILE,/LOCAL MYDATA,MYSUB,MYLIB,

Load local symbols with the programs MYDATA, MYSUB, and MYLIB.

/LOG

Function

The /LOG switch is used to specify an output log file into which LINK-10 places information that is useful for the user when he is debugging his program. This file is a report of LINK-10's progress in loading the user's program because the actions taken by LINK-10 are shown. The times at which these actions took place are also indicated.

This switch is not required to obtain a log file if the user assigns a device the logical name LOG before running LINK-10. Then all log information will be recorded in a file on this assigned device. The file is named nnnLNK.LOG where nnn is the user's job number. In this case, the /LOG switch merely causes the file to be renamed to the user's specifications.

If the user does not assign a device the logical name LOG prior to running LINK-10, he must use the /LOG switch in order to obtain a log file. However, any times and messages output before the /LOG switch is seen in the command string will not appear in the log file.

Switch Format

file specification/LOG

File specification is in the form dev:file.ext[directory] to specify the device and name associated with the log file. The

default file specification is DSK:name of main program.LOG [user's default directory]. The user's terminal may be specified as the log device.

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Examples

DSKB:MYLOG/LOG

Create a log file on DSKB: with the name MYLOG.

/LOGLEVEL

Function

The /LOGLEVEL switch is used to suppress LINK-10 messages to the user's log file. This switch permits the user to set the level of messages that are to appear in the log file. Refer to the /ERRORLEVEL switch and Chapter 5.

If the log file is output to the user's terminal (i.e., the log device is the user's terminal), the messages output are determined by the lower of the arguments specified in the /ERRORLEVEL and /LOGLEVEL switches. The user would rarely set the log device as the terminal because the /ERRORLEVEL switch with a low number allows him to obtain all messages on the terminal.

Switch Format

/LOGLEVEL:n

n is a decimal number from 0 to 30. Messages with a message level less than or equal to n will not be output to the log file. The user cannot suppress messages with a level of 31. If this switch, or the value of the switch is omitted, a message level of 0 is assumed (i.e., all messages are output to the log file).

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/LOGLEVEL:5

Do not output any message to the log file with a message level less than or equal to 5.

LINK-10

Switches

/MAP

Function

The /MAP switch is used to specify an output map file which consists of the types of symbols requested by the user with the /CONTENTS switch. The map file is useful to the user when he is debugging his program because it lists the symbols used by his program along with their values. Header information (e.g., relocation counters with their lengths and starting addresses) is also included in the map.

Switch Format

file specification/MAP:keyword

File specification is in the form dev:file.ext [directory] and specifies the device and name associated with the map file. The default specification is DSK:name of main program.MAP[user's default directory].

Keyword is one of the following:

END to produce a map file at the end of loading.

ERROR to produce a map file of the code loaded if a fatal error occurs (i.e.,an error from which LINK-10 cannot recover).

NOW to produce a map file at the time this keyword is seen. The map contains all of the information up to and including the last file loaded. Default libraries will not be searched unless specified. This keyword is normally used during debugging to determine how the load is progressing.

If the /MAP switch is not issued by the user, no map file will be

generated. If the switch is given, but the keyword is omitted, the keyword END is assumed.

Category of Switch

Output Switch. Also, /MAP:NOW is an immediate action switch.

Examples

DSKB:MYMAP/MAP

Specify a map file on DSKB: with the name MYMAP.

/MAXCOR

Function

The /MAXCOR switch is used to specify the maximum amount of core LINK-10 may use as its low segment while loading. LINK-10 will expand to this size if required and then will overflow to the disk, rather than expanding in core, when it reaches the maximum core size allowed. When LINK-10 must overflow to the disk, it writes out part or all of the symbol area, the low code area, and/or the high core area in order that loading can continue. If the current amount of core used is greater than the size specified by the user, the next time LINK-10 requests more core, the size will decrease to the amount specified by the user and the remaining code will overflow to the disk. If the amount specified by the user is less than the minimum amount required by LINK-10, he receives a warning message indicating the amount required. He should then respecify the switch with a larger argument.

Switch Format

/MAXCOR:n

n is a decimal number that represents the maximum low segment core size for LINK-10. An octal value can be given by preceeding it with a number sign (#). N is expressed in units of 1024 words or 512 words (a page) by following the number with K or P respectively. If K or P is omitted, K (1024 words) is assumed.

The default size is all of available user core. The minimum size is dependent upon the code already loaded.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/MAXCOR:30K

Allow LINK-10 to expand its low segment to 30K before overflowing to the disk.

LINK-10

Switches

/MPSORT

Function

The /MPSORT switch is used to arrange the symbol table for output to the map file in the order most convenient to the user.

Switch Format

/MPSORT:keyword

Keyword is one of the following:

UNSORTED to print the symbols in the order in which they are placed in the symbol table. This keyword is the default.

ALPHABETICAL to arrange the symbol table in alphabetical order for each module or for each block in a block-structured module.

NUMERICAL to arrange the symbol table in numerical order according to the values of the symbols for each module.

NOTE

For the first release of LINK-10, UNSORTED is the only keyword implemented. The other keywords listed above are ignored and a warning message is output.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

MYMAP/MAP/MPSORT:UNSORTED

Specify a map file with the name MYMAP and print the symbols in the order in which they appear in the symbol table.

LINK-10

Switches

/MTAPE

Function

The /MTAPE switch allows the user to perform magnetic tape functions such as rewind, backspace, and skip. If this switch is given in an input specification, the action is performed immediately. However, when the switch is part of an output specification, the action requested is not performed until the output device has been initialized.

Switch Format

/MTAPE:keyword

Keyword is one of the following:

MTWAT	to wait for spacing and I/O to finish,
MTREW	to rewind the tape to load point.
MTEOF	to write an EOF,
MTSKR	to skip one record.
MTBSR	to backspace one record.
MTEOT	to space to the logical end-of-tape.
MTUNL	to rewind and unload the tape.
MTBLK	to write 3 inches of blank tape.
MTSKF	to skip one file.
MTBSF	to backspace one file.
MTDEC	to initialize for Digital-compatible 9-channel tape.

MTIND   to initialize for industry-compatible 9-channel tape.

Category of Switch

Device Switch (refer to Paragraph 3.3.1)

Examples

MTAØ:/MTAPE:MTEOT/MAP

Output the map file to MTAØ: after spacing to the logical end of tape (i.e., to the first free block).

/NOINITIAL

Function

The /NOINITIAL switch is used to clear LINK-10's initial global symbol table. This initial global symbol table consists of the .JBxxx symbols in JOBDAT. (Refer to DECsystem-10 Monitor Calls for a description of JOBDAT.) This switch is normally employed when the user is loading LINK-10 itself (in order to get the latest copy of JOBDAT), when the user wants to load a private copy of JOBDAT in order to use new values, or when the user is loading a program (for the purpose of creating a core image file) that will eventually run as an exec mode program (e.g., the monitor, diagnostics, a bootstrap loader). This switch must appear before the first file specification in the command string or else the initial LINK-10 global symbol table (JOBDAT) will be loaded. If the /NOINITIAL switch is specified, JOBDAT will be searched when the default system libraries are searched.

Switch Format

/NOINITIAL

If this switch is omitted, LINK-10's internal JOBDAT area symbols are used as the initial global symbol table.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

/NOINITIAL,COMMON,COMDEV,COMMOD,TOPS10/SEARCH/GO

Load the monitor without LINK-10's initial global symbol table.

/NOINITIAL,DTBOOT,EDDT/GO

Load the exec mode program without LINK-10's initial global symbol table.

LINK-10

Switches

/NOLOCAL

Function

The /NOLOCAL switch is used to load the programs without their local symbols. This is the default action.

Switch Format

/NOLOCAL

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

/LOCAL FIRST,SECOND,THIRD,FOURTH/NOLOCAL

Load the programs FIRST, SECOND, and THIRD with their local symbols and load the program FOURTH without its local symbols.

/NOSEARCH

Function

The /NOSEARCH switch is used to turn off library search mode (i.e., to always load the entire indicated file or files whether or not the files are required). The files are not searched to determine if they are needed. This switch is normally used after a /SEARCH switch has set library search mode. This is the default action.

Switch Format

/NOSEARCH

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

PARTA,/SEARCH LIBMAC,LIBCBL,LIBFOR,/NOSEARCH PARTB,PARTC

The files LIBMAC, LIBCBL, and LIBFOR are searched as libraries. The files PARTA, PARTB, and PARTC are loaded in their entirety.

/NOSTART

Function

The /NOSTART switch indicates to LINK-10 to ignore all start addresses in the binary input programs. The start address for the current program is not changed.

Switch Format

/NOSTART

If this switch is omitted and more than one start address is encountered, the last one seen is used.

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

MAIN1,/NOSTART MAIN2,MAIN3

Start addresses are ignored in files MAIN2 and MAIN3.

LINK-10

Switches

/NOSYMBOL

Function

The /NOSYMBOL switch signals LINK-10 not to construct a table of the symbols used by the user's program. This switch affects the speed of loading in that LINK-10 is not required to spend time in generating a symbol table for the user. If this switch is given, the user is not able to obtain output symbol files or output map files containing symbol listings. A map file can be obtained,however, with header information only.

Switch Format

/NOSYMBOL

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

/NOSYM

LINK-10

Switches

/NOSYSLIB

Function

The /NOSYSLIB switch is used to inhibit the searching of one or more of the system libraries upon completion of the loading process. The system libraries required by the loaded modules are usually searched at the end of the load in order to satisfy undefined global requests. These libraries are LIBOL for COBOL modules, FORLIB for FORTRAN-10 modules, LIB40 for F40 modules, and ALGLIB for ALGOL modules.

Switch Format

```
/NOSYSLIB:keyword
/NOSYSLIB:(keyword, . . .,keyword)
```

Keyword is one or more of the following:

ALGOL	to suppress the searching of ALGLIB.
BCPL	to suppress the searching of BCPLIB (not supported by DEC).
COBOL	to suppress the searching of LIBOL.
FORTRAN	to suppress the searching of FORLIB.
F40	to suppress the searching of LIB40.
NELIAC	to suppress the searching of LIBNEL (not supported by DEC).

If the keyword is omitted, the searching of all system libraries is suppressed.

LINK-10

Switches

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/NOSYSLIB:ALGOL/NOSYSLIB:COBOL

Do not search ALGLIB and LIBOL.

/NOSYSLIB

Do not search any system libraries.

LINK-10

Switches

/OTS

Function

The /OTS switch is used to indicate the segment into which the appropriate object time system is to be loaded.

Switch Format

/OTS:keyword

Keyword is one of the following:

DEFAULT to load the object time system into the segment specified by its code. FORTRAN, NELIAC, and ALGOL specify the high segment. This keyword is used to reset to normal conditions after specifying a /OTS switch with either the HIGH or LOW keywords.

LOW to load the object time system into the low segment.

HIGH to load the object time system into the high segment.

If this switch, or the value of this switch, is omitted, the default action is to load the object time system into the high segment unless either:

Code already exists in the high segment and /SEGMENT:HIGH is not set, or

The user has specified the /SEGMENT:LOW switch.

In these two cases, the object time system is loaded into the low segment.

LINK-10

Switches

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

FILA.REL/SYSLIB/OTS:HIGH

Load the required object time system into the high segment.

LINK-10

Switches

/PATCHSIZE

Function

The /PATCHSIZE switch is used to allocate space between the top of the loaded code and the bottom of the symbol table. This space is then used for new symbols defined by the user with DDT and/or for patching. Note that when the user defines symbols with DDT, each symbol will occupy two words. The space is allocated in either the high or low segment, depending upon the placement of the symbol table as specified with the /SYMSEG switch. The default is to place the symbol table in the low segment.

Switch Format

/PATCHSIZE:n

n is a decimal number representing the number of words to be allocated as patching space. An octal value can be given by preceding it with a number sign (#). A global symbol, PAT.., is defined to be equal to the first location in the patching system.

If this switch, or the value of this switch, is omitted, the default allocation is 64 (decimal) or 100 (octal) words.

LINK-10

Switches

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/SYMSEG:HIGH/PATCHSIZE:200

Load the symbol table into the high segment and allocate 200 words between the loaded code and the symbol table.

LINK-10

Switches

/REQUIRE

Function

The /REQUIRE switch is used to generate global requests for the indicated symbols. Thus, this switch can be used to load library modules out of their normal loading sequence or to force the loading of modules for overlays.

The /REQUIRE switch is used to load a module by specifying one or more of its library search symbols (entry points), whereas the /INCLUDE switch is used to load a module by specifying its name. Thus, the /REQUIRE switch is useful when the user knows the function he wants loaded (e.g., SQRT), but does not know the name of the module containing that function.

Switch Format

/REQUIRE:symbol

/REQUIRE:(symbol, . . .,symbol)

Symbol is the SIXBIT symbol name for which the user wants a global request generated.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

/REQUIRE:NAME

Generate a global request for the symbol called NAME.

LINK-10

Switches

/REWIND

Function

The /REWIND switch is used to rewind the current input or output device. The device associated with this switch must be a DECtape or magnetic tape. If the device is not a tape device, the switch is ignored.

Switch Format

/REWIND

Category of Switch

Device Switch (refer to Paragraph 3.3.1)

Examples

,/REWIND MTA0:,

LINK-10

Switches

/RUNCOR

Function

The /RUNCOR switch is used to specify the amount of core to be assigned to the low segment of the program when it is executed. The effect of this switch is identical to that produced when the program is run by the system run commands (R or RUN) with the given core argument.

Switch Format

/RUNCOR:n

n is a decimal number that represents the amount of core to be used as the initial core size for the program when obtained with the GET system command. An octal value can be given by preceeding it with a number sign (#). N is expressed in units of 1024 words or 512 words (a page) by following the number with K or P respectively. If K or P is omitted, K (1024 words) is assumed. If n is omitted or is less than the amount required, the number of blocks required by the core image area is assumed.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/RUNCOR:50P

LINK-10

Switches

/RUNAME

Function

The /RUNAME switch is used to assign the name to the program that is to be used while the program is running. This name is stored in a job-associated table in the Monitor and is used by the SYSTAT program and the VERSION system command. This switch affects high segment programs only.

Switch Format

/RUNAME:symbol

Symbol is the name to be assigned to the program. Only the first six characters specified are used. If this switch is omitted, the default name is the name of the module with the last start address. If there is no module containing a start address, the name used is nnnLNK, where nnn is the user's job number.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/RUNAME:PRIV,MYPROG/SSAVE

Save the file with the name MYPROG (i.e., MYPROG.SHR), but the program is run with the name PRIV.

/SAVE

Function

The /SAVE switch is used to define an output save file which will contain the core image generated by LINK-10. The core image is saved as one or two files: a low segment file and/or a high segment file. After the core image is saved on the specified output device, it can later be brought into core and executed as a non-sharable program (by using the RUN or GET system commands) without rerunning LINK-10.

Before writing low segment files (i.e., files with extensions .SAV or .LOW), LINK-10 compresses the core image by eliminating all zero blocks. High segment files are not compressed. This action is known as zero-compression and is used to save space on the storage device. The resulting zero-compressed file is, in essence, identical to the one produced by the SAVE system command.

Switch Format

file specification/SAVE:n

File specification is in the form dev:file[directory] and specifies the device and name associated with the save file. The default specification is:

DSK:name of main program.[user's default directory]

LINK-10

Switches

User-supplied extensions are ignored and the extension given to the file depends on the number of segments saved. If there is only one segment, the extension .SAV is used. If there are two segments, the extension .LOW is used for the low segment and .HGH for the high segment.

N is a decimal number that represents the amount of core (sum of high and low segments) in which the program is later to be run. An octal value can be given by preceding it with a number sign (#). N is expressed in units of 1024 words or 512 words (a page) by following the number with K or P respectively. If K or P is omitted, K (1024 words) is assumed.

If the /SAVE is not used, a save file will not be generated. If the switch is given but the core argument is omitted, the minimum core required by the core image is used.

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Examples

DTA3:MYPROG/SAVE:4K=

Define a save file on DTA3: with the name MYPROG. The program will be run in 4K.

LINK-10

Switches

/SEARCH

Function

The /SEARCH switch is used to turn on library search mode (i.e., to search specified files in order to load only those modules of the file that are required to satisfy undefined global requests). The user gives this switch to search either library files that he may have created or ones that are not part of the required system libraries. The /NOSEARCH switch is used to turn off library search mode. The required system libraries are still searched unless the user has inhibited the searching with the /NOSYSLIB switch.

Switch Format

/SEARCH

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

PARTA,/SEARCH LIBMAC,LIBCBL,LIBFOR,/NOSEARCH PARTB,PARTC

The files LIBMAC, LIBCBL, and LIBFOR are searched as libraries. The files PARTA, PARTB, and PARTC are loaded in their entirety.

LINK-10

Switches

/SEGMENT

Function

The /SEGMENT switch is used to indicate to LINK-10 the segment into which to load the input modules.

Switch Format

/SEGMENT:keyword

Keyword is one of the following:

DEFAULT to follow the specifications in the program. The typical case is to load pure code into the high segment and impure code into the low segment. This keyword is used to reset to normal conditions after specifying a /SEGMENT switch with either the HIGH or LOW keywords.

LOW to load code into the low segment.

HIGH to load code into the high segment, even if the code is impure.

If this switch, or the value of the switch, is omitted, high segment code is loaded into the high segment and low segment code into the low segment.

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

/SEGMENT:LOW TESTPRG,ANSWER,ROUTIN/SEGMENT:HIGH,

Load the modules TESTPRG and ANSWER into the low segment and the module ROUTIN into the high segment.

LINK-10

Switches

/SET

Function

The /SET switch is used to set the value of a relocation counter to a specified number. Although LINK-10 will handle many relocation counters, in the first release only two relocation counters are implemented: the counter for the low segment (.LOW.) which begins at zero, and the counter for the high segment (.HIGH.) which begins at location 400000 or the end of the low segment, whichever is greater. Other counters can be set, but they are currently not used by LINK-10.

Switch Format

/SET:symbol:n

Symbol is the name of the relocation counter.

n is an octal number representing the value of the counter. For the first release of LINK-10, only two relocation counters can usefully be given, .LOW. and .HIGH.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

.SET:.LOW.:1000,/SET:.HIGH.:400000

LINK-10

Switches

/SEVERITY

Function

The /SEVERITY switch specifies to LINK-10 the level at which messages are to be considered fatal. Associated with each message is a decimal number from 0 to 31 called the severity level. With this switch, the user can specify that messages with a severity level less than or equal to a specific number are not to cause his job to be terminated. Any message with a severity level above the specified number will cause his job to abort.

Switch Format

/SEVERITY:n

n is a decimal number from 0 to 30. LINK-10 messages with a severity level above n will cause a user's job to be aborted. Even though the highest severity level is 31, the user cannot indicate that a message with this severity level is to be considered non-fatal. If this switch, or the value of the switch, is omitted, a fatal error for a timesharing job is one whose severity level is greater than 24 (decimal), and for a batch job, one whose level is greater than 16 (decimal).

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/SEVERITY:30

LINK-10

Switches

/SKIP

Function

The /SKIP switch is used to space forward over the specified number of input or output files. This switch is implemented for magnetic tape only and is ignored if it is given for any other device.

Switch Format

/SKIP:n

n is a decimal number representing the number of files to skip over.

Category of Switch

Device Switch (refer to Paragraph 3.3.1)

Examples

/SKIP:4 MTA3:

/SSAVE

Function

The /SSAVE switch is used to define an output save file which will contain the core image produced by LINK-10. It is similar to the /SAVE switch except that the high segment will be sharable when it is brought into core and executed. The saved file produced by this switch is the same as the one produced by the SSAVE system command. Refer to the /SAVE switch.

Switch Format

file specification/SSAVE:n

Arguments are the same as for the /SAVE switch except for the following difference: when there are two segments, the extension .LOW is assumed for the low segment and .SHR for the high segment.

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Examples

DTA:SHRPRG/SSAVE,

Define a sharable save file with the name SHRPRG on the user's DECtape. The minimum core required by the core image is assigned.

LINK-10

Switches

/START

Function

The /START switch is used to specify the start address of the loaded program or to allow a program to specify its own start address. When a start address is specified, all subsequent start addresses are ignored. This is the default action.

Switch Format

/START:n

n is either of the following:

an octal number preceded by a number sign (#) representing the starting address of the program, or

a SIXBIT global symbol whose value is the start address. The global symbol specified must be defined.

If n is omitted, LINK-10 does not change the current start address but will accept all start addresses from the following modules (i.e., the action is to turn off a /NOSTART switch setting).

Category of Switch

File Dependent Switch (refer to Paragraph 3.3.2)

Examples

,MAINPG/START,/NOSTART PROG1,PROG2,

Use the start address in MAINPG and ignore the start addresses in PROG1 and PROG2.

/SYMBOL

Function

The /SYMBOL switch is used to specify an output symbol file which will consist of local symbols (if loaded), information stored in the local symbol table, such as module names and lengths, and global symbols sorted for DDT.

Via keywords, the user can specify that the symbol file is to be either in radix-50 representation or in triplet format. These two symbol table formats can be distinguished from each other in several ways:

1. The first word of the radix-50 symbol table is always negative. The first word of the triplet symbol table is always zero.

2. The listing of each radix-50 symbol requires two words; the first word is the symbol name in radix-50 representation, and the second word is the value.

3. The listing of each triplet symbol requires three words; the first one contains flags, the second is the symbol name in SIXBIT, and the third is the value.

This switch is useful when DDT is not loaded with the user's program because it guarantees that the symbols will be available. Note that if the user issues the /NOSYMBOL switch in the command string, he is not able to obtain the output symbol file.

LINK-10

Switches

Switch Format

file specification/SYMBOL:keyword

File specification is in the form dev:file[directory] and specifies the device and name associated with the symbol file. The default specification is

DSK:name of main program .SYM[user's default directory]

If there is no main program, the filename nnnLNK, where nnn is the user's job number, is used.

Keyword is one of the following:

RADIX-50 to obtain the symbols in radix-50 representation.

TRIPLET to obtain the symbols in triplet format.

If the /SYMBOL switch is not issued by the user, no output symbol file will be generated. If the keyword is omitted, RADIX-50 is assumed.

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Examples

DSKB:SYMFIL[20,235]/SYMBOL,

Define a symbol file with the name SYMFIL on the [20,235] area of DSKB:. The symbols will be output in the RADIX-50 format.

LINK-10

Switches

/SYMSEG

Function

The /SYMSEG switch causes symbols to be loaded with the program and indicates the segment into which the symbol table is to be placed. With this switch, the user insures that his program when loaded with DDT will run in as much core as is available without overwriting the symbol table. Loading DDT or setting the JOBDAT location .JBDDT to a non-zero value also causes the symbols to be loaded.

Switch Format

/SYMSEG:keyword

Keyword is one of the following:

DEFAULT to move the symbol table from its current position at the top of core to the first free location after the patching space. The JOBDAT location .JBFF, which points to the first free location, is adjusted to point to the first free location after the symbol table. This keyword is used to reset to the normal action after invoking the /SYMSEG switch with either the HIGH or LOW keywords.

HIGH to place the symbol table into the high segment.

LOW to place the symbol table into the low segment.

If the switch, or the value of the switch, is omitted, the symbol table is moved from its current position in the segment to the first free location in that segment. The first free location is determined after the allocation of space (default allocation is 64 decimal or 100 octal words) for patching of symbols. A global symbol, PAT.., is defined to be equal to the first location in

the patching space.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/SYMSEG:HIGH

/SYSLIB

Function

The /SYSLIB switch forces the system libraries to be searched in order to satisfy any undefined global requests. LINK-10 examines the main program first and, depending on the compiler used, searches the appropriate library (e.g., an ALGOL main program causes ALGLIB to be loaded). Then LINK-10 looks at any remaining programs and searches the relevant libraries.

A system library is not automatically searched unless its corresponding compiler-produced code has been loaded. This means that a user must explicitly request a system library when he is not loading the corresponding compiler-produced code for that library. For example, if the user is loading only MACRO-10 programs and he wants the LIB40 library searched, he must specify it in the switch format; LIB40 is not automatically searched unless F40 code has been loaded.

The normal action taken by LINK-10 is to search all required libraries at the end of the loading procedure; however, this switch without any keywords causes the libraries to be searched at the time the switch is given. If keywords are specified on the switch, the searcing of the indicated libraries occurs at the end of the loading procedure or on a subsequent /SYSLIB switch with no arguments, whichever occurs first.

LINK-10

Switches

Switch Format

/SYSLIB:keyword

/SYSLIB:(keyword, . . .,keyword)

Keyword is one of the following:

ALGOL	to search ALGLIB
BCPL	to search BCPLIB (not supported by DEC)
COBOL	to search LIBOL
FORTRAN	to search FORLIB
F40	to search LIB40 or FORLIB. The library searched depends upon the /FOROTS or /FORSE switch, if given, or on the default FORTRAN library, which is normally FORLIB, if neither switch is given.
NELIAC	to search LIBNEL (not supported by DEC)

If the keyword is omitted, only the libraries for which corresponding compiler-produced code has been loaded will be searched.

Category of Switch

Creates an implicit file specification (refer to Paragraph 3.3.6)

Examples

/SYSLIB

LINK-10

Switches

/SYSORT

Function

The /SYSORT switch is used to arrange the symbol table for output to the symbol file into the order most convenient to the user.

Switch Format

/SYSORT:keyword

Keyword is one of the following:

UNSORTED to leave the symbols in the order in which they are placed in the symbol table. This is the default.

ALPHABETICAL to arrange the symbol table in alphabetical order for each module or for each block in a block-structured module.

NUMERICAL to arrange the symbol table in numerical order for each module according to the values of the symbols.

NOTE

For the first release of LINK-10, UNSORTED is the only keyword implemented. The other keywords described above are accepted but LINK-10's action is the same as that taken with the UNSORTED keyword.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/SYSORT:UNSORTED

/TEST

Function

The /TEST switch is used to load a debugging program and to specify execution of the user's program. Thus, it is similar to the /DEBUG switch except that it specifies execution of the user's program instead of the debugging program. This switch does not cause termination of the loading; the /GO switch is required to terminate laoding.

Switch Format

/TEST:keyword

Keyword is one of the following: COBDDT, COBOL, DDT, FORTRAN, MACRO, MANTIS. When a compiler or the assembler is specified, the debugging aid associated with that translator is used (e.g., if MACRO is specified, the debugging program DDT is loaded).

Category of Switch

Creates an implicit file specification (refer to Paragraph 3.3.6)

Examples

```
,MAIN1,/TEST:COBOL DATPRG,DATA,TEST,
```

/UNDEFINED

Function

The /UNDEFINED switch is used to type all undefined global requests on the user's terminal. The user can employ this switch to determine the undefined symbols and then use the /DEFINE switch to satisfy the requests for these symbols. Thus, the user can interactively satisfy requests before LINK-10 terminates the load with undefined symbols.

Switch Format

/UNDEFINED

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples
----

*/UNDEF ↵

1 UNDEFINED SYMBOL

NAME   400100

400100 is a word in the chain of fixups depending on the symbol.

/UNLOAD

Function

The /UNLOAD switch is used to rewind and unload the current input or output device. The device associated with this switch must be a DECtape or a magnetic tape; the switch is ignored for non-tape devices.

Switch Format

/UNLOAD

Category of Switch

Device Switch; however, the action of this switch is always performed after the file is processed regardless of its position in the specification (refer to Paragraph 3.3.1)

Examples

,/REWIND DTA3:FILNAM/UNLOAD,

LINK-10

Switches

/VALUE

Function

The /VALUE switch allows the user to interactively type in the names of global symbols in order to find out their current values. The output given to the user consists of the requested symbol, its current value, and its status. The status can be one of: DEFINED (i.e., in the symbol table with its final value), UNKNOWN (i.e., not in the symbol table), UNDEFINED (i.e., in the symbol table as undefined), COMMON (i.e., in the symbol table and defined as COMMON).

Switch Format

/VALUE:symbol

/VALUE:(symbol, . . .,symbol)

Symbol is the name of the symbol in ASCII.

Category of Switch

Immediate Action Switch (refer to Paragraph 3.3.4)

Examples

```
*/VALUE:(TAG1,START)↵

TAG1     400010     DEFINED

START       0       UNDEFINED
```

The symbol TAG1 is defined to be the value 400010, and the symbol START is undefined.

LINK-10

Switches

/VERBOSITY

Function

The /VERBOSITY switch gives the user control over the amount of text transmitted to both his terminal and his log file whenever he receives a message from LINK-10. Associated with each message is a verbosity indicating the amount of text contained in the message. A verbosity of SHORT indicates that the message consists only of a 3-letter code (e.g., STC). A message with a verbosity of MEDIUM consists of the 3-letter code and one line that explains the code (e.g., STC Symbol Table Completed). A message with a verbosity of LONG consists of the 3-letter code, the one line of explanation, plus a more detailed explanation of the message. Thus, the user can specify via this switch the amount of explanation output to his terminal and log file.

LINK-10 has the following feature to aid users receiving fatal messages (i.e., ones preceded by ?). If the user receives a fatal message but has not indicated that he wants to see the detailed explanations (i.e., verbosity LONG), he can give the CONTINUE system command after he receives the message. LINK-10 then types out the remainder of the message (if there is more information available) on the user's terminal. This additional information is not included in the user's log file nor is the job continuable after the message is output.

LINK-10

Switches

Switch Format

/VERBOSITY:keyword

Keyword is one of the following:

SHORT	3-letter code only.
MEDIUM	3-letter code and a one-line explanation.
LONG	3-letter code, a one-line explanation, and a detailed explanation.

The default value is MEDIUM if this switch, or the keyword to the switch, is omitted.

If the user specifies a verbosity greater than the one available for the message, the specified keyword is ignored for that message and only the available text is output. For example, if the user specifies MEDIUM as the verbosity but the message only has a 3-letter code available (i.e., SHORT), only the 3-letter code will be output because there is no additional information available for that message.

Category of Switch

Delayed Action Switch (refer to Paragraph 3.3.5)

Examples

/VER:SHORT

LINK-10

Switches

/XPN

Function

The /XPN switch is used to create or save on the disk the expanded core image file (XPN file) of the low segment. If the program has not been loaded onto the disk, this switch causes the file to be created with the name specified by the user. If the program has been loaded onto the disk, the file already exists, but with the name nnnLLC.TMP where nnn is the user's job number. Since this extension indicates a temporary file, the expanded file is normally deleted upon the completion of LINK-10's processing. Thus, in this case, the /XPN switch is used to rename the file with the .XPN extension, so that it will not be deleted.

Switch Format

file specification /XPN

File specification is in the form dev:file[directory] and specifies the device and name to be associated with the expanded core image file. The default specification is

DSK:name of main program.XPN[user's default directory]

If there is no main program, the filename nnnLNK, where nnn is the user's job number, is used.

LINK-10

Switches

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Example

DSKC:XPNFIL[20,270]/XPN

Save the expanded core image file on the [20,270] area of DSKC: and with the name XPNFIL.

/ZERO

Function

The /ZERO switch is used to clear the directory of the associated DECtape. The directory is always cleared before the file is written, regardless of the switch's position in the current specification. This switch is ignored for all non-DECtape devices.

Switch Format

file specification/ZERO

File specification is an output specification.

Category of Switch

Output Switch (refer to Paragraph 3.3.3)

Examples

DTA3:MYPROG/SAVE/ZERO

CHAPTER 5

LINK-10 MESSAGES

The following table of LINK-10 messages consists of four columns: CODE, LVL, SEV, and MESSAGE. The leftmost column (CODE) contains a 3-letter code, which represents a terse, abbreviated form of the message. The user can indicate, via the /VERBOSITY:SHORT switch, that he desires only this code to be output whenever he receives a LINK-10 message. Refer to the /VERBOSITY switch in Chapter 4 for additional information.

The second column of each message (LVL) indicates the message level associated with that message. The message level is the factor that determines if the message is to be output. Normally, informative messages are suppressed to the user's terminal and all messages are output to the log file, if the user has designated one. However, the user can override this action with the /ERRORLEVEL and /LOGLEVEL switches. These switches accept a decimal number and indicate to LINK-10 that messages with a message level less than or equal to the specified number are not to be output to the user's terminal (/ERRORLEVEL) or to his log file (/LOGLEVEL). Messages with a message level greater than the specified number will be output. The two switches are independent if the user's log file is not being output to his terminal. That is, he can have one set of messages printed on his terminal and another set listed in his log file. When the device for the log file is the user's terminal, only one set of messages is output. This set is the one generated by the lower argument in either

the /ERRORLEVEL or /LOGLEVEL switch.

There are currently representations for three message levels:

%I message level 1 (informative)

%W message level 10 (warning)

%F message level 31 (fatal)

Refer to the /ERRORLEVEL and /LOGLEVEL switches in Chapter 4 for additional information.

The third column (SEV) contains the severity level associated with each message. The severity level is the point at which LINK-10 considers a message to be fatal (i.e., one which will terminate the load). The predefined LINK-10 severity levels can be overridden by the user via the /SEVERITY switch. This switch accepts a decimal number and indicates to LINK-10 that messages with a severity level less than or equal to the specified number are not to be considered fatal. Messages with a severity level greater than the specified number will cause the load to be terminated. (Note that messages with a severity level of 31 are always fatal and that the user cannot override the action taken with these messages.) If the user does not give a /SEVERITY switch, or does not give an argument to the switch, a severity level of 24 is considered fatal for a timesharing job and a severity level of 16 is considered fatal for a batch job.

Currently the representations for the severity levels are as follows:

%I severity level 1. The message is enclosed in square brackets (informative).

%W severity level 10. The message is preceded by a percent sign (warning).

%E severity level 30. The message is preceded by a percent sign and followed by a line requesting the user to re-edit the current file specification, if he wishes. This option is available only to a time-sharing user (editing).

%F severity level 31. The message is preceded by a question mark (fatal).

Refer to the /SEVERITY switch in Chapter 4 for additional information.

The rightmost column (MESSAGE) contains a more detailed explanation of the message than the one appearing in the CODE column. This message, along with the three-letter code, is normally output. However, the user can override this action with the /VERBOSITY switch. Refer to the /VERBOSITY switch in Chapter 4 for further information.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
ANC	%F	%F	ADDRESS NOT IN CORE (1) LINK-10 expected a particular user address to be in core, but it is not there. This is a LINK-10 internal error.
AZW	%F	%F	ALLOCATING ZERO WORDS (1) LINK-10's space allocator was called with a request for zero words. This is an internal error in LINK-10.
CEF	%F	%F	CORE EXPANSION FAILED (1) All attempts to obtain more core, including writing files onto disk, have failed.
CLF	%I	%I	CLOSING LOG FILE, CONTINUING ON [file specification] This message occurs when the user changes the device on which the log file is being written. The log file is closed on the first device and the remainder of the file is written on the second device.
CMF	%F	%F	COBOL MODULE MUST BE LOADED FIRST The COBOL-produced file must be the first file loaded when loading COBOL modules. COBDDT, the COBOL debugging program, or any other modules, such as a MACRO routine, cannot be the first file in the command string. The user should begin loading again and place the COBOL main program or routine as the first file in the command string.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
CNW	%F	%F	CODE NOT YET WRITTEN AT [label] (1) The user attempted a feature that is not yet implemented. This is an internal error in LINK-10.
CSF	%I	%I	CREATING SAV FILE LINK-10 is generating the requested save file by running the core image through a zero compressor routine in order to produce a SAV format file.
DNS	%I	%I	DEVICE NOT SPECIFIED FOR /switch A device switch, such as /REWIND or /BACKSPACE, has been given, but there is no device to be associated with it. The switch is ignored. This occurs when the user does not give a device name in the specification containing the switch or has not specified a device name in the current line. (Remember that devices are cleared at the end of the line.) LINK-10's default device DSK does not apply to device switches nor does a device specified in a /DEFAULT switch apply. The user should respecify the command line and include the appropriate device name with the switch.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
DRC	%W	%W	DECREASING RELOCATION COUNTER [symbol] FROM [value] TO [value] The user is reducing the size of an already defined relocation counter via the /SET switch. The new value is accepted. The user should be extremely careful when he does this because code previously loaded under the old relocation counter may be overwritten. This practice of reducing counters is dangerous unless the user knows exactly where modules are loaded.
DSO	%F	%F	DATA STATEMENT OVERFLOW (1) Incorrect code has been generated by the F40 compiler.
DUZ	%F	%F	DECREASING UNDEFINED SYMBOL COUNT BELOW ZERO (1) On an internal check of the counter for undefined symbols, LINK-10 determined that the counter was negative. This is an internal error.
EID	%F	%F	ERROR ON INPUT DEVICE STATUS (xxxxxx) FOR [file specification] A read error has occurred on the input device. Use of the device is terminated and the file is released. The status is represented by the right half of the file status word. Refer to DECsystem-10 Monitor Calls for the explanation of the file status bits.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{ELC, EHC, ELS, EFX, EGS}	%F	%F	ERROR CREATING OVERFLOW FILE FOR AREA {LC, HC, LS, FX, GS}
			LINK-10 could not make the named file on the disk (LC=user's low segment code, HC=user's high segment code, LS=local symbol table, FX=fixup area, and GS=global symbol table). The user could be over quota, or the disk could be full or have errors.
EMS	%I	%I	END OF MAP SEGMENT
			Notification that the LINK-10 module LNKMAP has completed the writing of the map file. The map is now closed.
ESN	%F	%F	EXTENDED SYMBOL NOT EXPECTED (1)
			The code to handle symbols longer than six characters has not been completed. This code will be available in a future release.
EXP	%I	%I	EXPANDING LOW SEGMENT TO [n] K
			LINK-10 needs more core and is expanding to the specified amount. In future loads of the same programs, the user can run LINK-10 more efficiently by requesting this amount of core at the beginning of the load with the /CORE switch.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
EXS	%I	%I	EXIT SEGMENT LINK-10 is entering the completion stages of the loading process. These stages include the creation of save and symbol files and, if required, the execution of the core image.
FCD	%F	%F	FORTRAN CONFUSED ABOUT DATA STATEMENTS (1) Incorrect code was generated by the F40 compiler for a data statement in the form DATA A(I),I=1,4/1,2,3,4/ as opposed to a data statement in the form DATA (A(I),I=1,4)/1,2,3,4/
FCF	%I	%I	FINAL CODE FIXUPS LINK-10 is now reading the low and/or high segment overflow files backwards in order to do all remaining code fixups. This process may cause considerable disk overhead. Note that the message occurs only if the load was too large to fit entirely in core.
FIA	%F	%F	CANNOT MIX KI10 AND KA10 FORTRAN-10 COMPILED CODE The FORTRAN-10 compiler generates different output for the KA10 and the KI10 processors (e.g., double precision code) and the user cannot load this mixture. He should decide which processor he wants to use and then recompile the appropriate programs.

---

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
FIN	%I	%I	LINK-10 FINISHED LINK-10 has completed its task of loading the user's program and other required programs. Control is either returned to the monitor or given to the user's program for execution.
FON	%F	%F	CANNOT MIX F40 AND FORTRAN-10 COMPILED CODE Output from the F40 and FORTRAN-10 compilers cannot be used together in the same load. The user should decide which compiler he wants and then recompile the appropriate program with that compiler.
{FEE / FRE}	%F	%F	{ENTER / RENAME} ERROR (0) ILLEGAL FILENAME FOR [file specification] One of the following conditions occurred: 1. The filename given was illegal. 2. When updating a file, the filename given did not match the file to be updated. 3. The RENAME UUO following a LOOKUP UUO failed.
{FLE / GSE}	%F	%E	{LOOKUP / GETSEG} ERROR (0) FILE WAS NOT FOUND The file requested by the user was not found. The user should respecify the correct filename.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{FEE FLE FRE GSE}	%F	%E	{ENTER LOOKUP RENAME GETSEG} ERROR (1) NO DIRECTORY FOR PROJECT-PROGRAMMER NUMBER FOR [file specification]
			The UFD does not exist on the named file structure, or the project-programmer number given was incorrect.
{FEE FLE FRE GSE}	%F	%E	{ENTER LOOKUP RENAME GETSEG} ERROR (2) PROTECTION FAILURE FOR [file specification]
			The user does not have the correct privileges to access the named file.
FRE	%F	%E	ENTER ERROR (2) DIRECTORY FULL
			The directory on the DECtape has no room for the file.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (3) FILE WAS BEING MODIFIED FOR [file specification]
			Another user is currently modifying the named file. The user should try accessing the file later.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (4) RENAME FILENAME ALREADY EXISTS FOR [file specification] (1)
			The specified filename already exists, or a different filename was given on the ENTER UUO following a LOOKUP UUO.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
FEE / FLE / FRE / GSE	%F	%F	ENTER / LOOKUP / RENAME / GETSEG ERROR (5) ILLEGAL SEQUENCE OF UUOS FOR [file specification] (1)
			The user specified an illegal sequence of monitor calls, UUOs, (e.g., a RENAME without a preceding LOOKUP or ENTER, or a LOOKUP after an ENTER).
FEE / FLE / FRE / GSE	%F	%F	ENTER / LOOKUP / RENAME / GETSEG ERROR (6) BAD UFD OR BAD RIB FOR [file specification] (1)
			One of the following conditions occurred: 1. Transmission, device, or data error occurred while attempting to read the UFD or RIB. 2. A hardware-detected device or data error was detected while reading the UFD RIB or UFD data block. 3. A software-detected data inconsistency error was detected while reading the UFD RIB or file RIB.
FEE / FLE / FRE / GSE	%F	%F	ENTER / LOOKUP / RENAME / GETSEG ERROR (7) NOT A SAV FILE FOR [file specification] (1)
			The named file is not a core image file. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.

---

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (10) NOT ENOUGH CORE FOR [file specification] (1) The system cannot supply enough core to use as buffers or to read in a program. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (11) DEVICE NOT AVAILABLE FOR [file specification] (1) The device indicated by the user is currently not available. This message can never occur and is included only for completeness of the LOOKUP, ENTER and RENAME error codes.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (12) NO SUCH DEVICE FOR [file specification] (1) The device specified by the user does not exist. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (13) NOT TWO RELOC REG CAPABILITY FOR [file specification] (1) The machine does not have a two-register relocation capability. This message can never occur and is included only for completeness of the LOOKUP, ENTER and RENAME error codes.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (14) NO ROOM OR QUOTA EXCEEDED FOR [file specification] There is no room on the file structure for the named file, or the user's quota on the file structure would be exceeded if the file were placed on the structure.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (15) WRITE LOCK ERROR FOR [file specification] The user cannot write on the specified device because it is write-locked.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (16) NOT ENOUGH MONITOR TABLE SPACE FOR [file specification] There is not enough table space in the monitor's (FILSER) 4-word blocks for the specified file. The user should try running the job at a later time.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{FEE, FLE, FRE, GSE}	%W	%W	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (17) PARTIAL ALLOCATION ONLY FOR [file specification] Because of the user's quota or the available space on the device, the total number of blocks requested could not be allocated and a partial allocation was given.
{FEE, FLE, FRE, GSE}	%F	%F	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (20) BLOCK NOT FREE ON ALLOCATION FOR [file specification] (1) The block required by LINK-10 is not available for allocation. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.
{FEE, FLE, FRE, GSE}	%F	%F	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (21) CAN'T SUPERSEDE (ENTER) AN EXISTING DIRECTORY FOR [file specification] (1) The user attempted to supersede an existing directory. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{FEE, FLE, FRE, GSE}	%F	%F	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (22) CAN'T DELETE (RENAME) A NON-EMPTY DIRECTORY FOR [file specification] (1) The user attempted to delete a directory that was not empty. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.
{FEE, FLE, FRE, GSE}	%F	%F	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (23) SFD NOT FOUND FOR [file specification] The required sub-file directory in the specified path was not found.
{FEE, FLE, FRE, GSE}	%F	%F	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (24) SEARCH LIST EMPTY FOR [file specification] A LOOKUP and ENTER UUO was performed on generic device DSK and the search list is empty.
{FEE, FLE, FRE, GSE}	%F	%F	{ENTER, LOOKUP, RENAME, GETSEG} ERROR (25) SFD NEST LEVEL TOO DEEP FOR [file specification] (1) An attempt was made to create a subfile directory nested deeper than the maximum level allowed. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (26) NO-CREATE ON FOR ALL SEARCH LIST FOR [file specification] No file structure in the job's search list has both the no-create bit and the write-lock bit equal to zero and has the UFD or SFD specified by the default or explicit path.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (27) SEGMENT NOT ON SWAP SPACE FOR [file specification] (1) A GETSEG UUO was issued from a locked low segment to a high segment which is not a dormant, active, or idle segment. This message can never occur and is included only for completeness of the LOOKUP, ENTER, and RENAME error codes.
{FEE FLE FRE GSE}	%F	%F	{ENTER LOOKUP RENAME GETSEG} ERROR (nn) UNKNOWN CAUSE FOR [file specification] (1) This message indicates that a LOOKUP, ENTER, or RENAME error occurred which was larger in number than the errors LINK-10 knows about.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
HSL	%F	%F	ATTEMPT TO SET HIGH SEGMENT ORIGIN TOO LOW The user is trying to set the beginning of the high segment below 400,000 or below the end of the low segment, whichever is larger. This can be the result of a /SET:.HIGH. switch with a value less than 400,000. If this is the case, the switch is ignored and the user should again specify the /SET:.HIGH. switch with a valid argument. This message can also occur when the low segment is greater than 400,000 and a module being loaded is requesting the high segment to start at 400,000. The user can either give a /SET switch or retranslate the module.
HSO	%W	%W	ATTEMPT TO CHANGE HIGH SEGMENT ORIGIN FROM [value] TO [value] The user is attempting to change the starting address of the high segment. The specified value is ignored. The cause may be that the user gave a /SET:.HIGH.: value switch which does not agree with the LINK item type 3, or that two LINK item type 3's have different origins. The user should recompile the incorrect files.
HTL	%F	%F	SYMBOL HASH TABLE TOO LARGE (1) The user has more global symbols than can fit in the maximum hash table (about 25K in size) LINK-10 can generate. Possible action is to increase the maximum allowable size of the hash table.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{I4D, I4S, I4T}	%F	%F	ILLEGAL F40 {DATA CODE, SUB-BLOCK, TABLE ENTRY} (xxxxxx) (1) Incorrect code was produced by the F40 compiler.
IBC	%F	%F	ATTEMPT TO INCREASE SIZE OF BLANK COMMON An attempt was made to expand the blank COMMON area. Once a COMMON area is defined, the size cannot be expanded. The user should load the module with the largest blank COMMON area first or specify the larger area with the /COMMON switch before loading either module.
ICI	%F	%F	INSUFFICIENT CORE TO INITIALIZE LINK-10 There is not enough core in the system to initialize LINK-10.
IDM	%F	%E	ILLEGAL DATA MODE FOR DEVICE The data mode specified for a device is illegal, such as dump mode for the terminal (e.g., TTY:/SAVE). The user should respecify the correct device.
IFD	%F	%F	INIT FAILURE FOR DEVICE [dev] The OPEN or INIT UUO failed for the specified device. The device could be in use by another user.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
ILC / IHC / ILS / IFX / IGS	%F	%F	ERROR INPUTTING AREA {LC / HC / LS / FX / GS} - STATUS (xxxxxx) An error occurred while reading in the named area (LC=user's low segment code, HC=user's high segment code, LS=local symbol table, FX=fixup area, and GS=global symbol table). The status is represented by the right half of the file status word. Refer to DECsystem-10 Monitor Calls for the explanation of the file status bits.
ILI	%F	%F	ILLEGAL LINK ITEM TYPE (xxxxxx) ON [file specification] The input file either was generated by a translator that LINK-10 does not recognize (e.g., a non-supported translator) or is not in proper binary format (e.g., is an ASCII or SAV file).
IMA	%I	%I	INCREMENTAL MAPS NOT YET AVAILABLE The INCREMENTAL keyword for the /MAP switch is not implemented. The switch is ignored.
INS	%F	%F	I/O DATA BLOCK NOT SET UP (1) LINK-10 attempted to do I/O (LOOKUP, ENTER UUOs) for a channel that has not been set up. This is an internal LINK-10 error.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
IPO	%F	%F	INVALID POLISH OPERATOR (1) An incorrect link item type 11 was seen. This is an internal LINK-10 error.
ISD	%F	%F	INCONSISTENT SYMBOL DEFINITION FOR [symbol] An already-defined symbol has been given a second "partial" definition. The user should examine the usage of the named symbol.
ISO	%F	%F	INCORRECT STORE OPERATOR (1) An incorrect link item type 11 was seen. This is an internal LINK-10 error.
ISP	%F	%F	INCORRECT SYMBOL POINTER (1) The current symbol pointer does not point to a valid symbol triplet. This can occur if an extended symbol does not terminate properly. This is an internal LINK-10 error.
IST	%F	%F	INCONSISTENCY IN SWITCH TABLE (1) An internal error occurred in the switch tables built by the SCAN module.
IVC	%F	%F	INDEX VALIDATION CHECK FAILED AT [address] (1) The range checking of LINK-10's internal tables and arrays failed. The address output is the point in the appropriate LINK-10 segment at which this occurred.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Message

CODE	LVL	SEV	MESSAGE
LDS	%I	%I	LOAD SEGMENT Indication that the LINK-10 module LNKLOD has started its processing.
LIM	%I	%I	LINK-10 INITIALIZATION LINK-10 has begun its processing of the user's input.
LIT	%F	%F	LINK ITEM TYPE (xxxxxx) TOO SHORT FOR [file specification] An error occurred in the named link item. This could result from incorrect output generated by a translator (e.g., no argument is seen on an END block when one is required). The user should retranslate the module.
LMN	%I	%I	LOADING MODULE [name] LINK-10 is in the process of loading the named module.
MDS	%W	%W	MULTIPLY-DEFINED GLOBAL SYMBOL [symbol] IN MODULE [name] DEFINED VALUE = [value], THIS VALUE = [value] The user has given an existing global symbol a value different from its original one. The second occurrence of the global symbol is in the named module. The currently defined value is used. The user should change the name of the symbol or reassemble one of the files with the correct parameters.
MNS	%I	%I	MAP SORTING NOT YET IMPLEMENTED Alphabetic and numeric sorting of the map file is not yet implemented. The symbols appear in the order in which they were placed in the symbol table.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
MOV	%I	%I	MOVING LOW SEGMENT TO EXPAND AREA[area] This message indicates that LINK-10 is making inefficient use of core. In future loads of the same programs, the user should allocate more core to LINK-10 at the beginning of the load. Area is one of the following: LC=user's low segment code, HC=user's high segment code, LS=local symbol table, FX=fixup area, and GS=global symbol table.
MPS	%I	%I	MAP SEGMENT Indication that the LINK-10 module LNKMAP has begun to write a map file.
MSS	%W	%W	MAXCOR SET TOO SMALL, INCREASING TO nK The current value of MAXCOR is too small for LINK-10 to operate. In future loads of this program, the user can save LINK-10 time by setting MAXCOR to this new expanded size at the beginning of the load.
MTS	%W	%W	MAXCOR TOO SMALL, AT LEAST nK IS REQUIRED The user specified the /MAXCOR switch with an argument that is below the minimum size LINK-10 requires as its low segment. The switch is ignored. The minimum size is dependent upon the code already loaded. The user should respecify the switch.
NCL	%W	%W	NOT ENOUGH CORE TO LOAD JOB, SAVED AS [file specification] The user's program was too large to load into core. Thus, LINK-10 created a saved file on disk and cleared user core. The user can perform a GET or RUN operation on the program to load it into core. If the core image is still too big, the user can either employ a bigger machine or obtain a larger core limit (e.g., increase CORMAX).

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
NCX	%W	%I	NOT ENOUGH CORE TO LOAD AND EXECUTE JOB, WILL RUN FROM [file specification] The user's program was too large to load into core and LINK-10 created a saved file on disk. It automatically executes the program by performing a RUN UUO. However, the saved file remains on disk and the user must delete it, if he wishes.
NED	%F	%E	NON-EXISTENT DEVICE [dev]: The user has specified a device that does not exist in the system. The user can re-edit the input files to correct the device name or type control-C to abort the load.
NYI	%W	%W	NOT YET IMPLEMENTED - /switch The user issued a switch that is not implemented in this version of LINK-10.
{OLC, OHC, OLS, OFX, OGS}	%F	%F	ERROR OUTPUTTING AREA {LC, HC, LS, FX, GS} -STATUS (xxxxxx) An error occurred while writing out the named area (LC=user's low segment code, HC=user's high segment code, LS=local symbol table , FX=fixup area, and GS=global symbol table). The status is represented by the right half of the file status word. Refer to DECsystem-10 Monitor Calls for the explanation of the file status bits.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
{OEL, OEM, OES, OEX}	%W	%W	OUTPUT ERROR ON {LOG, MAP, SYMBOL, XPN} FILE. FILE CLOSED. JOB CONTINUING - STATUS [xxxxxx] An error has occurred on the output file. The output file is closed at the end of the last data that was successfully output. The status is represented by the right half of the file status word. Refer to DECsystem-10 Monitor Calls for the explanation of the file status bits.
OFN	%F	%F	OLD FORTRAN (F40) MODULE NOT AVAILABLE The standard released version of LINK-10 includes the LNKF40 module that loads F40 code. However, the installation has removed it by loading a dummy version of LNKF40 and thus LINK-10 is unable to handle F40 compiler output. The user should request his installation to load a version of LINK-10 with the real LNKF40 module.
OMN	%F	%F	OBSOLETE MONITOR WILL NOT SUPPORT LINK-10 LINK-10 requires a monitor that contains the DEVSIZ UUO.
{PLC, PHC, PLS, PFX, PGS}	%I	%I	AREA {LC, HC, LS, FX, GS} OVERFLOWING TO DSK The job is too large to fit into the allowed core and the named area is being moved to disk (LC=user low segment code, HC=user high segment code, LS=local symbol table, FX=fixup area, and GS=global symbol table).

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
PSF	%F	%F	POLISH SYMBOL FIXUPS NOT YET IMPLEMENTED The requested feature is not yet available.
RCF	%F	%F	RELOCATION COUNTER TABLE FULL The relocation counter table is a fixed length and cannot be expanded in the current version of LINK-10. This restriction will be eliminated in a future release.
RED	%I	%I	REDUCING LOW SEGMENT TO [n] K LINK-10's internal tables have been deleted and core has been reclaimed. This message occurs near the end of loading.
RGS	%I	%I	REHASHING GLOBAL SYMBOL TABLE FROM [old size] TO [new size] LINK-10 is expanding the global symbol table either to the next prime number as requested by the user (via /HASHSIZE) or to its next expansion of about 50%. In future loads of this program, the user can save LINK-10 time by setting the hash table to this new expanded size at the beginning of the load.
SIF	%F	%F	SYMBOL INSERT FAILURE, NON-ZERO HOLE FOUND (1) An internal LINK-10 error. LINK-10's hashing algorithm failed. A symbol already exists in the location in which LINK-10 needs to place the new symbol. The error may disappear if the user loads the files in a different order.

----

(1) This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DEC.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
SFU	%I	%I	SYMBOL TABLE FOULED UP (1) An internal LINK-10 inconsistency. LINK-10 cannot locate the TITLE triplets in order to store the lengths of the control sections. The loading process continues. Any maps requested by the user will not contain the lengths of the control sections.
SNC	%F	%F	SYMBOL [symbol] ALREADY DEFINED, BUT NOT AS COMMON The user has defined a non-COMMON symbol with the same name as a COMMON symbol. The user should decide which symbol definition he wants. If he uses the COMMON definition, the COMMON area should be loaded first.
SNL	%I	%I	SCANNING NEW COMMAND LINE LINK-10 has completed the scanning and processing of the current command line and is ready to accept the input on the next line.
SOE	%F	%F	SAVE FILE OUTPUT ERROR - STATUS (xxxxxx) An error has occurred on the save file. The file is closed at the end of the last data that was successfully output. The status is represented by the right half of the file status word. Refer to DECsystem-10 Monitor Calls for the explanation of the file status bits.
SSN	%I	%I	SYMBOL TABLE SORTING NOT YET IMPLEMENTED Alphabetic and numeric sorting of the symbol table is not yet implemented. The symbols appear in the order in which they were placed in the symbol table.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
SST	%I	%I	SORTING SYMBOL TABLE LINK-10 is arranging the symbol table in the order specified by the user via the /SYSORT switch, and if required, is converting the symbols from the new to old format as indicated on the /SYMSEG, /SYMBOL, or /DEBUG switch.
STC	%I	%I	SYMBOL TABLE COMPLETED The symbol table has bee.. sorted and moved according to the user's request via the /SYMSEG, /SYMBOL, or /DEBUG switch.
T13	%F	%F	LVAR (TYPE 13) CODE NOT IMPLEMENTED LINK item type 13 (LVAR) is not implemented in LINK-10 nor supported by DEC. The TWOSEG pseudo-op in the MACRO-10 language should be used.
TDS	%W	%W	TOO LATE TO DELETE INITIAL SYMBOLS The /NOINITIAL switch was placed in the command string after the first file specification. Because this switch was not first in the command string, LINK-10's initial symbol table was loaded.
TEC	%F	%F	TRYING TO EXPAND COMMON An attempt was made to expand a COMMON area. The largest occurrence of the COMMON area of a given name must be linked first. Once defined, the size cannot be expanded although new COMMON areas of different names can be defined. The user should load the largest occurrence first.

LINK-10

Messages

CODE	LVL	SEV	MESSAGE
TSO	%F	%F	CANNOT LOAD TWO SEGMENT MODULE INTO ONE SEGMENT The user attempted to force two segments into one segment via the /SEGMENT switch. However, the binary file does not contain the necessary information (i.e., the length of the high segment) in LINK item type 3. This situation is usually caused by a one-pass compiler (e.g.,ALGOL).
URC	%I	%I	UNKNOWN RADIX-50 SYMBOL CODE Bits 0-3 of the first word of the link item contain an unknown symbol code. Either the translator is generating incorrect code or the binary file is bad. The user should recompile the file.
USA	%W	%W	UNDEFINED STARTING ADDRESS The user has given a global symbol as the start address and the symbol is currently undefined. The user should load the module that defines the symbol.

CHAPTER 6

LINK-10 EXAMPLES

EXAMPLE 1 Loading and Executing COBOL Programs

The following files are on the user's disk area:

```
.DIRECT

FILA     CBL  1    <055>          6-FEB-73   DSKB:  [27,235]
FILB     CBL  2    <055>          6-FEB-73
FILC     CBL  1    <055>          6-FEB-73
006LNK   LOG  1    <055>         28-FEB-73
SIMPLE   MAC  1    <055>         28-FEB-73

  TOTAL OF 6 BLOCKS IN 5 FILES ON DSKB: [27,235]
```

In the command string shown below, the user is automatically compiling, loading, and executing the programs and generating a map. The /CONT:ZERO switch is passed to LINK.

```
.EXECUTE /LINK/MAP FILA,FILB,FILC%'CONT:ZERO'
COBOL:    CBS08A    [FILA.CBL]
COBOL:    CBS08B    [FILB.CBL]
COBOL:    CBS08C    [FILC.CBL]
LINK:     LOADING
[EXECUTION]
RUNNING CBS08A
RUNNING CBS08B
RUNNING CBS08C

EXIT
```

In the followng command sequences the user is compiling the files and then directly loading and executing them through LINK-10.

LINK-10
Examples

```
.COM FILA,FILB,FILC ⤸
COBOL:    CBS08A    [FILA.CBL]
COBOL:    CBS08B    [FILB.CBL]
COBOL:    CBS08C    [FILC.CBL]

EXIT

.R LINK ⤸

*FILA,FILB,FILC,/MAP/CONT:ZERO/EXECUTE/GO ⤸
[EXECUTION]
RUNNING CBS08A
RUNNING CBS08B
RUNNING CBS08C

EXIT

.
```

EXAMPLE 2 Loading and Executing a MACRO Program

The user assembles the following MACRO program:

```
.COMPILE SIMPLE.MAC ⤸
MACRO: SIMPLE


EXIT

.
```

In the following command sequences, the user loads the MACRO program, interactively requests a listing of the relocation counters, library search symbols, and undefined global symbols, and then executes the program.

```
.R  LINK ⤸

*SIMPLE ⤸
*/COUNTER ⤸
RELOCATION COUNTER        INITIAL VALUE    CURRENT VALUE (OCTAL)
.LOW.                           0                140
.HIGH.                     400000             400025
*/ENTRY ⤸
NO LIBRARY SEARCH SYMBOLS (ENTRY POINTS)
*/UNDEFINE ⤸

NO UNDEFINED GLOBAL SYMBOLS
*/EXECUTE/GO ⤸
```

```
[EXECUTION]
THIS IS A VERY SIMPLE TWO-SEGMENT MACRO PROGRAM.

EXIT

.
```

EXAMPLE 3 Loading COBOL Programs and Creating a Saved File

In the following example, the user is individually loading each file and requesting a listing of undefined global symbols after each file is loaded. He also is requesting the searching of the default system libraries. After searching has been performed, the user creates a saved file and executes the core image.

```
.R LINK
*FILA/U

6 UNDEFINED GLOBAL SYMBOLS
BTRAC,    1212
TRACE,    1277
TRPD,     1214
TRPOP,    1213
CBS08B    1327
CBDDT,    1260
*FILB/U

6 UNDEFINED GLOBAL SYMBOLS
BTRAC,    1367
TRACE,    1473
TRPD,     1371
CBS08C    1615
TRPOP,    1370
CBDDT,    1454
*FILC/U

5 UNDEFINED GLOBAL SYMBOLS
BTRAC,    2052
TRACE,    2147
TRPD,     2054
TRPOP,    2053
CBDDT,    2130
*/SYSLIB/U

NO UNDEFINED GLOBAL SYMBOLS
```

LINK-10

Examples

```
*FILE/SAV/EXECUTE/GO
[EXECUTION]
RUNNING CBS08A
RUNNING CBS08B
RUNNING CBS08C

EXIT

.DIR *.SAV

FILE     SAV     5  <055>  23-APR-73     DSKC:   [27,235]

.
```

Example 4 Loading LINK-10

The Command File

```
/NOINITIAL /LOGLEVEL:1 DSK:LINK/MAP /CONT:NOABS =/RUNAME:LINK-
/HASHSIZE:1000/TEST:DDT/SYMSEG:H,LNKEXO,SCAN,HELPER-
,LNKINI,/NOSTART LNKSCN,LNKWLD,LNKFIO,LNKLOD,LNKOLD,LNKNEW,LNKF40-
,LNKCST,LNKCOR,LNKLOG,LNKERR,LNKMAP,LNKXIT,LNKSUB/SEARCH/GO
```

Running LINK-10 With the Terminal as the Log Device

```
        LINK-10 LOG file        23-Apr-73
8:58:11  1  1 LIM LINK-10 initialization
8:58:16  1  1 EXP Expanding low segment to 14P
8:58:16  1  1 MOV Moving low segment to expand area DY
8:58:16  1  1 LMN Loading module UDDT
8:58:16  1  1 MOV Moving low segment to expand area GS
8:58:17  1  1 EXP Expanding low segment to 18P
8:58:17  1  1 MOV Moving low segment to expand area LC
8:58:17  1  1 MOV Moving low segment to expand area LC
8:58:17  1  1 LMN Loading module LNKEXO
8:58:17  1  1 EXP Expanding low segment to 22P
8:58:17  1  1 MOV Moving low segment to expand area LC
8:58:18  1  1 LMN Loading module SCNDCL
8:58:18  1  1 LMN Loading module .SCAN
8:58:18  1  1 MOV Moving low segment to expand area HC
8:58:18  1  1 MOV Moving low segment to expand area HC
8:58:18  1  1 MOV Moving low segment to expand area HC
8:58:18  1  1 EXP Expanding low segment to 26P
8:58:18  1  1 MOV Moving low segment to expand area LS
8:58:18  1  1 MOV Moving low segment to expand area LS
8:58:18  1  1 LMN Loading module .TOUTS
```

LINK-10

Examples

```
8158118  1  1 MOV Moving low segment to expand area HC
8158118  1  1 LMN Loading module ,CNTDT
8158118  1  1 EXP Expanding low segment to 30P
8158118  1  1 MOV Moving low segment to expand area LS
8158118  1  1 LMN Loading module ,SAVE
8158118  1  1 LMN Loading module HELPER
8158119  1  1 LMN Loading module LINK
8158119  1  1 MOV Moving low segment to expand area HC
8158119  1  1 EXP Expanding low segment to 34P
8158119  1  1 MOV Moving low segment to expand area LS
8158119  1  1 LMN Loading module LNKSCN
8158119  1  1 MOV Moving low segment to expand area HC
8158119  1  1 LMN Loading module LNKWLD
8158119  1  1 MOV Moving low segment to expand area HC
8158120  1  1 EXP Expanding low segment to 38P
8158120  1  1 MOV Moving low segment to expand area HC
8158120  1  1 MOV Moving low segment to expand area HC
8158120  1  1 EXP Expanding low segment to 42P
8158120  1  1 MOV Moving low segment to expand area LS
8158121  1  1 LMN Loading module LNKFIO
8158121  1  1 MOV Moving low segment to expand area LS
8158121  1  1 LMN Loading module LNKLOD
8158121  1  1 MOV Moving low segment to expand area HC
8158121  1  1 EXP Expanding low segment to 46P
8158121  1  1 MOV Moving low segment to expand area HC
8158121  1  1 MOV Moving low segment to expand area LS
8158122  1  1 LMN Loading module LNKOLD
8158122  1  1 EXP Expanding low segment to 50P
```

•

•

•

```
8158125  1  1 PLS Area LS overflowing to DSK
8158128  1  1 MOV Moving low segment to expand area LS
8158128  1  1 LMN Loading module LNKMAP
8158128  1  1 LMN Loading module LNKXIT
8158129  1  1 LMN Loading module LNKPRM
8158129  1  1 LMN Loading module ,TSUBS
8158129  1  1 LMN Loading module ,INSUB
8158129  1  1 LMN Loading module JOBDAT
8158130  1  1 MPS MAP segment
8158130  1  1 MOV Moving low segment to expand area LS
8158132  1  1 EMS End of MAP segment
8158132  1  1 EXS EXIT segment
8158133  1  1 SST Sorting symbol table
8158134  1  1 EXP Expanding low segment to 67P
8158134  1  1 MOV Moving low segment to expand area HC
8158134  1  1 STC Symmbol table completed
8158136  1  1 FIN LINK-10 finished
       [END OF LOG FILE]
```

```
              LINK-10 symbol map of   LINK    version (33)          page 1
        Produced by LINK-10 version (33) on 23-Apr-73 at  8:58:30

Low  segment starts at       0 ends at    5247 length    5247 = 6P
High segment starts at  400000 ends at  445460 length   45460 = 38P
        Start address is 405045, located in program LINK

                ***********

UDDT    from SYS:DDT.REL[1,5]    created on 10-Mar-73 at  0:00:00
        Low  segment starts at     140 ends at    4555 length    4415 (octal), 2317 (decimal)

        DDT              140    Entry  point   Relocatable
        DDTEND          4555    Global symbol  Relocatable
        $1B             4426    Global symbol  Relocatable
        $2B             4431    Global symbol  Relocatable
        $3B             4434    Global symbol  Relocatable
        $4B             4437    Global symbol  Relocatable
        $5B             4442    Global symbol  Relocatable
        $6B             4445    Global symbol  Relocatable
        $7B             4450    Global symbol  Relocatable
        $8B             4453    Global symbol  Relocatable
        $I              4422    Global symbol  Relocatable
        $M              4425    Global symbol  Relocatable

                ***********

LNKEXO  from DSK:LNKEXO.REL[11,131]     created on 22-Apr-73 at  9:44:00
        High segment starts at  400010 ends at  400020 length      10 (octal), 8 (decimal)


                ***********

SCNDCL  from DSK:SCAN.REL[11,131]       created on  1-Nov-72 at  0:00:00

        zero length module

                ***********
```

```
.SCAN    from DSK:SCAN.REL[11,131]        Created on  1-Nov-72 at  0:00:00
         Low  segment starts at    4555 ends at    5245 length     470 (octal), 312 (decimal)
         High segment starts at  400020 ends at  404224 length    4204 (octal), 2180 (decimal)

         E.FMI          400731    Global symbol   Relocatable
         E.FMO          400720    Global symbol   Relocatable
         E.SVNG         402102    Global symbol   Relocatable
         E.SVTL         402074    Global symbol   Relocatable
         E.UKK          402037    Global symbol   Relocatable
         F.NAM            5117    Global symbol   Relocatable
         .ALDON         400700    Global symbol   Relocatable
         .CLRBF         403735    Global symbol   Relocatable
         .DATIC         402166    Global symbol   Relocatable
         .DATIF         402136    Global symbol   Relocatable
         .DATIG         402137    Global symbol   Relocatable
         .DATIM         402165    Global symbol   Relocatable
         .DATIP         402152    Global symbol   Relocatable

                             .
                             .
                             .

         Index to LINK-10 symbol map of  LINK    version (33)          page 15

         Name     Page     Name     Page     Name     Page     Name     Page

         HELPER     4      LNKF40    11      LNKOLD     9      UDDT       1
         JOBDAT    14      LNKFIO     7      LNKPRM    14      .CNTDT     3
         LINK       4      LNKCOD     8      LNKSCN     5      .INSUB    14
         LNKCOR    11      LNKLOG    13      LNKWLD     6      .SAVE      3
         LNKCST    11      LNKMAP    13      LNKXIT    13      .SCAN      1
         LNKERR    13      LNKNEW    10      SCNDCL     1      .TOUTS     3
         LNKEXO     1

         [End of LINK-10 map of  LINK]
```

LINK-10

Examples

Example 5 Loading the Monitor

The Command File

```
/NOINITIAL /LOGLEVEL:1 /HASH:7000 TOPS10/SAVE-
,TOPS10/MAP = /LOCALS /MAXCOR:200K COMMON,COMDEV,COMMOD-
,TOPS10/SEARCH /NOSYSLIBRARY /GO
```

The Log File

```
           LINK-10 LOG file         23-Apr-73
9:04:12  1  1 LIM LINK-10 initialization
9:04:19  1  1 EXP Expanding low segment to 14P
9:04:19  1  1 MOV Moving low segment to expand area DY
9:04:19  1  1 LMN Loading module COMMON
9:04:19  1  1 MOV Moving low segment to expand area LC
9:04:19  1  1 EXP Expanding low segment to 28P
9:04:19  1  1 EXP Expanding low segment to 32P
9:04:19  1  1 MOV Moving low segment to expand area LC
9:04:19  1  1 MOV Moving low segment to expand area LC
9:04:19  1  1 MOV Moving low segment to expand area LC
9:04:20  1  1 EXP Expanding low segment to 36P
9:04:20  1  1 MOV Moving low segment to expand area LS
9:04:20  1  1 EXP Expanding low segment to 40P
9:04:20  1  1 MOV Moving low segment to expand area LS
9:04:20  1  1 EXP Expanding low segment to 44P
9:04:20  1  1 MOV Moving low segment to expand area LS
9:04:21  1  1 EXP Expanding low segment to 48P
9:04:21  1  1 MOV Moving low segment to expand area LS
9:04:21  1  1 EXP Expanding low segment to 52P
9:04:21  1  1 MOV Moving low segment to expand area LS
9:04:22  1  1 MOV Moving low segment to expand area LS
9:04:22  1  1 LMN Loading module COMDEV
9:04:22  1  1 EXP Expanding low segment to 56P
9:04:22  1  1 MOV Moving low segment to expand area LC
9:04:22  1  1 MOV Moving low segment to expand area LC
9:04:22  1  1 MOV Moving low segment to expand area LC
9:04:23  1  1 EXP Expanding low segment to 60P
9:04:23  1  1 MOV Moving low segment to expand area LS
9:04:23  1  1 MOV Moving low segment to expand area LS
9:04:23  1  1 LMN Loading module COMMOD
9:04:23  1  1 EXP Expanding low segment to 61P
9:04:23  1  1 MOV Moving low segment to expand area LC
9:04:23  1  1 EXP Expanding low segment to 62P
```

LINK-10

Examples

```
9|04|23  1  1 MOV Moving low segment to expand area LC
9|04|23  1  1 EXP Expanding low segment to 63P
9|04|23  1  1 MOV Moving low segment to expand area LC
9|04|23  1  1 MOV Moving low segment to expand area LC
9|04|24  1  1 MOV Moving low segment to expand area LC
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 MOV Moving low segment to expand area GS
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 MOV Moving low segment to expand area GS
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 MOV Moving low segment to expand area LS
9|04|24  1  1 PLS Area LS overflowing to DSK
9|04|25  1  1 MOV Moving low segment to expand area GS
9|04|27  1  1 MOV Moving low segment to expand area LS
9|04|27  1  1 MOV Moving low segment to expand area LS
9|04|27  1  1 MOV Moving low segment to expand area LS
9|04|27  1  1 MOV Moving low segment to expand area GS
9|04|27  1  1 LMN Loading module EJBDAT
9|04|27  1  1 MOV Moving low segment to expand area LS
9|04|27  1  1 LMN Loading module FILFND
9|04|28  1  1 MOV Moving low segment to expand area LS
9|04|28  1  1 MOV Moving low segment to expand area LS
9|04|28  1  1 MOV Moving low segment to expand area GS
```

•
•
•

```
9|04|56  1  1 MOV Moving low segment to expand area LS
9|04|57  1  1 MOV Moving low segment to expand area LS
9|04|57  1  1 MOV Moving low segment to expand area LS
9|04|57  1  1 LMN Loading module ONCE
9|04|57  1  1 MOV Moving low segment to expand area LS
9|04|57  1  1 MOV Moving low segment to expand area LC
9|04|57  1  1 MOV Moving low segment to expand area LC
9|04|57  1  1 MOV Moving low segment to expand area LC
9|04|58  1  1 MOV Moving low segment to expand area GS
9|04|58  1  1 MOV Moving low segment to expand area LS
9|04|58  1  1 MOV Moving low segment to expand area LS
9|04|58  1  1 MOV Moving low segment to expand area LS
9|04|58  1  1 FCF Final core fixups
9|05|02  1  1 MPS MAP segment
9|05|02  1  1 MOV Moving low segment to expand area LS
9|05|18  1  1 EMS End of MAP segment
9|05|29  1  1 EXS EXIT segment
9|05|31  1  1 SST Sorting symbol table
9|05|35  1  1 STC Symmbol table completed
9|05|37  1  1 CSF Creating SAV file
9|05|51  1  1 FIN LINK-10 finished
      [END OF LOG FILE]
```

The Map File

```
             LINK-10 symbol map of   TOPS10  version (50534)          page 1
         Produced by LINK-10 version (33) on 23-Apr-73 at  9:05:02

Low   segment starts at        0 ends at  132776 length   132776 = 91P
         Start address is 106037, located in program SYSINI

             *************

COMMON   from DSK:COMMON.REL[11,131]     Created on 12-Dec-72 at 18:40:00
         Low  segment starts at      140 ends at    7357 length    7217 (octal), 3727 (decimal)

         A                    0     Global symbol    Non-Relocatable Suppressed
         A0.NOC           77000     Global symbol    Non-Relocatable Suppressed
         A00VER           50534     Global symbol    Non-Relocatable Suppressed
         A1050S               0     Global symbol    Non-Relocatable Suppressed
         ABSTAB             410     Global symbol    Non-Relocatable
         ADVEVM            5023     Global symbol    Non-Relocatable Suppressed
         AL                   1     Global symbol    Non-Relocatable Suppressed
         ALLJSP            5023     Global symbol    Non-Relocatable Suppressed
         ANYRUN            5021     Global symbol    Non-Relocatable Suppressed
         AP0BCK            6520     Global symbol    Non-Relocatable
         AP0CHL            5574     Global symbol    Non-Relocatable Suppressed
         AP0CHN               3     Global symbol    Non-Relocatable Suppressed
         AP0INT            6510     Global symbol    Non-Relocatable
         AP0JEN            5606     Global symbol    Non-Relocatable Suppressed
         AP0NUL          433553     Global symbol    Non-Relocatable Suppressed
         AP0PDP            5645     Global symbol    Non-Relocatable Suppressed
         AP0RET            5604     Global symbol    Non-Relocatable Suppressed
         AP0RST          633553     Global symbol    Non-Relocatable Suppressed
         AP0SAC            5625     Global symbol    Non-Relocatable Suppressed
         AP0SAV            5576     Global symbol    Non-Relocatable Suppressed
         APR0SN             240     Global symbol    Non-Relocatable Suppressed
         APR1SN               0     Global symbol    Non-Relocatable Suppressed
         APRSN              240     Global symbol    Non-Relocatable Suppressed
         APRSTS            4625     Global symbol    Non-Relocatable
         ASSCON          400000     Global symbol    Non-Relocatable Suppressed
         ASSPRG          200000     Global symbol    Non-Relocatable Suppressed
         AUAVAL            7140     Global symbol    Non-Relocatable
         AUQ                  4     Global symbol    Non-Relocatable
         AUUSER            7153     Global symbol    Non-Relocatable
         AVALTB            7140     Global symbol    Non-Relocatable
```

```
AVTBMQ          7133    Global symbol   Non-Relocatable Suppressed
BATMAX          4475    Global symbol   Non-Relocatable
BATMIN          4476    Global symbol   Non-Relocatable
BATNUM          4501    Global symbol   Non-Relocatable
BIGHOL          2411    Global symbol   Non-Relocatable
BISTDV          1355    Global symbol   Non-Relocatable
BLKSPK             3    Global symbol   Non-Relocatable Suppressed
BLKSPP             3    Global symbol   Non-Relocatable Suppressed
BNXMTS          4717    Global symbol   Non-Relocatable
BOOTWD            22    Global symbol   Non-Relocatable
BOOTXT          5004    Global symbol   Non-Relocatable
BTHN               1    Global symbol   Non-Relocatable Suppressed
BYTTAB          1300    Global symbol   Non-Relocatable
CØDPN              Ø    Global symbol   Non-Relocatable Suppressed
CØFHN              1    Global symbol   Non-Relocatable Suppressed
```

•
•
•

```
Index to LINK-10 symbol map of  TOPS10  version (50534)        page 84

Name    Page    Name    Page    Name    Page    Name    Page

BTHINT  55      DTASRN  63      ME+CON  66      REMDLX  68
CDPSER  55      EDDT    79      MTXSER  66      RTTRP   69
CDRSRX  55      EJBDAT  47      ONCE    82      SCHED1  69
CLOCK1  56      ERRCON  63      ONCMOD  80      SCNSER  70
COMCON  58      FHXKON  54      PATCH   78      SEGCON  74
COMDEV  27      FILFND  47      PLTSER  67      SWPSER  75
COMMOD  32      FILIO   50      PTPSER  67      SYSCHK  80
COMMON   1      FILUUO  52      PTRSER  67      SYSINI  78
CORE1   61      KALOCK  65      PTYSER  67      TMPUUO  75
DATMAN  62      KASER   64      REPSTR  81      UUOCON  75
DPXKON  54      LPTSER  65

[End of LINK-10 map of  TOPS10]
```

APPENDIX A

LINK Item Types

Input to LINK-10 is in the form of relocatable binary (.REL) files. Each .REL file is composed of link items of varying lengths. Each link item contains a specific type of information for LINK-10. The first word of these items is a header word containing, in the left half, an octal code for the item type and, in the right half, usually the number of words in the item. For item types 0-37, the count of words does not include overhead words (i.e., relocation words); for item types 1000-1777, the count does include these words. The format of the remaining words depends upon the individual link item. The link items are as follows:

Link Item Type	Use
0-37	Reserved for DEC
40-77	Reserved for customers
100-377	Reserved for DEC
400	FORTRAN-IV (F40) marker block
401	FORTRAN-IV (F40) with MANTIS information
402-777	Reserved for customers
1000-1777	Reserved for DEC (not used in the first release of LINK-10)
2000-3777	Reserved for customers
4000-777777	Reserved to avoid conflict with ASCII text

A.1 Link Item Types 0-37

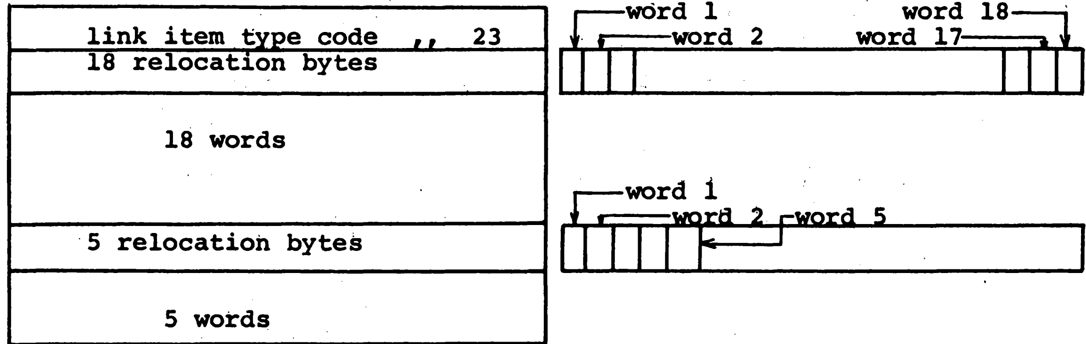
Link items in this range are the LOADER program block types and all have an identical format. The first word of the item, the header word, contains the item type in the left half and the count of data words in the right half. Following the header word is a relocation word containing up to 18 2-bit bytes which specify the relocation bits for the 18 words or less that follow. The relocation bits are left-justified and have the following meanings:

Byte Value	Meaning
0	Do not relocate
1	Relocate right half of word
2	Relocate left half of word
3	Relocate both halves of word

Following the relocation word are up to 18 words of the item. If there are more than 18 words in the item, there is another word of relocation bytes for the next 18 words. The relocation words are not included in the count of data words appearing in the left half of the header word. Thus, an item with a word count of 23 decimal would be as follows:

LINK-10

Item Types Ø-37



A.1.1 Link Item Type 0

This item type is ignored by LINK-10 and therefore can be used to store information not required by it. Totally null words look like this item type.

A.1.2 Link Item Type 1 CODE

This item type contains code and data. The first data word specifies the beginning address into which the item is to be loaded. The remaining words of the item are loaded into contiguous locations starting at that address. All words, including the load address, are relocated as specified by the relocation bytes.

If bit 0 of the first data word is 1, the word is assumed to be a Radix-50 symbol. The load address is then the value of this symbol plus the next word. Thus, in this case, there is one less actual data word than is indicated by the count in the header word.

A.1.3 Link Item Type 2 SYMBOLS

This item type consists of symbols, with each symbol occupying two words. The first word of each symbol contains 4 bits of code (bits 0-3) and 32 bits of the Radix-50 representation of the symbol (bits 4-35). The second word is the value of the symbol.

The code bits are as follows:

00 This symbol is a program name. It is entered into the symbol table by a link item type 6, not an item type 2. (This code should never happen.)

04 This symbol is a global definition. Its value is available to other programs. Two global symbols with the same name but different values cause an error message.

10 This symbol is a local symbol and is not loaded unless the user requests the loading of local symbols. Local symbols of the same name can occur in different modules without causng an error, even though the values may be different.

14 This symbol is a block name and is used by translators that are block structured. This symbol is not loaded unless the user requests loading of local symbols.

44 Same as code 04, with the addition that the global symbol is suppressed to DDT typeout.

50 Same as code 10, with the addition that the local symbol is suppressed to DDT typeout.

60 This symbol is a global request.

If bit 0 of the second word in the pair is 0, then bits 18-35 contain the address of the first word in a chain of requests for the global. In each request, the right half of the second word contains the address of the next request. The chain is terminated when the right half of the second word contains zero.

If bit 0 of the second word in the pair is 1, the request involves additive global processing. When bit 2 of this word is 0, bits 18-35 contain an address of a word of code. The right half of the value of the symbol requested is added to the left or right half of this word of code according to the following rule:
If bit 1 of the second word in the pair is 1, the add

is to the left half.
If bit 1 of the second word is 0, the add is to the right half.

The result is stored back into the word of code. (Note that there is no full word add; that result must be accomplished by a left and a right add.)

When bit 2 of the second word is 1, bits 3-35 contain the Radix-50 representation of a second symbol, whose value depends upon the global being requested. The second symbol must be the last symbol defined before the global request or else it will be treated as a local symbol and no action will occur, unless local symbols are being loaded. When the value of the requested global is determined, it is added to the right half of the value of the second symbol if bit 1 of the second word is 0, or to the left half if bit 1 is 1. Since the actual value of the symbol is not determined until the definition of the global upon which it depends, the code bits of the symbol indicate that the value of the symbol will change and cannot be used to satisfy requests until the symbol is fully defined.

A.1.4 Link Item Type 3 HISEG

This item type indicates to LINK-10 that code is to be loaded into the high segment. This item type has either one or two data words. The right half of the first data word is the initial address in the high segment (usually 400000). When the left half of the data word is zero, subsequent CODE items are assumed to have been produced by the HISEG pseudo-op in MACRO-10. This means that the addresses are relative to zero but are to be placed into the high segment. When the left half of the first data word is negative (i.e., greater than the right half) , subsequent CODE items have been generated by the TWOSEG pseudo-op in MACRO-10. This requires that addresses greater than the right half be placed into the high segment and addresses less than the right half be placed into the low segment. The left half is interpreted as the high segment break (i.e., the first free location

after the high segment) with the maximum length of the high segment being the difference between the left and right halves of the word. One-pass translators that cannot determine the high segment break should set the left half of the data word equal to the right half.

If there is a second data word (e.g., as in FORTRAN-10), the right half of this word is the low segment origin (usually 0) and the left half is the low segment program break.

A.1.5 Link Item Type 4 ENTRY

This item type is the entry item and must be the first item in a .REL file if the .REL file is to be loaded in a library search. It consists of a list of Radix-50 symbols which are separated every 18 words by a relocation word of zeroes. When LINK-10 is in library search mode, each global symbol in the list is checked against the undefined global requests for the load. If one or more matches occur, the following module is loaded. If a match does not occur, the module is ignored. If LINK-10 is not in library search mode, this checking of undefined global requests is not performed.

The entry items are stored. If the module is not loaded, these items are ignored. If the module is loaded, the entry items are scanned again and the entry point bit is turned on for the corresponding symbol in the symbol table.

A.1.6 LINK Item Type 5 END

This item type is the end item and is the last link item in the .REL

file. It contains two words whose meanings depend on whether the file contains two segments or one. If the file has two segments, the first word is the high segment break and the second word is the low segment break. If the file has only one segment, the first word is the first free location above the program (this word is relocatable) and the second word is the highest absolute address seen, if higher than location 137.

A.1.7 Link Item Type 6 NAME

This item is the name item and must appear before any type 2 link item (SYMBOL), The item has one or two data words. The first word is the program name in Radix-50 symbol format. The second word, if it appears, contains in bits 6-17 a code for the translator that produced the binary file, and in the right half the length of blank COMMON. (FORTRAN programs use both named and blank COMMON. COBOL uses blank COMMON to indicate the length of LIBOL's static area. Thus, the length has meaning for FORTRAN and COBOL programs.) The octal codes (bits 6-17) for the various translators are as follows:

Octal Code	Translator
0	UNKNOWN
1	F40
2	COBOL
3	ALGOL-60
4	NELIAC
5	PL/1
6	BLISS-10
7	SAIL
10	FORTRAN-10
11	MACRO
12	FAIL

Bits 0-5 of the second word indicate the processor on which the program will execute. If the value of these bits is 0, the program will execute on either processor; if the value is 1, the program will execute only on the KA10 processor; and if the value is 2, the program will execute only on the KI10 processor. Remaining values are reserved for the future.

A.1.8 Link Item Type 7 START ADDRESS

This item type contains in the right half of the data word the address at which execution of the program is to begin. The start address for a relocatable program may be relocated by means of the relocation bits. The last link item of this type encountered by LINK-10 is the one used, unless LINK-10 is ignoring start addresses (indicated by the user via switches). If the program is not to specify a start address, no item of this type should be included.

A.1.9 Link Item Type 10 INTERNAL REQUEST

This item type is provided for one-pass language translators when internal symbols are used before they are defined. The item type consists of a series of data words where each word represents one request. Each data word has a value in the right half and a pointer to the last request in the chain of requests for that value in the left half. Each quantity may be relocatable. The symbols are chained in a manner similar to the global requests which have bit 0 in the second word of each pair equal to zero (i.e., the value is substituted in the right half of each location in the chain). However, if a data

word is -1, then the next data word indicates a chained request to the left half of the word specified rather than the right half.

A.1.10 Link Item Type 11 POLISH

This item type is provided for Polish fixups involving arithmetic and logical operations on relocatable or externally-defined quantities. Each item contains only one Polish string. The data words in each item are a series of half-words consisting of operators and operands followed by store operators and store addresses. The operators and operands are as follows:

0	The next half word is an operand.
1	The next two half-words form a 36-bit operand.
2	The next two half-words form a Radix-50 symbol which is a global request. The operand is the value of the global.
3	Add.
4	Subtract.
5	Multiply.
6	Divide.
7	Logical AND.
10	Logical OR.
11	Left shift.
12	Logical XOR.
13	One's complement (not).
14	Two's complement (negative).

The store operators are as follows:

18 bit value

-1	Right half chained fixup (777777).
-2	Left half chained fixup (777776).
-3	Full word chained fixup (777775). The entire word pointed to is replaced and the old right half points to the next full word.

The half word following the store operator is used as the address of the first element in the chain.

LINK-10

Item Types Ø-37

A.1.11 Link Item Type 12 LINK

Data words in this item type occur in pairs. The first word of the pair is a link number and the second word is an address. There are 20 (octal) links numbered from 1 to 20. When LINK-10 is initialized, the value of each link is set to zero. Each time a specific link is seen, the current value of the link is stored in the address specified by the second word of the word pair, and the specified address becomes the new value of the link. If the number of the link seen is negative, the address is saved as the end of the link . At the end of loading, the current value for each link is stored in the address indicated by the end of each link. If the end of the link is 0, no storing is done.

A.1.12 Link Item Type 13 LVAR

This item type is used in LVAR fixups and is not currently handled by LINK-10. It is not supported by DEC and is not needed because the TWOSEG pseudo-op is superior. The first data word is the location of a variable area in the low segment. The second data word is the length of the area needed. The low segment relocation counter is incremented by the area needed. Data words after the first two data words occur in pairs. If bit 2 of the first word of the pair is zero, then the second word contains, in its left half, the address of a fixup chain, and in the right half, the relative location in the variable area to use for this fixup. The chaining occurs with the right half of the words if bit 0 of the first word is 0; otherwise, chaining occurs with the left half of the words.

If bit 2 of the first word of the pair is one, then the pair is used to make a symbol table fixup. The right half of the first word is the value of the fixup. The second word is the Radix-50 representation for the symbol.

A.1.13 Link Item Type 14 INDEX

This item type is produced by FUDGE2 to identify an index to LINK-10. The index is a list of all entry points (Link item type 4) in a library .REL file with pointers to the beginning of the individual modules. The index is 200 octal words long and if there are more entries in the library than will fit in 200 words, other item types 14 are created to contain the remainder of the entries. Each index is divided into sub-items of various lengths. The sub-items do not include the relocation word normally found in entry items of a library. Each sub-item has a header word with the word count in the right half and the link item type 4 in the left half. Following this header is the list of Radix-50 entry symbols. After the list of entries, there is a pointer to the individual module within the library file. The right half of the pointer is the block number of the module, and the left half is the word count within the block for the start of the module. The last word of the index item type contains a -1 in the left half to signal the end of the index item and the block number of the next index item in the right half. If LINK-10 is not in library search mode, index items are ignored.

A.1.14 Link Item Type 15 ALGOL

This item type is the special ALGOL OWN item. The first data word is the length of the OWN area to be allocated in the low segment. The remaining words are chained with the right half of the OWN fixups.

A.1.15 Link Item Type 16 REQUEST LOAD

This item type is produced by the SAIL compiler and is used to request the loading of programs. Thus, a .REL file can request libraries and other files to be loaded, thereby keeping the command string to LINK-10 simple. LINK-10 maintains a table for the names of libraries to be loaded and another table for the names of standard relocatable binary files to be loaded. When a new file is requested by link item type 16 or 17, LINK-10 searches the appropriate table to verify that the file has not already been specified. If it has not been specified, an entry is made in the appropriate table. After all files in the LINK-10 command string have been loaded, the files specified in the two tables are loaded. The relocatable binary files are loaded first; the libraries are loaded last.

The data words in this link item type appear in triplets. The first word contains the filename in SIXBIT (the extension of .REL is assumed). The second word is the UFD number in binary, and the third word is the SIXBIT name of the device containing the file.

A.1.16 Link Item Type 17 REQUEST LIBRARY

This item type is the same as item type 16 except that the specified

files are loaded only if they are needed to satisfy global requests. That is, the files are loaded in library search mode. The data words are identical to those in item type 16.

A.1.17 Link Item Type 20 COMMON ALLOCATION

This item type is used to allocate named COMMON areas. The relocation word must be present, but the bits should be zero. The data words are grouped in pairs, where the first word contains the Radix-50 symbol for the name of the COMMON area and the second word contains the length of the area required by this program.

This item type causes LINK-10 to search for the specified COMMON area to determine if it has been previously loaded. If it has, the length given in this item type must be less than or equal to the length already allocated. Thus, the first program that defines a COMMON area also defines the maximum size of that COMMON area. No subsequent program can expand this particular area, although COMMON areas of different names can be defined.

If the specified COMMON area has not been loaded, the symbol name is given the current low segment relocation value, and the length of the area is added to the low segment relocation counter.

A.1.18 Link Item Type 21 SPARSE DATA

This item type is used to load data into arrays when link item type 1 is inefficient for this purpose. The data words are grouped in sub-items and each sub-item is treated in the same manner as link item

type 1. The first word of each sub-item contains in the left half a count of the number of data words in the sub-item, and in the right half the beginning address into which the data words are to be loaded. The remaining words of each sub-item are the data words.

If bit 0 of the first word of a sub-item is 1, the first word is assumed to be a Radix-50 symbol. The left half of the second word is the count of data words and the right half contains an offset. The load address is then the value of the symbol plus the offset.

A.1.19 Link Item Types 22-36

These item types are not yet defined and return an error message if used.

A.1.20 Link Item Type 37 DEBUG

This item type is used for the debugging symbol table for COBDDT (the COBOL debugging program). If debugging is requested in local symbol mode, the data from this item type is loaded in the same manner as the data from link item type 1. If local symbols are not required, this item type is ignored.

A.2 Link Item Type 400 FORTRAN (F40)

This item type is output by the old one-pass FORTRAN-IV complier (F40). It does not contain a word count, relocation words, or data words. It contains only the one word indicating the item type code.

A.3 Link Item Type 401 FORTRAN (F40)

This item type is similar to link item type 400 and in addition it indicates that the file contains MANTIS debugging information.

A.4 Link Item Types 1000-1777

Link items in this range do not have identical formats. There is a general pattern in that the first word of each item contains an item type number in the left half and a word count in the right half. However, unlike link item types 0-37, the word count of item types 1000-1777 is a count of all following words including overhead words (relocation words). The structure of the relocation words depends upon the link item; there may be any number of relocation bits from 1 to 18 per half or full word. Link items that do not need relocation do not have relocation words. These item types are not used in the first release of LINK-10.

A.4.1 Link Item Type 1000

This item type is ignored by LINK-10 and thus can be used to store information not required by it.

A.4.2 Link Item Type 1001 ENTRY

This item type is the simple entry item and consists of a list of SIXBIT symbols. Each data word contains one left-adjusted symbol which can be a maximum of six characters in length. There are no relocation words, thus distinguishing this item type from item type 4. However, the two item types are used in the same manner.

A.4.3 Link Item Type 1002 LONG ENTRY

This item type contains one extended symbol (i.e., the symbol contains more than six characters) in SIXBIT, which is tested to determine if it is required as an entry point. This link item type is used in the same manner as link item type 1001.

A.4.4 Link Item Type 1003 NAME

This item type contains information about the file and the translator that produced it. The information in this item is stored in the symbol table and can be output on a map listing.

The data words occur in triplets. The left half of the first word of each triplet contains flag bits for that triplet and the right half is unused. The first triplet of data (the primary triplet) contains the program name in SIXBIT in the second word. This program name is taken from the TITLE statement in a MACRO-10 program. If the program name is longer than six characters, one or more triplets follow containing the remaining characters of the name. Triplets following the program name are identified by the flag bits in the first word of each triplet. The triplet after the name triplets contains the low segment relocation counter in the second word and the high segment relocation counter in the third word. The next triplet has, in the second word, the SIXBIT name of the translator that produced the file and in the third word, the version number of the translator. This version number is taken from location 137. The following triplet contains the compilation date and time obtained from the LOOKUP UUO block in the

second word, and in the third word, a default code for the translator used, in case LINK-10 could not determine the translator from the information in the previous triplet. The default translator codes are listed in Paragraph A.1.7. The next triplet contains in the second word, the name of the device on which the source file is stored, and in the third word, the SIXBIT filename of the source file. The information in the next triplet is the source filename extension in the second word and the name of the UFD containing the source file in the third word. The next triplet contains sub-file directory information. The following triplet contains the version number of the source file as obtained by the translator that processed the file. The information in the last triplet is interpreted as ASCII text and is stored in the format in which it is given.

More than one NAME link item may be seen per module for programs made from several source files. The program and compiler name triplets must be the same in the the NAME link items, but the source filename and any remaining triplets can be different.

A.4.5 Link Item Type 1004 RELOCATION

This item type consists of groups of words (usually pairs) without any relocation words. The first data word of the item type contains the total number of relocation groups in the item in order that sufficient space can be allocated. The first word of each relocation group has a relocation level in the left half and the count of the number of words in the relocation counter name in the right half. The remaining words in each group are the relocation counters. The relocation level is

the position in the table of relocation counters, such that for any word needing relocation, the value of the relocation byte will receive the correct constant for addition.

If a relocation counter is not yet defined (or a complex Polish expression not yet resolved), it must be placed in the undefined table, and its slot in the relocation tables is marked as undefined. All code referring to the undefined counter is stored in the fixup area or on the disk. In other words, if the location into which code is to be loaded is not yet defined, all the code under the relocation counter must be placed in the fixup table or on the disk. Link item type 1004 can appear anywhere and must be used whenever a new relocation counter is used. The standard name for the low segment relocation counter is .LOW. and the standard for the high segment counter is .HIGH.. These counters normally occupy positions 1 and 2 in the table of relocation counters.

A.4.6 Link Item Type 1005

This item type is undefined and reserved for future definition.

A.4.7 Link Item Type 1006 START

This item type contains the start addresses for the program. It consists of a relocation word with 4-bit bytes for full word relocation, followed by the list of relocatable start addresses in order of their use. These addresses are used or ignored depending on the switches given by the user. Currently, only one start address per program is recognized.

A.4.8 Link Item Type 1007 START

This item type is used for additional start addresses or external symbolic start addresses. The link item is divided into groups of words for each start address. The first word of each group contains flag bits in the left half and the count of the number of words in the group in the right half. Currently, bit 0 is the only flag bit. If this bit is 1, a Polish expression follows; if it is 0, a symbol follows. This item type does not include relocation words.

A.4.9 Link Item Types 1010-1017 CODE

The link items in the range 1010-1017 are similar except for the size of the relocation byte. The most general case uses 18 bits per half word, but this method consumes too much space for simple programs. Item type 1010 has a byte size of 2 bits, thereby allowing three relocation counters and absolute code. Relocation occurs only on the right half of the word and is positive; the left half is considered absolute. Since in most programs the code consists of constants in the left half (op-codes, indexes, ACs) and relocatable addresses or constants in the right half, this item type should be sufficient for most programs.

Item type 1011 also has 2-bit bytes but has relocation for the left half as well as the right half of the word. This item type allows three relocation counters plus absolute code. Link item type 1011 is used mainly for table generation.

Item type 1012 allows relocation only for the right half of the word (similar to item type 1010) but has a byte size of 4 bits, giving allowances for 15 relocation counters.

Item type 1013 allows relocation for both the left and right halves of the word (similar to item type 1011) but uses a 4-bit byte size.

Item types 1014-1016 are reserved for future use.

Item type 1017 has 18 bits of relocation per half word.

A.4.10 Link Item Types 1020-1027 SYMBOL

All symbols are in triplet format. The link items in the range 1020-1027 differ only in the size of the relocation b-te. This byte is the same as the byte size for the corresponding CODE item. For example, symbol type 1020 and code type 1010 use 2-bit bytes, symbol type 1022 and code type 1012 use 4-bit bytes, and so forth. The relocation word applies only to the third word of the triplet (the symbol value). Thus, for example, in the case of symbol type 1020, each relocation word is followed by up to 18 triplets rather than 18 words.

A.4.11 Link Item Type 1030 POLISH

This item type is provided for Polish fixups and consists of operators and operands, including store operators and store operands in pre-fixup form. Each item contains only one Polish string, but may contain many different store pointers. Operators are stored one per half word, and symbols are stored in contiguous half words. Store

pointers are in the form of either an address in a halfword or a byte pointer in a full word. Associated with store pointers are store operators that shift the value to the correct field and store operator. The store operator may also point to a symbol that is to be stored in the symbol table.

The operators and operands are as follows:

0	The next half word is an operand.
1	The next two half words form a 36-bit operand.
2	The next two half words form a 36-bit symbol which is a global request. The operand is the value of the global.
3	The next half word is the count of half words in an extended symbol. The subsequent half words are the symbol.
4	The next half word is a numeric relocation counter for the program.
5	The next two half words are a symbolic relocation counter.
6	The next half word is a count of the number of half words in an extended symbolic relocation counter. The following halfwords are the relocation counter.
7	The next two half words are a byte pointer to code already loaded.
10-77	Reserved for future use.
100	Add
101	Subtract
102	Multiply
103	Divide
104	Logical AND
105	Logical OR
106	Left Shift (LSH)
107	Logical XOR
110	One's complement (not)
111	Two's complement (negate)
112	Get contents (MOVE)
113	Reserved for future use

The store operators are as follows:

18 Bit Value

-1	Right half chained fixup (777777).
-2	Left half chained fixup (777776).
-3	Full word chained fixup (777775).
-4	The next two half words are a byte pointer

	(777774).
-5	The next two half words are an instruction plus an address (ANDM,XORM) (777773).
-6	The next two half words are a symbol and the value is stored in the half words (777772).
-7	The next half word is the count of the number of half words in an extended symbol. The half words following are the extended symbol and the value is stored in these half words (777771).
-10	The next half word is a numeric relocation counter (777770).
-11	The next two half words are a symbolic relocation counter (777767).
-12	The next half word is a count of the number of half words in an extended symbolic relocation counter. The following half words are the counter (777766).
-13	Reserved for future use.

The store operators obtain their arguments from a stack; the first word is usually the value and the second is the memory address. Addresses can be built using other Polish operators. For chained fixups, the half word preceding the store operator is used as the address of the first element in the chain.

A.4.12 Link Item Type 1031 POLISH

This item type is similar to item type 1030 except that Polish notation in post-fixup form is used. The operators and operands are the same.

A.4.13 Link Item Types 1032-1033

These item types are reserved for future use.

A.4.14 Link Item Types 1034-1037 CONDITIONAL

There are three kinds of conditonal loading item types: the Begin

conditional, the End conditional, and the Else conditional. The Begin conditional has a unique number assigned by the translator which is matched with the unique number in the End and Else conditionals. It also contains a conditional operand and operator. The End conditional cancels the conditional loading, updates the relocation counters, and generates the next implicit relocation counter, if it is not explicitly defined by the user, so that following code can be loaded. The Else conditional is the inverse of the condition in the Begin conditional in that code is loaded if the condition is false. The three kinds of condition items can be nested.

A.4.14.1 The Begin Conditional - Link Item Type 1034 - This item type has four relocation bits per half word thereby allowing 15 possible relocation counters. The first data word contains the unique conditional number. If a number is not specified, zero is assumed and LINK-10 matches the Begin with the first End or Else conditional at that level. The second data word contains the conditional operator in the left half and the conditional operand in the right half. The remaining words contain the rest of the operand.

The conditional operators are coded as follows:

0	null
1	if zero
2	if greater than zero
3	if greater than or equal to zero
4	if less than zero
5	if less than or equal to zero
6	if not equal to zero
7	if defined
10	if not defined
11	if global
12	if local

LINK-10

Item Types 1000-1777

The operand is either a symbol or a Polish expression. If the operand cannot be evaluated, the words are stored on the disk. The operands are:

100	The next two half words contain a SIXBIT symbol.
101	The next half word is a count of the number of half words in an extended symbol. The following words contain the SIXBIT symbol.
102	A pre-fixup Polish expression follows (refer to Paragraph A.4.11).
103	A post-fixup Polish expression follows (refer to Paragraph A.4.12).

If the condition is met, all code up to an End or Else conditional is loaded. When the condition is not met, the code is not loaded.

A.4.14.2 The Begin Conditional - Link Item Type 1035 - This item type is similar to Link Item Type 1034 except that it has half word relocation per half word.

A.4.14.3 The Else Conditional - Link Item Type 1036 - This item type contains no relocation words and has one data word containing a unique number matching the one in the Begin conditional. If the condition in the Begin conditional is true, the code in the current Else conditional to its matching End conditional or to the next matching Else conditional is ignored. If the condition is not true, the code is loaded.

A.4.14.4 The End Conditional - Link Item Type 1037 - This item type also has no relocation words. The first data word is a unique number matching the one in the Begin conditional. If the condition in the Begin conditional is false and no Else conditional is seen, the End

conditional is ignored. However, if code was loaded, the End conditional is read. The item type contains one data word for each relocation counter used in the same order as specified in the last relocation setting link item. The data words are the highest value of the relocation counter used in the conditionally-loaded code. These values are added to the current values, and to the accumulation of such values, until the final END item type of the REL file.

A.4.15 Link Item Type 1040 END

This link item marks the end of a link module. It does not contain relocation words but does contain a list of all relocation counters used and their final values. Any conditional code that was loaded plus other overhead items, such as the ALGOL item, are added to the final values. The resulting values are then added to the current values of the relocation counters to obtain the value for the next module. The beginning and ending addresses are stored in the symbol table in order that DDT has the range of the program and that they can be output in a map listing.

A.4.16 Link Item Type 1041 Special FORTRAN-10 Block

This link item defines a call to a special once-only routine that is to be executed by LINK-10 after all code has been loaded.

A.4.17 Link Item Type 1042 Program Request

This link item requests the loading of .REL files required for this program. It is similar to link item type 16; however, there are no

relocation words. This item replaces the need for library searches and is useful when loading real and dummy routines because it specifies filenames rather than modules names.

The data appears in groups of four or more words. Each group contains the following words:

Name of the device in SIXBIT containing the file.

Name of the file in SIXBIT.

Extension of the file in SIXBIT in the left half, and the length of the directory in the right half.

UFD in octal.

Remaining words in the group are sub-file directory names in SIXBIT.

The requests are stored until the end of loading and are loaded before the default libraries and requested libraries (link item type 1043). Any number of files can be requested.

A.4.18 Link Item Type 1043 Library Request

This item type requests the searching of libraries, either in search mode for all unresolved entries or for particular modules. The data is identical to that in item type 1042.

A.4.19 Link Item Types 1044-1047

These item types are reserved for future use.

LINK-10

Item Types 1000-1777

A.4.20 Link Item Type 1050 Global Data

This item type contains data that is common to many programs (i.e., constants, argument lists, literals in MARCO-10 language). The global data item consists of two other link items: the relocation setting item (type 1004) and a code item (types 1010-1017). The initial global data item has no relocation words. The first data word is the header of the relocation item and only the relocation actually used should appear in this word; all other entires should be zero. The next data words are the data for the relocation item. Following these data words is a code item with relocation bits and data which may be relocatable or absolute. LINK-10 collects all the global data blocks, compares them, and keeps only one copy of those with the same data and relocation. The global data items are loaded at the end of loading or immediately after a /DATA switch is seen. These items should reduce the size of loaded programs because of pooling of literals.

A.4.21 Link Item Types Greater Than 3777 ASCII

These items are recognized by the first seven bits being non-zero (i.e., an ASCII character). There is no word count in the item. Termination of the item occurs at a null byte. These items are generated by translators and contain ASCII commands similar to those typed on the user's terminal. Thus they are similar to an indirect file. ASCII items allow the overlay structure to be embedded in the file to simplify the maintaining of large overlay programs.

APPENDIX B

LOADER AND LINK-10 DIFFERENCES

This appendix is intended as an aid for users who have been employing the LOADER program and who are now converting to the LINK-10 program. Both programs are linking loaders. Both have the same basic functions of loading and relocating user's object code modules and resolving references among the modules. But LINK-10 is not just an updated version of LOADER. It is a completely new, more sophisticated, and more flexible piece of software. This appendix itemizes the differences between the two programs in order to facilitate conversion to LINK-10.

LOADER	LINK-10
The default output device is TTY.	The default output device is DSK.
The default name of the MAP file is MAP.MAP.	The default name of the MAP file is the name of the last program with a start address. If there is no program with a start address, the default name is nnnLNK.MAP, where nnn is the user's job number.
Command files are specified in the form * file @ The default extension of the command file is .TMP.	Command files are specified in the form * @ file The default extension of the command file is .CCL.
Input and output specifications are separated by a back-arrow (←). Thus, an output file is defined as being on the	Input and output specifications may be separated by an equals sign (=), but this is not required. An output file is

LINK-10

LOADER and LINK-10 Differences

left side of the back-arrow.

specified by giving a file specification followed by an output switch.

The only output file produced by LOADER is a map file.

LINK-10 can be instructed to produce map, save, log, symbol, and XPN files.

Exit conditions are /G, altmode, and ↑Z.

The only exit condition is /GO.

Line terminators (e.g. <carriage return, line feed>) are treated in the same way as commas (i.e., they terminate the specification). File dependent switches remain in effect until overridden by a subsequent switch or until the end of the load. The most recently specified source device remains the default until a new device is specified or until the end of the load. Defaults carry across lines.

LINK-10 has a line oriented scanner. All file-dependent switches are turned off at the end of the line to which they belong. The most recently specified source device remains the default until a new device is specified or until the end of a line is reached. Standard defaults are restored at the beginning of each line. In general, it is best to place all the commands for loading a program on a single line. A hyphen is used as the line continuation character.

To load local symbols for FILE1 and FILE2 and then load DDT, the following sequence could be used:

```
*/S
*FILE1,FILE2
*/W/D$
```

To load local symbols for FILE1 and FILE2 and then to load DDT, the following sequence is used:

```
*/LOCALS FILE1,FILE2,
*/TEST /GO
```

Note that if the /LOCALS switch had appeared on a line by itself, it would have had no effect.

To search FILEA and FILEB in library search mode, the sequence:

```
*/L
* FILEA,FILEB
```

To search FILEA and FILEB in library search mode, the sequence is:

```
*/SEARCH FILEA,FILEB
```

The sequence

LINK-10

LOADER and LINK-10 Differences

LOADER	LINK-10
could be used.	*/SEARCH *FILEA,FILEB does not cause FILEA and FILEB to be searched. Instead, they are loaded in their entirety.
When performing a search of the default libraries at the end of the load, LOADER makes one pass through all required libraries. In addition, LIB40 is always searched.	LINK-10 performs multiple passes through all required libraries until no undefined symbols remain or until no additional routines have been loaded. In addition, LIB40 is not automatically searched unless it is required by an F40 program. Thus, when loading MACRO programs which utilize routines in LIB40, the user must explicitly request that LIB40 be searched. Also, JOBDAT.REL is not searched unless the /NOINITIAL switch is used. LINK-10 automatically initializes its global symbol table to include JOBDAT symbols.
The /D and /T switches load with local symbols. This mode remains in effect until it is turned off with the /W switch, and remains off until another switch which loads local symbols is given.	The /TEST and /DEBUG switches instruct LINK-10 to load all subsequent files with their local symbols. The /NOLOCAL switch can be used to suppress the loading of local symbols. However, since the /NOLOCAL switch is file dependent, it is cleared at the end of the line and load with local symbols mode is reinstated.

The following table lists each LOADER switch and the LINK-10 switch which performs the nearest equivalent action. Note that there is not always a one-to-one correspondence between the action performed by the LOADER switch and by the LINK-10 switch. Refer to Chapter 4 of the LINK-10 Programmer's Reference Manual for the complete descriptions of the LINK-10 switches.

LINK-10

LOADER and LINK-10 Differences

LOADER	LINK-10
/A	/CONTENT:ZERO
/B	/SYMSEG:LOW
/1B	/SYMSEG:HIGH
/nnnnB	/PATCHSIZE:nnnn
/C	No equivalent switch. LINK-10 does not support the old CHAIN facility.
/D	/TEST:DDT or /TEST:MACRO
/E	/EXECUTE
/F	/SYSLIB
/1F	/FORSE
/2F	/FOROTS
/G	/GO
/nnnG	/START:nnn
/H	/SEGMENT:LOW
/1H	/SEGMENT:HIGH
/nnnnH	/SET:.HIGH.:nnnn
/-H	/SEGMENT:DEFAULT
/I	/NOSTART
/J	/START
/nK	/RUNCOR:n
/-K	No equivalent switch. Use /RUNCOR.
/L	/SEARCH
/M	/MAP:END
/1M	/MAP:END/CONTENT:LOCALS

LINK-10

LOADER and LINK-10 Differences

/N	/NOSEARCH
/nnnO	/SET:.LOW.:nnn
/P	/NOSYSLIB
/Q	/SYSLIB at the end of the command string.
/R	No equivalent switch. LINK-10 does not support the old CHAIN facility.
/S	/LOCALS
/T	/DEBUG:DDT or /DEBUG:MACRO
/U	/UNDEFINED
/V	/OTS:HIGH
/-V	/OTS:LOW
/W	/NOLOCALS
/X	/CONTENT:NOZERO
/Y	/REWIND
/Z	/RUN:LINK

GLOSSARY

Absolute Address

A fixed location in user virtual address space which cannot be relocated. For example, the high-speed accumulators on the DECsystem-10 occupy locations 0 through 17 (octal) in the user's virtual address space. All modules that reference the accumulators must reference these locations. Thus the addresses 0 through 17 (octal) are absolute addresses.

Absolute Module

A module whose location counters are set to absolute addresses only.

Address Binding

The assignment of virtual address space to the physical address space in computer memory. This is automatically performed by the DECsystem-10 monitor and is completely invisible to user programs.

Assemble

To prepare a machine-language module from a symbolic-language module by substituting the actual numeric operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

LINK-10

Glossary

Assembler

A program which accepts symbolic assembly code and translates it into machine instructions. MACRO-10 is the standard assembler supplied by DEC.

Base Address

An address used as a basis for computing the value of some other address. This computation is usually of the form

final address = base address (+ or -) offset.

Clear

To erase the contents of a location by replacing the contents with blanks or zeroes.

COMMON Area

A section in a program's address space which is set aside for common use by many modules. COMMON is usually set up by modules written in the FORTRAN language. It is used by independently-compiled modules to share the same data locations.

Control Section

A unit of code (instructions and/or data) that is considered an entity and that can be relocated separately at load time without destroying the logic of the program. Control is passed properly from one Control Section to another regardless of their relative positions in user virtual address space. A Control Section is

identified by a relocation counter and thus is the smallest unit of code that can be relocated separately.

Default Directory

The directory in which the Monitor searches if a directory specification has not been given by the user. Typically, this is the UFD corresponding to the user's logged-in project-programmer number but it may another UFD or a SFD (sub-file directory).

Directory

A file which contains the names and pointers to other files on the device. The MFD, UFDs, and SFDs are directory files. The MFD is the directory containing all the UFDs. The UFD is the directory containing the files existing in a given project-programmer number area. The SFD is a directory pointed to by a UFD or a higher-level SFD. The SFDs exist as files under the UFD.

External Symbol

A global symbol which is referenced in one module but defined in another module. The EXTERN statement in MACRO-10 is used to declare a symbol external. A subroutine name referenced in a CALL statement in a FORTRAN module is automatically declared external.

LINK-10

Glossary

File

An ordered collection of 36-bit words comprising computer instructions and/or data. A file is stored on a device, such as disk or magnetic tape, and can be of any length, limited only by the available space on the device and the user's maximum space allotment on that device.

File Specification

A list of identifiers which uniquely specify a particular file. A complete file specification consists of: the name of the device on which the file is stored, the name of the file including its extension, and the name of the directory in which the file is contained.

FUDGE2

A system utility program used to update libraries containing one or more modules and to manipulate modules within these libraries.

GET

To transfer a saved program from a file on a device into core memory using a bootstrap program or the Monitor. The GET command places a program into memory. The RUN command performs the same operation and, in addition, starts the program. The GET operation differs from the LOAD operation (refer to LOAD).

LINK-10

Glossary

GLOB

A system utility program used to read libraries and to generate an alphabetical cross-referenced list of all the global symbols encountered. When a program is composed of many modules which communicate via global symbols, it is useful to have an alphabetical list of all global symbols with the names of the modules in which they are defined and referenced.

Global Request

A request to LINK-10 to link a global symbol to a module.

Global Symbol

A symbol that is accessible to modules other than the one in which it is defined. The value of a global symbol is placed in LINK-10's global symbol table when the module containing the symbol definition is loaded.

High Segment

That portion of the user's addressing space, usually beginning at relative location 400000, which generally is used to contain pure code that can be shared by other users. This segment is usually write-protected in order to preserve the data. The user can place information into a high segment with the TWOSEG pseudo-op in MACRO-10. Higher-level language, such as COBOL and FORTRN, also have provisions for loading pure code in the high segment.

LINK-10

Glossary

Initialize

To set counters, switches, or addresses to zero or other starting values at prescribed points in the execution of a computer routine.

Internal Symbol

A global symbol located in the module in which it is defined. In a MACRO-10 program, a symbol is declared internal with the INTERN or ENTRY pseudo-op. These pseudo-ops generate a global definition which is used to satisfy all global requests for the symbol. In FORTRAN programs, internal symbols are generated to match the names of SUBROUTINEs, FUNTIONs, and ENTRYs. An internal symbol is similar to a library search symbol; however, it will not cause a module to be linked in search mode.

Job Data Area (JOBDAT)

The first 140 octal locations of a user's address space. This area provides storage for certain data items used by both the Monitor and the user's program. Refer to the DECsystem-10 Monitor Calls Manual.

Label

A symbolic name used to identify a location in a program.

LINK-10

Glossary

Library

A relocatable binary file containing one or more modules which may be loaded in Library Search Mode. FUDGE2 is a system utility program which enables users to merge and edit a collection of relocatable binary modules into a library file. PIP can also be used to merge relocatable binary modules into a library, but it has no facilities for editing libraries.

Library Search Mode

The mode in which a module (one of many in a library) is loaded only if one or more of its declared entry points satisfy an unresolved global request.

Library Search Symbol (Entry Symbol)

A list of symbols that are matched against unresolved symbols in order to load the appropriate modules. This list is used only in library search mode. A library search symbol is defined by an ENTRY statement in MACRO-10 and BLISS-10 and a SUBROUTINE, FUNCTION, or ENTRY statement in FORTRAN.

Linker

A program that combines many input modules into a single module for loading purposes. Thus, it allows for independent compilations of modules. Typically, it satisfies global references and may combine control sections.

LINK-10

Glossary

Link

To combine independently-translated modules into one module in which all relocation of addresses has been performed relative to that module and all external references to symbols have been resolved based on the definition of internal symbols.

Linking Loader

A program that provides automatic loading, relocation, and linking of compiler and assembler generated object modules.

Load

To produce a core image and/or a saved file from one or more relocatable binary files (REL files) by transforming relocatable addresses to absolute addresses. This operation is not to be confused with the GET operation, which initializes a core image from a saved file (refer to GET).

Local Symbol

A symbol known only to the module in which it is defined. Because it is not accessible to other modules, the same symbol name with different values can appear in more than one module. These modules can be loaded and executed together without conflict. Local symbols are primarily used when debugging modules; symbol conflicts between different modules are resolved by mechanisms in the debugging program.

Low Segment

The segment of user virtual address space beginning at zero. The length of the low segment is stored in location .JBREL of the Job Data Area. When writing two-segment programs, it is advisable to place data locations and impure code in the low segment.

Main Program

The module containing the address at which object program execution normally begins.

Module

The smallest entity that can be loaded by LINK-10. It is composed of a collection of control sections. In MACRO-10, the code between the TITLE and END statements represents a module. In FORTRAN, the code between the first statement and the END statement is a module. In COBOL, the code between the IDENTIFICATION DIVISION statement and the last statement is a module.

Module Origin

The first location in user virtual address space of the module.

Object Module

The primary output of an assembler or compiler, which can be linked with other object modules and loaded into a runnable program. This output is composed of the relocatable machine

language code for the translated module (i.e., link items), relocation information, and the corresponding symbol table listing the definition and use of symbols within the module.

Object Time System

The collection of modules that supports the compiled code for a particular language. This collection usually includes I/O and trap-handling routines.

Offset

The number of locations relative to zero that a Control Section must be moved before it can be executed.

Operating System

The collection of modules that automatically permits continuous job processing by scheduling and controlling the operation of user and system programs, performing I/O, and allocating resources for efficient use of the hardware.

Physical Address Space

A set of memory locations where information can actually be stored (i.e., core memory) for the purpose of program execution.

Program

A collection of routines which have been linked and loaded to produce a saved file or a core image. These routines typically

consist of a main program and a set of subroutine which may have come from a library.

Pure Code

Code which is never modified in the process of execution. Therefore, it is possible to let many users share the same copy of a program.

REL File

One or more relocatable object modules composed of link items (refer to Appendix A).

Relocatable Address

An address within a module which is specified as an offset from the first location in that module.

Relocatable Control Section

A control section whos addresses have been specified relative to zero. Thus, the control section can be placed into any area of core memory for execution.

Relocation Counter

The number assigned by LINK-10 as the beginning address of a Control Section. This number is assigned in the process of

loading specific Control Sections into a saved file or a core image and is transformed from a relocatable quantity to an absolute quantity.

Relocation Factor

The contents of the relocation counter for a control section. This number is added to every relocatable reference within the Control Section. The relocation factor is determined from the relocatable base address for the control section (usually 0 and 400000) and the actual address in user virtual address space at which the module is being loaded.

Routine

A set of instructions and data for performing one or more specific functions.

Segment

An absolute Control Section.

Source Language Program

The original, untranslated version of a program written in a higher-level language (e.g., FORTAN, COBOL, MACRO). Source programs, when translated, produce object modules as their primary output. A program may exist as a source program, an object module, and a runnable core image.

LINK-10

Glossary

Symbol

Any identifier (composed of SIXBIT characters) used to represent a value that may or may not be known at the time of its original use in a source language program. Symbols can appear in source language statements as labels, addresses, operators, and operands.

Symbol Binding

To resolve references in one module to symbols which are defined (i.e., are assigned a value) in another module.

Symbol Table

A table containing entrie for each symbol defined or used within a module.

Translate

To compile or assemble a source program into a machine language program, usually in the form of a (relocatable) object module.

User Virtual Address Space

A set of memory addresses within the range of 0 to 256K words. These addresses are mapped into physical core addresses by the paging or relocation-protection hardware when a program is executed. On a KA10 processor, the range of addresses is limited by the amount of physical core available to a given user.

LINK-10

Glossary

User's Program

All of the code running uder control of the Monitor in a user virtual address space of its own.

Zero Length Module

A module containing symbol definitions but no instruction or data words (e.g., JOBDAT). Note that the word "length" in this context refers to the program length of the module after loading.

DEC-10-CDDE-D

DDT-10
PROGRAMMER'S REFERENCE MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Printing January 1968
2nd Printing (Rev) April 1969
3rd Printing (Rev) June 1969
4th Printing (Rev) November 1969
5th Printing (Rev) August 1970

Copyright © 1968, 1969, 1970 by Digital Equipment Corporation

The material in this manual is for information purposes and is subject to change without notice.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

DEC	PDP
FLIP CHIP	FOCAL
DIGITAL	COMPUTER LAB

CONTENTS

Page

CHAPTER 1
INTRODUCTION

1.1	Loading Procedure	869
1.2	Learning to Use DDT	869

CHAPTER 2
BASIC DDT COMMANDS

2.1	Examining Storage Words	871
2.2	Type-Out Modes	871
2.3	Modifying Storage Words	872
2.4	Type-In Modes	873
2.5	Symbols	873
2.6	Expressions	874
2.7	Breakpoints	874
2.7.1	Setting Breakpoints	875
2.7.2	Breakpoint Restrictions	875
2.7.3	Breakpoint Type-Outs	875
2.7.4	Removing and Reassigning Breakpoints	876
2.7.5	Proceeding From a Breakpoint	876
2.8	Starting the Program	876
2.9	Deleting Typing Errors	876
2.10	Error Messages	877
2.11	Summary	877

CHAPTER 3
DDT COMMANDS

3.1	Examining the Contents of a Program Storage Word	879
3.2	Changing the Contents of a Word	880
3.3	Inserting a Change, and Examining the Contents of the Last Typed Address	881
3.4	Starting the Program	883
3.5	One-Time Typeouts	883
3.5.1	Type-Out Numeric	883
3.5.2	Type-Out Symbolic	883
3.5.3	Type-Out in Current Mode	883

CONTENTS (Cont)

		Page
3.6	Symbols	883
3.7	Typing In	884
3.7.1	Typing In Symbolic Instructions	885
3.7.2	Typing In Numbers	885
3.7.3	Typing In Text Characters	885
3.7.4	Arithmetic Expressions	886
3.8	Delete	886
3.9	Error Messages	886
3.10	Upper and Lower Case (Teletype Model 37)	887

CHAPTER 4
MORE DDT-10 COMMANDS

4.1	Changing the Output Radix	889
4.2	Type-Out Modes	889
4.2.1	Primary Type-Out Modes	890
4.3	Breakpoints	891
4.3.1	Setting Breakpoints	891
4.3.2	Removing Breakpoints	891
4.3.3	Restrictions for Breakpoints	892
4.3.4	Restarting After a Breakpoint Stop	892
4.3.5	Automatic Restarts from Breakpoints	892
4.3.6	Checking Breakpoint Status	893
4.3.7	Conditional Breakpoints	893
4.3.7.1	Using the Proceed Counter	894
4.3.7.2	Using the Conditional Break Instruction	894
4.3.8	Entering DDT from a Breakpoint	895
4.4	Searches	895
4.5	Miscellaneous Commands	897

CHAPTER 5
SYMBOLS AND DDT ASSEMBLY

5.1	Defining Symbols	899
5.2	Deleting Symbols	900
5.3	DDT Assembly	900

CONTENTS (Cont)

		Page
5.4	Field Separators	901
5.5	Expression Evaluation	902
5.6	Special Symbols	902
5.6.1	Order of Symbol Table Search	902
5.6.2	Order of Symbol Table Search for Symbol Education	903
5.7	Special Symbols	903
5.8	Binary Value Interpretation	903

CHAPTER 6
PAPER TAPE

6.1	Paper Tape Control	905

APPENDIX A
SUMMARY OF DDT FUNCTIONS

A.1	Type-Out Modes	907
A.2	Address Modes	907
A.3	Radix Change	907
A.4	Prevailing vs. Temporary Modes	907
A.5	Storage Words	908
A.6	Related Storage Word	908
A.7	One-Time Only Typeouts	909
A.8	Typing In	909
A.9	Symbols	910
A.10	Special DDT Symbols	910
A.11	Arithmetic Operators	911
A.12	Field Delimiters in Symbolic Type-Ins	911
A.13	Breakpoints	911
A.14	Conditional Breakpoints	912
A.15	Starting the Program	912
A.16	Searching	912
A.17	Unused Functions	913
A.18	Zeroing Memory	913
A.19	Special Characters	913
A.20	Paper Tape Commands	914

CONTENTS (Cont)

Page

APPENDIX B
EXECUTIVE MODE DEBUGGING (EDDT) 915

APPENDIX C
STORAGE MAP FOR USER MODE DDT 917

APPENDIX D
OPERATING ENVIRONMENT

D.1 Entering and Leaving DDT 919
D.2 Loading and Saving DDT 920
D.3 Explanation 921

ILLUSTRATIONS

6-1 RIM10B Block Format 906

TABLES

3-1 Special Character Functions 882

CHAPTER 1
INTRODUCTION

DDT-10 (for Dynamic Debugging Technique)* is used for on-line checkout and testing of MACRO-10 and FORTRAN programs and on-line program composition in all PDP-10 software systems.

After the user's source program has been assembled or compiled, the user's binary object program (with its symbol table) may be loaded along with DDT. DDT occupies about 2K of core.

By typing commands to DDT, the user may set breakpoints where DDT will suspend execution of his program and await further commands. This allows the user to check out his program section by section. Either before starting execution or during breakpoint stops, the user may examine and modify the contents of any location. Insertions and deletions may be done in symbolic source language or in various numeric and text modes at the user's option. DDT also performs searches, gives conditional dumps, and calls user-coded debugging subroutines at breakpoints.

Symbolic on-line debugging with DDT provides a means for rapid checkout of new programs. If a bug is detected, the programmer makes changes quickly and easily and may then immediately execute the corrected section of his program.

1.1 LOADING PROCEDURE

The user loads the program to be debugged and DDT with the Linking Loader. (The /D switch commands the Loader to load DDT.) To transfer control to DDT, the user types the monitor command,

DDT ↵

After DDT responds by skipping two lines, the user may begin typing commands to DDT.

1.2 LEARNING TO USE DDT

This manual is designed to make DDT easy to use. A survey was made of several programmers who use DDT frequently, and it was learned that most debugging is done with a limited set of commands. These basic commands are described in the next chapter. When learning DDT, it is recommended that the reader concentrate on learning to use the commands in Chapter 2. If more detailed information is required, skip ahead to later chapters.

After reading Chapter 2, practice debugging, using the basic commands. This may be all that will ever be needed. Read the following chapters which describe the entire command set in detail; this should be read when the basic commands are understood.

After learning the system, the Summary of Commands, listed by function in Appendix A, will be useful for quickly finding any DDT command.

CHAPTER 2
BASIC DDT COMMANDS

The DDT commands most frequently used by programm s are described in this chapter. Many programs are debugged successfully using only these basic commands.

This chapter introduces the main features of DDT to the uninitiated user. Later chapters describe in detail these basic commands, less frequently used commands and other more complex options.

2.1 EXAMINING STORAGE WORDS

By using DDT, a programmer may examine the contents of any storage word by typing the address of the desired word followed immediately by a slash (/). For example, to type out the contents of a location whose symbolic address is CAT, the user types,

```
CAT/
```

DDT now types out the contents (preceded and followed by tabs) on the same line[1].

```
CAT/  MOVEM AC,DOG+21
```

The word labeled CAT is now considered to be opened, and DDT has set its location pointer to point to this address.

2.2 TYPE-OUT MODES

The preceding example showed DDT typing out the contents of location CAT as a symbolic instruction with its address field also relative to a symbol. This is the type-out mode in which DDT is initialized. It is also initialized to type all numbers in the octal radix. The user may ask DDT to re-type the preceding quantity as a number in the current radix by typing an equal sign (=). For example[2],

```
CAT/  MOVEM AC,DOG+21 = 202400,,6736
```

DDT has numerous commands which reset the type-out mode permanently, temporarily, or for only one typeout. The modes that can be selected include numeric constants, floating point numbers, ASCII and SIXBIT text modes, and half-word format. Absolute or relative addressing and different radixes may similarly be selected. For example, to change the current type-out mode to ASCII text, the user types the command[3]

```
$T
```

[1] In this manual information typed out by DDT is underlined to distinguish DDT output from user-typed input.
[2] The two commas indicate that 202400 is in the left half of CAT, and 6736 is in the right half.
[3] The Teletype keys ALTMODE (ALT), PREFIX (PREFIX), or ESCAPE (ESC) are all equivalent and echo as $.

or, to change the current type-out mode to half-word format, he types

```
$H
```

or, to select decimal numbers in his typeouts, he types

```
$10R
```

Using these commands (and others described in Chapter 3), a programmer may examine any location in the mode most appropriate to the information stored there. A semicolon (;) commands DDT to retype the preceding quantity in the current mode. Combining this command with a mode change gives results such as the following:

```
       CAT/  MOVEM AC,DOG+21   $10R;  MOVEM AC,DOG+17
or     CAT/  MOVEM AC,DOG+21   $H;    202400,,DOG+21
or     TEXT/  ANDM 1,342212(10)  $T;  ABCDE
```

2.3 MODIFYING STORAGE WORDS

Once a word has been opened, its contents may be changed by typing the desired new contents immediately following the typeout produced by DDT. A carriage return will command DDT to make the indicated modification and close the word. For example,

```
CAT/  MOVEM AC,DOG+21   MOVNM AC2,DOG+21 ⤸
```

The carriage return simply closes the previously examined register without opening another[1]. The line feed (↓) may also be used to close a word after examining (and optionally modifying) it. The line feed commands DDT to (1) echo a carriage return, (2) close the current word (making a modification if one was typed), (3) add one to DDT's location pointer, and (4) type out the new pointer value and the contents of that address. Thus, if a line feed had been used in the previous example, the result would be:

```
CAT/  MOVEM AC,DOG+21   MOVNM AC2,DOG+21↓
CAT+1/  AOBJN XR6,LOOP5
```

Location CAT+1 is now open and may be modified if desired.

The vertical arrow (↑) is similar to the line feed command except that the location counter is decremented by one. Therefore, if the user continued the previous example by typing ↑ the result would be

```
CAT+1/  AOBJN XR6, LOOP5↑
CAT/  MOVNM AC2,DOG+21
```

[1] The carriage return command has the additional property of causing temporary type-out modes to revert to permanent mode.

Location CAT is thus displayed and shows the result of the modification made in the previous example.

The tab (⇥) and backslash (\ ) both close the current register and open the address last typed (whether typed by DDT or the user). However, tab sets DDT's location pointer (.) to this new address while backslash leaves it unaltered. A more complex example may clarify the usefulness of these commands.

```
CAT+1/      AOBJN XR6,LOOP5 ⇥
LOOP5/      CAMGE AC2,TABL(XR6)  CAMG AC2,TABL+1(XR6)\  SETZI 0=401000,,0↓
LOOP5+1/    JUMPL AC3,FAULT JUMPL AC2,FAULT⇥
FAULT/      JRST 4,FAULT
```

2.4 TYPE-IN MODES

The examples in the preceding section showed modifications made as symbolic instructions in a form identical to MACRO-10 machine language. It is also possible to enter various numbers and forms of text.

Octal values may be typed in as octal integers with no decimal point. To be interpreted as a decimal number, an integer must be followed by a decimal point. Numeric strings with numbers following the decimal point imply decimal floating-point numbers. The E-notation may also be used on floating-point numbers. Some examples are:

Octal:	1234	777777777777	-6	0
Decimal integers:	6789.	99999999.	-25.	0.
Floating-point numbers:	78.1	0.249876E-10	-4.00E+20	0.0
Incorrect formats:	76E+2	76.E+2 (instead write 76.0E+2)		

To enter ASCII text (up to five characters, left justified in a word), type a double quote (") followed by any printing character to serve as a delimiter, then type the one to five ASCII characters and repeat the delimiter. For example:

"/ABCDE/	(/ is the delimiter)
"ABCDA	(A is the delimiter)

Note that the mode of a quantity typed in is determined by the user's input format and is unaffected by any type-out mode settings.

2.5 SYMBOLS

The user's symbol tables are loaded by the Linking Loader when it loads programs and DDT. However, initially DDT is set to treat only global symbols (created by INTERNAL and ENTRY pseudo-ops in MACRO-10) as being defined. This means that only global symbols will be used for relative

address typeouts and, likewise, only these globals can be referenced when typing in symbolic modifications. In order to make the local symbols within a particular program available to DDT, the user types the program name (this comes from the MACRO-10 TITLE statement or the FORTRAN IV SUBROUTINE or FUNCTION statement) followed by ALTMODE and a colon ($:). For example, the command

```
ARCTAN$:
```

will unlock the local symbols in the program named ARCTAN. This provision in DDT permits the user to debug several related subroutines simultaneously and reference the local symbol table of each independently without fear of multiply-defined local symbols. If the user's program is not titled, the command MAIN.$: will unlock the local symbol table.

NOTE

> DDT is not quite so stringent on the use of local symbols as indicated above (see Section 5.6). However, the user is advised to unlock symbols with $: until he is fairly familiar with DDT.

The user may also insert symbols into the symbol table. To insert a symbol with a particular value, type the value, followed by a left angle bracket (<), the symbol, and a colon (:). Some examples are

```
707<CONS: 27<S:   12.1E+<NUMB: ADR+12<ADRX:
```

To assign a symbol with a value equal to DDT's location pointer, simply type the symbol followed by a colon. For example,

```
XREF+4/   JRST @ TABL(3)   BRNCH:
```

will cause BRNCH to be defined with the value XFER+4.

2.6 EXPRESSIONS

DDT permits the user to combine symbols and numeric quantities into expressions by using the following characters to indicate arithmetic operators.

- \+ The plus sign indicates 2's complement addition
- \- The minus sign indicates 2's complement subtraction
- * The asterisk indicates integer multiplication
- ' The single quote or apostrophe indicates integer division (remainder discarded)--slash cannot be used to indicate division since it has another use in DDT.

As usual in arithmetic expressions, the evaluation proceeds from left to right with multiplication and division performed before addition and subtraction.

2.7 BREAKPOINTS

The breakpoint facility in DDT provides a means of suspending program operation at any desired point to examine partial results and thus debug a program section by section. The simpler facts

about breakpoints are presented next; the use and control of conditional breakpoints is deferred to Paragraph 4.2.

2.7.1 Setting Breakpoints

The programmer can automatically stop his program at strategic points by setting as many as eight breakpoints. Breakpoints may be set before the debugging run is started, or during another breakpoint stop. To set a breakpoint, the programmer types the symbolic or absolute address of the word at the location point in which he wants the program to stop, followed by $B. For example, to stop when location 6004 is reached, he types,

```
6004$B
```

Breakpoint numbers are normally assigned by DDT in sequence from 1 to 8. The user may instead assign breakpoint numbers himself when he sets a breakpoint by typing,

```
$NB
```

when n is the breakpoint number ($1 \leq n \leq 8$), for example,

```
CAT+3$4B   DOG+1$7B   6004$8B
```

When the programmer sets up a breakpoint he may request that the contents of a specified word be typed out when the breakpoint is reached. To do this, the address of the word to be examined is inserted, followed by two commas, before the breakpoint address. Some examples are

```
DOG,,CAT$3B AC1,,LOOP+2$B X,,6004$8B
```

2.7.2 Breakpoint Restrictions

The locations where breakpoints are set may not

a. be modified by the program
b. be used as data or literals
c. be used as part of an indirect addressing chain
d. contain the user mode monitor command INIT
e. be accumulator 0.

2.7.3 Breakpoint Type-Outs

When the breakpoint location is reached, DDT suspends program execution without executing the instruction at the breakpoint location. DDT then types the breakpoint number and the Program Counter value at the time the breakpoint is reached (this value will differ from the typed-in breakpoint address if the breakpoint is executed by an XCT instruction elsewhere in the program). The format of this typeout is as shown in the following examples:

```
$4B >> CAT+3 $7B >> DOG+1 $8B >> 6004
```

If the user requested that a specified address be examined at that breakpoint, it will be opened; for example,

```
$3B >> CAT DOG/ SOJGE 3,GOAT+6
```

2.7.4 Removing and Reassigning Breakpoints

The user may remove a breakpoint by typing,

ØSNB

where n is the number of the breakpoint to be removed. For example,

Ø$2B

removes the second breakpoint. All assigned breakpoints are removed by typing

$B

The user may reassign a breakpoint without formally removing it. Thus, if he has set breakpoint No. 2 at location ADR (via the command ADR$2B) he may reassign No. 2 to LOC+6 by typing LOC+6$2B.

2.7.5 Proceeding From a Breakpoint

Program execution may be resumed (in sequence) following a breakpoint stop by typing the proceed command, $P.

If the user does not wish to stop until the nth time that this breakpoint is encountered he types,

N$P

Then this breakpoint will be passed n-1 times before a break occurs.

2.8 STARTING THE PROGRAM

The program is started by typing

$G

This starts the program at the previously specified starting address in location JOBSA. (Typically this is the address from the MACRO-10 END statement.) The programmer may start at any other location by typing that address followed by $G. For example,

4ØØØ$G

starts the program at the instruction stored at location 4000. BEGIN$G starts the program at the symbolic location BEGIN.

The start command may also be used to restart from a breakpoint stop when it is not desired to continue in sequence from the point where program execution was suspended.

2.9 DELETING TYPING ERRORS

Any partially typed command may be deleted by pressing the RUB OUT key. This causes DDT to ignore any preceding (unexecuted) partial command, and DDT types XXX. The correct command may then be retyped.

2.10 ERROR MESSAGES

If the user types an undefined symbol which cannot be interpreted by DDT, U is typed back. If an illegal DDT command is typed, or a location outside the user's assigned memory area is referenced ? is typed back.

2.11 SUMMARY

As was said in the beginning, these basic commands are sufficient for debugging many programs. Complete descriptions of all DDT commands are explained in the following chapters.

CHAPTER 3
DDT COMMANDS

When DDT is initialized, it is set to type out in the symbolic instruction format with relative addresses, and to type out numbers in octal radix.

3.1 EXAMINING THE CONTENTS OF A PROGRAM STORAGE WORD

To type out the contents of a storage word, the programmer types the address, followed immediately by a slash (/). For example, to examine the contents of a word whose symbolic address is ADR, the user types,

```
ADR/
```

DDT types out the contents on the same line. In this manual, information typed out by DDT is underlined.

```
ADR/  MOVE A,CC1
```

The word labeled ADR is now considered to be opened, and DDT continues to point to this address. The point, or period, character (.) represents DDT's location pointer, and may be used to type out its contents, as in the following command.

```
./  MOVE A, CC1
```

Since we did not change the contents, they are the same, but we used the location pointer to re-examine the currently opened word. Similarly, the programmer may use the period (.) as an arithmetic expression component, such as

```
.+5/  SOJGE  2,ADR+3
```

DDT's location pointer is set to a new value by the / command when immediately preceded by an address. For example,

```
201/  0
```

sets the location pointer to 201. If the user types / without typing an address, the contents of the location addressed in the last typeout are typed.

```
667/  MOVE 1,6  /  0
./    MOVE 1,6
```

Location 667 contains the instruction MOVE 1,6. The second slash displays the contents of Accumulator 6, which is zero. This does not change the location pointer, which is still pointing to location 667.

```
ADR/  MOVE A,CC1  /  ADD 2,SUM+7
```

It should also be noted that the spaces, which occur after DDT completes the typing of the contents of ADR, are automatically produced by DDT, not the user.

The left square bracket ([)[1] has the same effect as the slash, (the address immediately preceding the [will be opened). However, [forces the typeout to be in numbers of the current radix.

```
ADR[  11   (OCTAL)
ADR]  9.   (DECIMAL)
```

The right bracket (])[1] has the same effect as the slash except that it forces the typeout to be in symbolic instructions.

```
ADR+23]   MOVE  15,LIST+2
```

The exclamation point (!) works like the slash except that it suppresses type out of contents of locations until either /, [, or] is typed by the user. The LINE FEED (↓) commands DDT to type out the contents of ADR+1.

```
ADR!  MOVE AC,555↓     (1)
ADR+1! ↵               (2)
ADR/  MOVE AC,555      (3)
```

Thus, in step (1) of the example the contents of ADR are not typed out, but the address is opened to modification and MOVE AC,555 has been typed in by the user.

Step (2) of the example shows that the location pointer has been incremented by one and the contents of ADR+1 are not typed out. This is because the exclamation point is still in effect and will continue to take effect until /, [, or] is typed in by the user. In this case, the slash terminates the effect of the exclamation point.

Step (3) shows that the modification (MOVE AC,555) of ADR typed in Step (1) has been accomplished.

3.2 CHANGING THE CONTENTS OF A WORD

After a word is opened, its contents can be changed by typing the new contents following the type out by DDT, followed by a carriage return. For example,

```
ADR/  MOVE A,CC1  MOVE A,CC2 ↵
```

The carriage return closes the open word, but does not move the location pointer. A LINE FEED (↓) command could also be used to make this modification. A LINE FEED causes a carriage return, adds

[1]On Teletype Models 33 and 35 the left square bracket ([) is produced by holding the SHIFT key down and striking the K key. The right square bracket (]), is produced by holding the SHIFT key down and striking the M key.

one to DDT's location counter (moves the pointer), types out the resulting address and the contents of the new address. Thus, if we conclude our last example with a LINE FEED

```
ADR/      MOVE A,CC1  MOVE A,CC2↓
ADR+1/    ADD 3,CC3
```

ADR+1 is now open, and may be modified by the user.

The vertical arrow (↑)[1] works similarly, except that one is subtracted from the location pointer. The open word is closed (modified if a change is given) and the new address and contents are typed out.

```
ADR+1/    ADD 3,CC3↑
ADR/      MOVE A,CC2
```

Since the vertical arrow subtracts one from the pointer, the resulting address is ADR, and the contents now show the change made in the previous example.

3.3 INSERTING A CHANGE, AND EXAMINING THE CONTENTS OF THE LAST TYPED ADDRESS

The horizontal tab (→|) causes a carriage-return line feed, then sets the location pointer to the last address typed (the new address if a modification was made) of the instruction in the register just closed. Then DDT types this new address, followed by a slash and the contents of that location, as shown below.

```
ADR5/   JRST ADR1   JRST ADR→|
ADR/    MOVEM B,CC2 →|
CC2/    666
```

The backslash (\)[2] opens the word at the last address typed and types out the contents. However, backslash does not change the location pointer. The backslash closes the previously opened word and causes it to be modified if a new quantity has been typed in.

```
ADR/  MOVE A,CC2   JRST X\ MOVE AC,3
```

The use of the backslash accomplishes two things. First it changes ADR by replacing its contents with JRST X. Second, the backslash causes DDT to type out the contents of X, namely, MOVE AC,3. The location pointer continues to point to ADR, but now location X is open and may be modified if desired.

---

[1] ↑ is produced by SHIFT-N on Teletype Models 33 and 35. The backspace key may be used instead of ↑ on Teletype Model 37.

[2] \ is produced by SHIFT-L on Teletype Models 33 and 35.

If the line-feed control character and the vertical arrow were used in conjunction with the backslash, the results would be as follows.

```
ADR/      MOVEM B,CC2  MOVE A,CC1\ 105776 ↓
ADR+1/    MOVE A,C     ↑
ADR/      MOVE A,CC1   \  105776
```

The following is a summary in table form of these special control characters and their corresponding functions. For example, the chart shows that the forward slash (/) will examine the contents of an address, type out in the current mode, open the address, change the location pointer to the address just opened, but it does not cause a new quantity to be inserted in that address.

Table 3-1
Special Character Functions

Command Character	Type Out Contents	Mode	Address Opened	Change Location Pointer	Insert New Qty If New Qty Has Been Typed
/	Yes	Current	Yes	Yes[1]	No
[	Yes	Numeric	Yes	Yes[1]	No
]	Yes	Symbolic	Yes	Yes[1]	No
!	No	None	Yes	Yes[1]	No
\	Yes[2]	Current	Yes	No	Yes
TAB (⇥)	Yes[2]	Current	Yes	Yes	Yes
↑ or backspace	Yes[2]	Current	Yes	Yes (.-1)	Yes
Line-feed (↓)	Yes[2]	Current	Yes	Yes (.+1)	Yes
Carriage return (↵)	No	None	No (closes)	No	Yes

A ? typed by DDT when examining a location indicates that the address of the location is outside the user's assigned memory area. A ? typed when depositing indicates that the location cannot be written in, because it is either outside the assigned memory area or inside a write-protected memory segment.

[1] If a user-typed quantity preceded.
[2] If ! has not suppressed typeout.

3.4 STARTING THE PROGRAM

The program is started by typing

```
$G
```

This starts the program with the instruction beginning at the user's previously specified starting address taken from location JOBSA. The programmer may start at any other instruction by typing the address of that instruction followed by $G. For example,

```
4000$G  OR  ADR+5$G
```

starts the program at the instruction stored at location 4000 or, in the second part, at the symbolic address ADR+5. The start command may also be used to restart from breakpoints when the user does not wish to proceed to the next instruction.

3.5 ONE-TIME TYPEOUTS

These commands cause a single typeout of the opened word in the mode indicated.

3.5.1 Type Out Numeric

Although DDT is initialized to type out in symbolic mode, it is often useful to change to numeric typeout. When the programmer types the equal sign (=), the last expression typed is retyped by DDT in the current radix (initially octal). This is useful when a symbolic typeout is meaningless. Since this usually indicates that numeric data is stored in that word, the user can verify this by typing = and checking the value.

3.5.2 Type Out Symbolic

If a typeout is numeric, and the user wants to examine it in symbolic mode, he types the left arrow (←). The last typed quantity is retyped as a symbolic instruction. The address mode is determined by $A or $R.

3.5.3 Type Out in Current Mode

To retype a typeout in the current mode, the user types a semicolon (;). This may be used, for example, if the user has changed the typeout mode. For example,

```
TEXT/ ANDM 1,342212 (10) $T; ABCDE
```

3.6 SYMBOLS

Before DDT commands can be used to reference local symbols in the program Symbol Table, the user should type the program name as specified in the MACRO-10 TITLE statement, or the FORTRAN IV

SUBROUTINE or FUNCTION statement, followed by an ALTMODE and a colon. For example,

```
MAIN$:
```

makes the local symbols in the program called MAIN available. Since the user can debug several related subroutines simultaneously, reference to several independent symbol tables is permitted, each of which may use the same local symbols with different values. DDT allows the user to reference unique local symbols in other programs without respecifying the program name with $: (see Section 5.6.2). However, to access a local symbol that is used in several programs, the user must specify the program name to remove the ambiguity. Global symbols, such as those specified in MACRO-10 INTERNAL statements, may always be referenced.

The user may insert (or redefine) a symbol in the symbol table by typing the symbol, followed by a colon. The symbol will have a value equal to the address of the location pointer (.).

```
X/  ADDI 3,N  TAG:
```

causes TAG to be defined with the same value as X. All user defined symbols are global.

The user may also directly assign a value to a symbol by typing the value, a left angle bracket (<) and the symbol, terminated by a colon. This is the equivalent of a MACRO-10 direct assignment statement. Some examples are,

```
707<CONS:   12.1E+2<NUMB:
 27<X:      101<MIL:
```

3.7 TYPING IN

To change or modify the contents of a word, the user may type symbolic instructions, numbers, and text characters. Type-ins are interpreted by DDT in context. That is, DDT tests the data typed in to determine whether it is to be interpreted as an instruction, a number (octal or decimal), or text. Typeout mode settings, such as $S, $C, and $nR, do not affect typed input.

The user may type the following:

a. Symbolic Instructions

b. Numbers

   (1) Octal integers
   (2) Fixed-point decimal integers
   (3) Floating-point decimal mixed numbers

c. Text

   (1) Up to five PDP-10 ASCII characters, left justified in a word
   (2) Up to six SIXBIT characters, left justified in a word
   (3) A single PDP-10 ASCII character, right justified in a word
   (4) A single SIXBIT character, right justified in a word

d. Symbols

   Anything that is not a number or text is interpreted by DDT as a symbol.

3.7.1 Typing In Symbolic Instructions

In general, a symbolic instruction is written for insertion by DDT, in the same way the instruction is written as a MACRO-10 source program statement. For example,

```
X/  0  ADD  AC1,DATE
```

where a space terminates the operation field, and a comma terminates the accumulator field. For example: (1) In DDT, the operation code determines the interpretation of the accumulator field. If an I/O instruction is used, DDT inserts the I/O device number in the correct place, and (2) indirect and indexed addresses are written, as in MACRO-10 statements, where @ precedes the address to set the indirect bit, and the index register specified follows in parentheses.

```
X/0  ADD  4,@NUM(17)
```

To type in two 18-bit halfwords, the left and right expressions are separated by two commas. For example,

```
X/  0 A,,B
```

This is similar to the MACRO-10 statement

```
XWD  A,B
```

3.7.2 Typing In Numbers

A typed-in number is interpreted by DDT as octal if it does not contain a decimal point. The following examples are octal type-ins:

```
1234   -10101
772    777777777777
```

Fixed-point decimal integers must contain a decimal point with no digits following.

```
1234.   -99.   877.
```

Floating-point numbers may be written in two formats. With a decimal point and a digit following the decimal point:

```
101.1  1234.5  999.0  -2.71828
```

Or as in MACRO-10, with E indicating exponentiation:

```
12.0E+2  77.0E+5  12.34E2  31.4159E-1
```

3.7.3 Typing In Text Characters

To type in up to five PDP-10 ASCII characters, left justified in an opened word, the user types a quotation mark, followed by any printing delimiting character, then the text characters, and terminated by the delimiting character. The following examples are legal:

```
"/TEXT/    "ABCDEFA
```

In these cases, / and A are the delimiting characters

To type in up to six SIXBIT characters, left justified in an opened word, the user types ALTMODE quotation mark ($"), followed by any delimiting character, then the text characters, and terminated by repeating the delimiting character. Lower case letters are converted to upper case. Characters outside the SIXBIT set are illegal, and DDT types a question mark. The two examples below are SIXBIT type ins.

```
$"/DIVIDE/   $"EXXXXXXE
```

To type in a single PDP-10 ASCII character, right justified in an opened word, the user types a quotation mark, followed by a single ASCII text character, then by an ALTMODE.

```
"@$   "/$   "?$
```

To type in a single SIXBIT character, right justified in an opened word, the user types an ALTMODE, followed by a quotation mark, a single SIXBIT text character and terminated by an ALTMODE.

```
$"@$   $"M$   $"$$
```

3.7.4 Arithmetic Expressions

Numbers and symbols may be combined into expressions using the following characters to indicate arithmetic operations.

+ The plus sign means 2's complement integer addition.
- The minus sign means 2's complement integer subtraction.
* The asterisk means integer multiplication.
' The single quote means integer division with any remainder discarded. (The slash has another function.)

Symbols and numbers are combined by +, -, *, ' to form expressions. Examples:

```
6+2
S'2.51+BASE
2*3+1
```

3.8 DELETE

Any partially typed command may be deleted by pressing the RUB OUT or DELete key. This causes DDT to ignore any preceding (unexecuted) partial command and DDT types XXX. The correct command may then be retyped.

3.9 ERROR MESSAGES

If the user types an undefined symbol which cannot be interpreted by DDT, U is typed back. If an illegal DDT command is typed, ? is typed back. Examining or depositing into a location outside the user's assigned memory area causes DDT to type a ?. Depositing in a write-protected high memory segment also results in a ? typeout.

3.10 UPPER AND LOWER CASE (TELETYPE MODEL 37)

DDT will accept alphabetic input in either upper or lower case. Lower case letters are internally converted to upper case, except when inputting text where they are taken literally as explained in Section 3.7.3.

DDT output is in upper case, except for text which is taken literally.

CHAPTER 4
MORE DDT-10 COMMANDS

This chapter describes other type-out modes, conditional breakpoints, searches and additional features. Commands are available to change modes from the initial settings so that numeric data can be typed out in a radix chosen by the user, in floating-point format, in RADIX50 format, as halfwords (two addresses) and as bytes of any size. The contents of a storage word may also be typed out as 7-bit PDP-10 ASCII text, or SIXBIT text characters. (See MACRO-10 Manual, Appendix E.)

Searches can be made in any part of the program for any word, not-word (inequality), or effective address. The user specifies the instruction or data to be searched for and the limits of the search.

Breakpoints can be set conditionally, so that a program stop occurs if the condition is satisfied. In addition, a counter can be set up allowing the user to specify the number of times a breakpoint is passed before a program stop occurs.

4.1 CHANGING THE OUTPUT RADIX

Any radix ($\geq$2) may be set by typing $nR, where n is the radix for the next typeout only, and n is interpreted by DDT as a decimal value. The radix is permanently changed when the double ALTMODE is used in the command $$nR. To change the type-out radix permanently to decimal, the user types,

```
$$10R
```

When the output radix is decimal, DDT follows all numbers with a point.

4.2 TYPE-OUT MODES

When DDT-10 is loaded, the type-out modes are initialized to produce symbolic instructions with addresses relative to symbolic locations. For numeric typeouts, the radix is initially set to octal.

These modes may be changed by the user. The duration, or lasting effect of a type-out mode change is also set by the user. Prevailing modes, which are semipermanent, are preceded by two ALTMODEs. Temporary modes are preceded by a single ALTMODE. In addition, some mode changes effect only one typeout, such as the equal sign, which causes DDT to retype the last typed quantity in numeric mode.

In general, prevailing modes are changed by replacing them with another prevailing mode or by reinitializing the system. Temporary modes remain in effect until the user types a carriage return (⤸), or re-enters DDT. One-time modes apply only to a single typeout.

4.2.1 Primary Type-out Modes

$S (OR $$S) — Type out symbolic instructions. The address part interpretation is set by $R or $A.

$S ADR/ ADD AC1,TABLE+3

$A (OR $$A) — Type out the address parts of symbolic instructions, and both addresses when the mode is halfword, as absolute numbers in the current radix.

$A ADR/ ADD 4002

$R (OR $$R) — Type out addresses as relative addresses.

$C (OR $$C) — Type out constants, i.e., as numbers in the current radix.

$C ABLE/ 254111,,4050

If the output radix is octal and the left half is not 0, the word will be divided into halves separated by commas.

$F (OR $$F) — Type out the contents of storage words as floating-point numbers.

$F X/ 0.17516230E-45

Unnormalized numbers are typed out as signed decimal integers.

$T (OR $$T) — Type out as 7-bit ASCII text characters. Left-justified characters are assumed unless the leftmost character is null. If the leftmost character is null, then right-justified characters are assumed.

$T REX/ ABCDE

$6T (OR $$6T) — Type out as SIXBIT text characters.

$6T HEX/ ABCDEF

$5T (OR $$5T) — Type out symbols in radix 50 mode. (See MACRO-10 Manual, Appendix 6.)

$5T 13774/ 4 CREF = 40003,,261550

$H (OR $$H) — This command causes the typeout to be in halfwords, the left half separated from the right half by double commas. The address mode interpretation is determined by $R or $A.

$A $H Z/ 4503,,4502

$R $H Z/ TABL+14,,TABL+13

$NO (OR $$NO) — Type out in n-bit bytes, where n is decimal. (Use the letter O, not zero).

$6O BYTS/ 22,23, 1, 73, 51, 46

As in all DDT typeouts, leading zeros are suppressed.

4.3 BREAKPOINTS

4.3.1 Setting Breakpoints

The programmer can automatically stop his program at strategic points by setting up to eight breakpoints. Breakpoints may be set before the debugging run is started, or during another breakpoint stop. To set a breakpoint, the programmer types the symbolic or absolute address of the word at the location which he wants the program to stop, followed by $B. For example, to stop when location 4002 is reached, he types,

```
4002$B
```

If all eight breakpoints are in use, DDT will type a question mark. The user may assign breakpoint numbers when he sets a breakpoint by typing ADR $nB, where n is the breakpoint number $(1<n<8)$. For example,

```
SYM$3B   ADR$7B
```

If n is not entered DDT will assign 1 through 8 in sequence. In the previous example, when ADR is reached, DDT types,

```
$7B >> ADR
```

indicating that the break has occurred at location ADR, and breakpoint No. 7 was encountered. The break always occurs before the instruction at the breakpoint address is executed.

If the instruction at the breakpoint location is executed by an XCT instruction, the typeout will show the address of the XCT instruction, not the location of the breakpoint. The program stops at each breakpoint address, and the programmer can then type other commands to examine and debug his program.

When the programmer sets a breakpoint, he may request that the contents of a word be typed out when a breakpoint is reached. To do this, the address of the word to be examined is inserted, followed by two commas, before the breakpoint address.

```
X,,4002$2B
```

When address 4002 is reached, DDT types out,

```
$2B>>4002  X/  ADD  AC,Y+2
```

where ADD AC, Y+2 is the contents of X. Location X is left open at this point. Location 0 may not be typed out in this way because a zero argument implies no typeout.

4.3.2 Removing Breakpoints

The user may remove a breakpoint by typing,

```
0$NB
```

where n is the number of the breakpoint to be removed. Therefore,

```
0$2B
```

removes the second breakpoint. All assigned breakpoints are removed by typing

```
$B
```

The user may reassign a breakpoint. If he has set breakpoint No. 2 at location ADR (ADR$2B), he may reassign No. 2 to ADR+1 by typing ADR+1$2B.

4.3.3 Restrictions for Breakpoints

Breakpoints may not be set on instructions that are

a. Modified by the program

b. Used as data or literals

c. Used as part of an indirect addressing chain

d. The user mode monitor command, INIT

   A breakpoint at any other monitor command will operate correctly, except that if the monitor command is in error, the monitor will type out an error and the Program Counter, but the Program Counter will be internal to DDT and meaningless to the user.

e. A breakpoint may not be assigned to accumulator 0.

4.3.4 Restarting After a Breakpoint Stop

To resume the program after stopping at a breakpoint, the user types the proceed command,

```
$P
```

The program is restarted by executing the instruction at the location where the break occurred. If the user types n$P, this breakpoint will be passed n-1 times before a break can occur; the break will occur the nth time. If n is not specified, it is assumed to be one. If the user proceeds by typing $$P (or n$$P), the program will proceed automatically when the program breaks again. If DDT encounters an XCT loop or the monitor command INIT when proceeding, a question mark will be typed.

Alternatively, the user may restart at any location by typing the start command,

```
ADR$G
```

where ADR is any program address, or $G, which restarts at the previously specified starting address in location JOBSA.

4.3.5 Automatic Restarts from Breakpoints

If the user requests DDT to type out the contents of a word and then continue program execution without stopping, he types two ALTMODES when specifying the breakpoint address.

```
AC,,ADR$$B
```

When ADR is encountered, the contents of AC are typed out and program execution continues. To get out of the automatic proceed mode, type any Teletype key during the typeout, and then remove the breakpoint or reassign it with a single ALTMODE. It may be necessary to use ↑C and DDT ↵ to get back to DDT to remove or reassign the breakpoint.

4.3.6 Checking Breakpoint Status

The user may determine the status of a breakpoint by examining locations $nB, $nB+1, and $nB+2.

$nB contains the address of the breakpoint in the right half and the address of the location to be examined in the left half. If both halves equal zero, the breakpoint is not in use.

$nB+1 contains the conditional breakpoint instruction. (See Paragraph 4.3.7.)

$nB+2 contains the proceed count.

4.3.7 Conditional Breakpoints

Breakpoints may be set up conditionally in two ways. The user may provide his own instruction or subroutine to determine whether or not to stop, or he may set a proceed counter which must be equal to or less than zero in order for a break to occur.

When a breakpoint location is reached, DDT enters its breakpoint analysis routine consisting of five instructions.

```
SKIPE   $NB+1            ; Is the conditional break instruction 0?
XCT     $NB+1            ; No, execute conditional break instruction
SOSG    $NB+2            ; Decrement and test the proceed counter
JRST    break routine
JRST    proceed routine
```

If the contents of $nB+1 are zero (indicating that there is no conditional instruction), the proceed counter at $nB+2 is decremented and tested. If it is less than or equal to zero, a break occurs; if it is greater than zero the execution of the user's program proceeds with the instruction where the break occurred.

If the conditional break instruction is not zero, it is executed. If the instruction (or the closed subroutine) does not cause a program counter skip, the proceed counter is decremented and tested as above. If a program counter skip does occur, a break occurs. If the conditional instruction is a call to a closed subroutine which returns skipping over two instructions, execution of the user's program proceeds.

If the user wishes a break to occur based only on the conditional instruction, he should set the proceed counter to a large positive number so that the proceed counter will never reach zero.

4.3.7.1 Using the Proceed Counter - If the user wishes to proceed past a breakpoint a specified number of times, and then stop, he inserts the number of passes in $nB+2, which contains the proceed count.

The proceed counter may be set in two ways. The first way is by direct insertion. For example,

```
$NB+2/  0  20
```

sets the counter to 20. The second method is as follows. After stopping at a breakpoint, the proceed count may be set (or reset) by typing the count before the proceed command:

```
20$P
```

($P will proceed from the interrupted instruction sequence even if the breakpoint has been removed or reassigned.)

4.3.7.2 Using the Conditional Break Instruction - The user inserts a conditional instruction, or a call to a closed subroutine at $nB+1. For example,

```
$3B+1/  0  CAIGE  ACC,15)
```

or

```
$4B+1/  0  JSA    16, TEST)
```

When the breakpoint is reached, this instruction or subroutine is executed. If the instruction does not skip or the subroutine returns to the next sequential location, the proceed counter is decremented and tested, as explained in Paragraph 4.2.7. If the instruction skips or the subroutine returns skipping over one instruction, the program breaks. If the subroutine causes a double skip return, the program proceeds with the instruction at the breakpoint address.

Examples of Conditional Breakp ints

If address 6700 is reached and DDT's No. 4 breakpoint registers are as follows:

```
$4B/        AC1,,6700
$4B+1/      CAIE AC1,100
$4B+2/      200
```

AC1 contains 100, and DDT types

```
$4B>6700  AC1/  100
```

Since AC1 contains 100, the compare instruction skips and the program breaks. If AC1 did not contain 100, $4B+2 would be decremented by one and the user's program would continue running.

If the conditional break instruction transfers to a subroutine which, after the subroutine is executed, returns to the calling location +3, a break will never occur regardless of the proceed counter.
Example: If the internal DDT breakpoint registers ($2B and $2B+1) have the following contents, a break would not occur unless accumulator 3 contains 100.

```
$2B/        ADR
$2B+1/      JSR TEST     (contains PC when JSR to subroutine
TEST/       0             TEST is made)
TEST+1/     AOS TEST
TEST+2/     CAIE 3,100
TEST+3/     AOS TEST
TEST+4/     JRST @ TEST
```

The subroutine TEST causes a double skip (the return is to the third instruction after the call) in DDT if accumulator 3 does not equal 100. A break will never occur at address ADR (regardless of the proceed counter) unless accumulator 3 contains 100.

4.3.8 Entering DDT from a Breakpoint

When a break occurs, the state of the user's program is saved, the JSR breakpoint instructions are removed, and the programmer's original instructions are restored to the breakpoint locations. DDT types out the number of the breakpoint and a symbol indicating the reason for the break, >for the conditional break instruction, >>for the proceed counter and the address in the user's program where the break occurred.

Example: If address ADR is reached in the user's program and DDT's breakpoint registers contain:

```
$2B/        ADR
$2B+1/      0
$2B+2/      0              (proceed counter contains zero)
```

DDT stops the program and types,

```
$2B>>ADR
```

4.4 SEARCHES

There are three types of searches: the word search, the not-word search, and the effective address search.

Searches can be done between limits. The format of the search command is,

a<b >c$	W	Word search
	N	Not-word search
	E	Effective address search

where:

- a Is the lower limit of the search; 0 is assumed if this argument and its delimiter are not present.
- b Is the upper limit of the search. The lower numbered end of the symbol table is assumed if this argument and its delimiter are not present.
- c Is the quantity searched for.

The effective address search (E) will find and type out all locations where the effective address, following all indirect and index-register chains to a maximum depth of 64_{10} levels, equals the address being searched for.

Examples:

```
4517<5000>X$E

INPUT  <5000>700$E
```

Examples of DDT output, when searching for X in the above example, are as follows.

```
4517/  SETZM X

4721/  MOVE 2,X

5000/  MOVE 3, @ 4721
```

(indirectly addresses X through address 4721)

The word search (W) and the not-word search (N) compare each storage word with the word being searched for in those bit positions where the mask, located at $M, has ones. The mask word contains all ones unless otherwise set by the user. If the comparison shows an equality, the word search types out the address and the contents of the register; if the comparison results in an inequality, the word search will type out nothing. The not-word search types nothing if an equality is reached. It types the contents of the register when the comparison is an inequality.

Examples:

```
INPT<INPT+10>NUM$W

INPT<INPT+10>0$N
```

$M/ — This command types out the contents of the mask register, which is then open. The contents of the mask register are ordinarily all ones unless changed by the user.

N$M — Inserts n into the mask register.

0$M — `FIRST<LAST>0$W` — Lists a block of locations by setting the MASK to zero then performing a word search for zero.

4.5 MISCELLANEOUS COMMANDS

$Q This command represents the value of the last quantity typed.

ADR/100,,200 $Q↵ puts back in ADR the quantity 100,200.
$Q+1↵ puts back in ADR the quantity 100,201.
$Q/ displays the contents of location 200.
$Q+1/displays the contents of location 201.

$V This command reverses the two halves of the word and then represents the value of the last quantity typed.

ADR/100,,200 $V↵ puts back in ADR the quantity 200,100.
$V+1↵ puts back in ADR the quantity 200,101.
$V/displays the contents of location 100.
$V+1/displays the contents of location 101.

inst$X This command causes the instruction inst to be executed.

JRST ADR$X starts the user's program at ADR.

FIRST<LAST$$Z This command zeros the memory locations between the indicated FIRST and LAST address inclusively. If the first address is not present, location 0 is assumed. If the last address is not present, the location before the low-numbered end of the symbol table is assumed. Locations 20-137, DDT, and the symbol table are not zeroed.

$Y This command causes a command file to be read and executed. In user mode, the default name for the command file is DSK: PATCH.DDT. The command string $"/NAME/$Y causes the file DSK:NAME.DDT to be interpreted. In exec mode, the command reads a command file from the paper tape reader.

When DDT is reading a command file, rubouts and the character immediately following a carriage return (assumed to be a linefeed) are ignored. Any sequence of DDT commands including $X, $G is legal.

The ? error message is given if (1) a lookup failure occurs on the command file, or (2) this command is not implemented.

CHAPTER 5
SYMBOLS AND DDT ASSEMBLY

A symbol is defined in DDT as a string of up to six letters and numbers including the special characters period (.), percent sign (%), and dollar sign ($). Characters after the sixth are ignored. A symbol must contain at least one letter. If a symbol contains numerals and only one letter, that letter must not be a B, D, or an E. These letters are reserved for binary-shifted and floating-point numbers.

Certain symbols can be referenced in one program from another. These symbols are called "global". Those which can only be referenced from within the same program are called "local" or "internal". Any symbol which has been defined as global by MACRO-10 (using the INTERNAL or ENTRY statements) will be considered as global by DDT-10 when it is referenced. FORTRAN subroutine entry points and COMMON block names are globals. All symbols which the user defines via DDT are defined or redefined as global symbols.

The user may want to reference a local symbol within a particular program. In order to do this he should first type the program name followed by $:. Thus, if a user wishes to use a symbol local to program MIN, he types the command,

```
MIN$:
```

This command unlocks the symbol table associated with MIN. DDT allows the user to reference unique local symbols in other programs without respecifying the program name with $: (see Section 5.6.2). However, to access a local symbol that is used in several programs, the user must specify the program name to remove the ambiguity. The program name is that specified in the MACRO-10 TITLE statement. In FORTRAN, the program name is either MAIN., the name from the SUBROUTINE or FUNCTION statement, or DAT. for BLOCK DATA subprograms.

5.1 DEFINING SYMBOLS

There are two ways to assign a value to a symbol.

NUMERIC VALUE <SYMBOL: This command puts SYMBOL into DDT-10's symbol table with a value equal to the specified NUMERIC VALUE. SYMBOL is any legal symbol defined or undefined.

Example:

```
305<XVAR:
```

XVAR has now been defined to have the value 305.

TAG: This command puts TAG into DDT-10's symbol table with a value equal to the address of the location pointer.

Example:

```
400/    ADD 2, 12012    X:
```

This puts the symbolic tag X into DDT-10's symbol table and sets X equal to 400, the address of the last register opened.

5.2 DELETING SYMBOLS

There are times when the user will want to restrict or eliminate the use of a certain few defined symbols. The following three ways give the user of DDT-10 these capabilities.

SYMBOL $$K — SYMBOL is killed (removed) in the user's symbol table. SYMBOL can no longer be used for input or output.

Example:

```
X$$K
```

This command removes the symbol X from the symbol table.

SYMBOL $K — This command prevents DDT from using this symbol for typeout; it can still be used for typein. For example, the user may have set the same numeric value to several different symbols. However, he does not wish certain symbol(s) to be typed out as addresses or accumulators.

```
X/ MOVE J, SAV J$K ← MOVE N, SAV N$K ← MOVE AC,SAV
```

Since the user does not wish J to be typed out as an accumulator, he types in J$K, followed by a left arrow to type out the contents of X again and MOVE N,SAV is typed out. He then repeats the above process until the desired result, namely AC, is typed out. Any further symbolic typeouts with the same number in the accumulator field of the instruction will type out as AC.

$D — The last symbol typed out by DDT has $K performed on it. The value of the last quantity output is then retyped automatically. For example,

```
A/  MOVE AC,LOC $D MOVE AC,ABC+1
```

5.3 DDT ASSEMBLY

When improvising a program on-line to the PDP-10 on a Teletype, the user will want to use symbols in his instructions in making up the program. In this and in other situations, undefined symbols may be used by following the symbol with the number sign (#). The symbol will be remembered by DDT from then on. Until the symbol is specifically defined by the use of a colon, the value of the symbol is taken to be zero. Successive use of the undefined symbol causes DDT to type out #. Appending # to all subsequent uses of the symbol enables the user to readily identify undefined (not yet defined by a colon) symbols. When an undefined symbol is finally defined, all previously tagged (#) occurrences of the symbol will be filled in.

Example:

```
MOVE 2,VALUE#
```

VALUE is now remembered by DDT and may be used further without the user appending the #. If subsequent instructions are given involving VALUE, DDT appends a # automatically to that symbol. Thus VALUE will always appear as VALUE followed by the # (until VALUE is defined).

Example:

START!	MOVE 2,VALUE#↓	(user types the #)
START+1!	ADDI 2, 50↓	
START+2!	MOVEM 2, VALUE↓	
#		(DDT types #)
START+3!	JRST VALUE+#1↓	(DDT types # after the plus sign because only at that point does DDT realize the symbol VALUE is complete.)
START+4!		

Undefined symbols can be used only in operations involving addition or subtraction. The undefined symbols may be used only in the address field.

Example:

```
MOVEI 2,3*UNDEF#
```

This is an illegal operation - multiplication with a symbolic tag (UNDEF) which has not previously been defined.

The question mark (?) is a command to DDT to list all undefined symbols that have been used in DDT up to that point in the program.

Example:

```
?
VALUE
UNDEF
```

5.4 FIELD SEPARATORS

The storage word is considered by DDT to consist of three fields: the 36-bit wholeword field; the accumulator or I/O device field; and the address field. Expressions are combined into these three fields by two operators:

Space — The space adds the expression immediately preceding it (normally an op code) into the storage word being formed. It also sets a flag so that the expression going into the address field is truncated to the rightmost 18 bits.

Single Comma	The comma does three things: the left half of the expression is added into the storage word; the right half is shifted left 23 bits (into the accumulator field) and added into the storage word. If the leftmost three bits of the storage word are ones, the comma shifts the right half expression left one more place (I/O instructions thus shift device numbers into the device field). The comma also sets the flag to truncate addresses to 18 bits.
Double Comma	Double commas are used to separate the left and right halves of a word with contents expressed in halfword mode.

The address field expression is terminated by any word termination command or character.

5.5 EXPRESSION EVALUATION

Parentheses are used to denote an index field or to interchange the left and right halves of the expression inside the parentheses. DDT handles this by the following generalized procedure.

A left parenthesis stores the status of the storage-word assembler on the pushdown list and reinitializes the assembler to form a new storage word. A right parenthesis terminates the storage word and swaps its two halves to form the result inside the parentheses. This result is treated in one of two ways:

a. If +, -, ', or * immediately precede the left parenthesis, the expression is treated as a term in the larger expression being assembled and therefore may be truncated to 18 bits if part of the address field.

b. If +, -, ', or * did not immediately precede the left parenthesis, this swapped quantity is added into the storage word.

Parentheses may be nested to form subexpressions, to specify the left half of an expression, or to swap the left half of an expression into the right half.

5.6 SYMBOL EVALUATION

5.6.1 Order of Symbol Table Search

DDT references two symbol tables: (1) a built-in operation table containing the machine language instructions and monitor UUOs (e.g., MOVE, JRST, and INIT) and (2) a symbol table constructed by LOADER during the loading process, containing all the user-defined symbols. When a user types into DDT a symbol, which must be converted into a binary value, DDT has two places to look for the symbol. If the expression (see Section 5.5) constructed has a zero value (the normal case when typing in the operation code of an instruction such as the JRST part of a JRST ADDRESS instruction), DDT looks for the symbol first in its internal operation table, and then, if the symbol is not found, in the LOADER constructed symbol table. If the expression constructed is non-zero, DDT searches the LOADER constructed table first, and then the internal operation table. This method of searching the

tables allows instructions such as JRST JRST to work correctly (the first JRST is an operation code, and the second JRST is a user-defined address location).

5.6.2 Order of Symbol Table Search for Symbol Evaluation

When DDT searches the LOADER constructed symbol table to evaluate a symbol typed in, it begins the search by looking through the symbols specified by <program name>$: (see Section 2.5). DDT searches the table in the following order:

1. Looks for the symbol as a local or global symbol in the currently unlocked (by $:) program symbols.
2. Looks for the symbol as a global symbol anywhere in the symbol table.
3. Looks for the symbol as a local symbol in the symbol table of one and only one program.
4. Looks for the symbol as a local symbol that appears in the symbol table of more than one program, but with the same value in each table. (If the symbol appears with different values in different tables, it will not be recognized as defined because there is no way to resolve the ambiguity.)
5. If all the above fail, the symbol is undefined unless it appears in the internal operation table of the DDT.

Fortunately, the searching is accomplished with a single pass over the symbol table.

If one of the several identical local symbols (in step 4) is redefined, it becomes a global, and the symbol is then found at either step 1. or step 2.

This procedure relaxes the requirement of Sections 2.5, 3.6, and the beginning of Chapter 5 on the use of $: to unlock local symbols.

5.7 SPECIAL SYMBOLS

The @ sign sets the indirect bit in the storage word being formed.

Example:

```
MOVE AC,@X
```

5.8 BINARY VALUE INTERPRETATION

When DDT is typing the symbolic equivalent of a binary word or address, it looks for the symbol with a value that best matches the binary. DDT looks through the symbol values in the following order:

1. Searches the symbols of the currently unlocked (by $:) program for a local or global symbol with a value that exactly matches the binary to be interpreted.
2. Searches for a global symbol outside the currently unlocked program with a value that exactly matches the binary to be interpreted.

3. Searches all the other local symbol tables for one or more entries with values that match the binary to be interpreted. If more than one symbolic equivalent is found, the DDT does not use any of them but goes on to step 4. If exactly one symbolic equivalent is found (this includes the case of the same symbol with the same value in more than one local symbol table), then this symbol is used. However, the symbol has a # appended to it to warn the user that this symbol might have a different value in some other local symbol table.

4. Searches the currently unlocked program symbols for a local symbol, and searches the entire symbol table for a global symbol, with the value closest to but less than the binary to be interpreted. The closest symbol is then used for typeout if it is not more than 64 smaller than the binary being interpreted.

If a usable symbol is not found in any of the above steps, the binary is typed out as an integer in the current output radix.

The purpose of this complicated procedure is to output the best symbol without forcing the user to continually respecify the program symbol table names by using $:.

CHAPTER 6
PAPER TAPE[1]

6.1 PAPER TAPE CONTROL

The following commands are used in paper tape control:

$L — This command causes DDT to punch a RIM10B loader on paper tape RIM10B loader. (See MACRO-10 Manual, Chapter 6.) Thus, if the user wishes to punch out a program on paper tape he gives a $L command first in order to get a loader punched on the same tape as the program. Later when the user wishes to read in the program from the paper tape, the hardware READ-IN feature will load the RIM10B loader into the accumulators and then the program will be loaded by the RIM10B loader. (See Figure 6-1.)

FIRST<LAST (TAPE)[2] — This command punches out checksummed blocks in RIM10B format on paper tape from consecutive locations between FIRST and LAST address inclusively. For example, this command will punch out a program existing in core memory in its present state of check-out for later use.

Example:

```
4000<20000 (TAPE)
```

FIRST<LAST $ (TAPE) — This command is similar to the preceding command, except that locations whose contents are zero are not punched out whenever more than two consecutive zeroes are detected.

ADR$J — This command punches a 2-word block that causes a transfer to address ADR after the preceding program has been loaded from paper tape. If ADR is not present, a JRST 4, DDT is punched as the first word.

The following succession of steps will punch a program on paper tape ready to be used as an independent entity.

a. $L

b. 5000<20000 (TAPE)[2]

c. 6000$J (Transfer to address 6000 after program is loaded.)

---

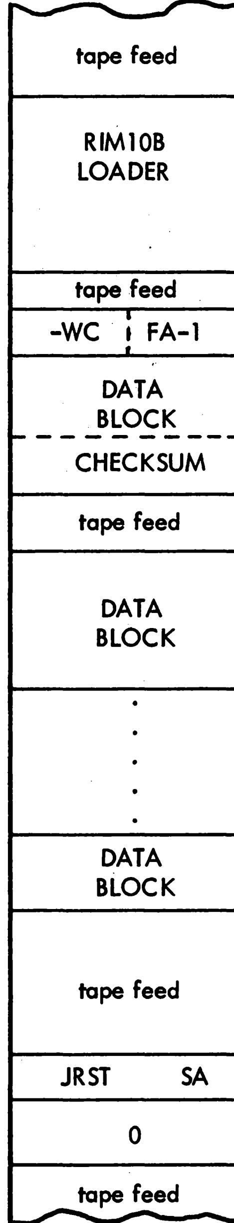
[1] The paper tape functions are not available in the timesharing user mode version of DDT.
[2] (TAPE) is a single control key on the Teletype, and is identical to ↑R (control-R).

Typed in:

$L

FIRST ADDRESS<
LAST ADDRESS(TAPE)

SA$J



Beginning of Tape

Checksum includes pointer word
WC = word count

transfer block
SA = starting address

Figure 6-1 RIM10B Block Format

APPENDIX A
SUMMARY OF DDT FUNCTIONS

A.1 TYPE-OUT MODES

The following are used to set the type-out mode:

	Type	Sample Output(s)
Symbolic instructions	$S	ADD 4, TAG+1 ADD 4, 4002
Numeric, in current radix	$C	69. 105
Floating point	$F	0.125E-3
7-bit ASCII text	$T	PQRST
SIXBIT text	$6T	TSRQPO
RADIX50	$5T	4 DDTEND
Halfwords, two addresses	$H	4002,,4005 X+1,,X+4
Bytes (of n bits each)	$NO	$8O COULD YIELD 0,14,237,123,0

A.2 ADDRESS MODES

The following are used to set the address made for typeout of symbolic instructions and halfwords (see examples above):

Relative to symbolic address	$R	TAG+1
Absolute numeric address	$A	4005

A.3 RADIX CHANGE

The following is used to change the radix of numeric type-outs

to n (for n≥2): $NR $2R COULD YIELD
110101100000001000000000000011100101100

A.4 PREVAILING VS. TEMPORARY MODES

The following are used in prevailing vs. temporary modes:

To set a temporary type-out or address mode or a temporary radix as shown in the commands above, type	$	$C $10R

	Type	Sample Output(s)
To set a prevailing type-out or address mode on a prevailing radix, in the commands above, substitute	$$	$$C $$10R
To terminate temporary modes and revert to prevailing modes, type a carriage return	↵	
Initial prevailing (and temporary) modes are	$$S $$R $$8R	

A.5 STORAGE WORDS

The following are used to examine storage words:

To open and examine the contents of any address in current type-out mode	adr/	LOC/	254020,,DDTEND
To open a word, but inhibit the type out of contents	adr!	LOC!	
To open and examine a word as a number in the current radix	adr[	LOC[	254020,,3454
To open and examine a word as a symbolic instruction	adr]	LOC]	JRST @DDTEND
To retype the last quantity typed (particularly used after changing the current type-out mode)	;	$6O;	25,40,20,00,34,54
		$6T;	5%0 <L

A.6 RELATED STORAGE WORD

The following are used to examine related storage words:

To close the current open word (making any modification typed in) and to open the following related words, examining them in the current type-out mode:

To examine ADR+1	↓ (line feed)
To examine ADR-1	↑ (or backspace, on the Teletype Model 37)

	Type	Sample Output(s)
To examine the contents of the location specified by the address of the last quantity typed, and to set the location pointer to this address	⇥ (TAB)	
To examine the contents of address of last quantity typed, but not change the location pointer	\ (backslash)	
To close the currently open word, without opening a new word, and revert to permanent type-out modes	↵ (carriage return)	

A.7 ONE-TIME ONLY TYPEOUTS

The following typeouts occur only one time:

	Type
To repeat the last typeout as a number in the current radix	=
To repeat the last typeout as a symbolic instruction (the address part is determined by $A or $R)	←
To type out, in the current type-out mode, the contents of the location specified by the address in the open instruction word, and to open that location, but not move the location pointer	/
To type out, as a number, the contents of the location specified by the open instruction word and to open that location, but not move the location pointer	[
To type out, as a symbolic instruction, the contents of the location specified by the open instruction word, and to open that word, but not move the location pointer	]

A.8 TYPING IN

Current type-out modes do not affect typing in; instead, the following are performed:

To type in a symbolic instruction	ADD AC1,@DATE(17)
To type in half words, separate the left and right halves by two commas	402,,403
To type in octal values	1234
To type in a fixed-point decimal integer	99.

	Type	Sample Output(s)
To type in a floating-point number	101.11 77.0E+2	
To type in up to five 7-bit PDP-10 ASCII characters, left justified, delimited by any printing character	"/ABCDE/	(/ is delimiter)
To type in one PDP-10 ASCII character, right justified	"A$	($ must be ALTMODE)
To type in up to six SIXBIT characters, left justified, delimited by any printing character	$"ABCDEFGA	(A is delimiter)
To type in one SIXBIT character, right justified	$"Q$	($ must be ALTMODE)

A.9 SYMBOLS

The following are DDT symbols:

To permit reference to local symbols within a program titled name	name$:	MAIN.$:
To insert or redefine a symbol in the symbol table and give it the value n	n<symbol:	14<TABL3:
To insert or redefine a symbol in the symbol table, and give it a value equal to the location pointer (.)	symbol:	SYM:
To delete a symbol from the symbol table	symbol$$K	LPCT$$K
To kill a symbol for typeouts (but still permit it to be used for typing in)	symbol$K	TBITS$K
To perform $K on the last symbol typed out and then to retype the last quantity	$D	
To declare a symbol whose value is to be defined later	symbol#	JRST AJAX#
To type out a list of all undefined symbols (which were created by #)	?	

A.10 SPECIAL DDT SYMBOLS

The following are special DDT symbols:

To represent the address of the location pointer	. (point)	
To represent the last quantity typed	$Q	

	Type	Sample Output(s)
To represent the last quantity typed, halves reversed	$V	
To read and execute a command file	$Y	
To represent the indirect address bit	@	
To represent the address of the search mask register	$M	
To represent the address of the saved flags, etc. (see Appendix D)	$I	
To represent the pointers associated with the nth breakpoint	$nB	

A.11 ARITHMETIC OPERATORS

The following arithmetic operators are permitted in forming expressions:

Two's complement addition	+
Two's complement subtraction	-
Integer multiplication	*
Integer division (remainder discarded)	' (apostrophe)

A.12 FIELD DELIMITERS IN SYMBOLIC TYPE-INS

The following are field delimiters:

To delimit op-code name	one or more spaces	JRST SUBRTE
To delimit accumulator field	, (comma)	
To delimit two halfwords	left,,right	-6,,BEGIN-1
To delimit index register	( )	
To indicate indirect addressing	@	

A.13 BREAKPOINTS

The following are used for breakpoints:

To set a specific breakpoint n(1<n<8)	adr$nB	CAR$8B
To set the next unused breakpoint	adr$B	303$B
To set a breakpoint with automatic proceed	adr$$nB adr$$B	CAR$$8B 303$$B
To set a breakpoint which will automatically open and examine a specified address, x	x,,adr$nB x,,adr$B x,,adr$$nB x,,adr$$B	AC3,,Z+6$5B AC4,,ABLE$B AC3,,Z+6$$5B AC4,,ABLE$$B

	Type	Sample Output(s)
To remove a specific breakpoint	0$nB	0$8B
To remove all breakpoints	$B	$B
To check the status of breakpoint n	$nB/	
To proceed from a breakpoint	$P	$P
To set the proceed count and proceed	n$P	25$P
To proceed from a breakpoint and thereafter proceed automatically	$$P n$$P	$$P 25$$P

A.14 CONDITIONAL BREAKPOINTS

The following are used for conditional breakpoints:

To insert a conditional instruction (INST), or call a conditional routine, when breakpoint n is reached	$nB+1/ $2B+1/ 0	INST CAIE 3,100
If the conditional instruction does not cause a skip, the proceed counter is decremented and checked. If the proceed count ≤0, a break occurs		
If the conditional instruction or subroutine causes one skip, a break occurs.		
If the conditional instruction or subroutine causes two skips, execution of the program proceeds.		

A.15 STARTING THE PROGRAM

The following commands are used to start the program:

To start at the starting address in JOBSA	$G	$G
To start, or continue, at a specified address	adr$G	LOC$G
To execute an instruction	inst$X	JRST 2, @JOBOPC$X returns to program after ↑C and DDT commands

A.16 SEARCHING

The following commands are used for searching:

To set a lower limit (a), an upper limit (b), a word to be searched for (c), and search for that word	a<b>c$W	200<250>0$W

	Type	Sample Output(s)
To set limits and search for a not-word	a<b>c$N	351<731>0$N
To set limits and search for an effective address	a<b>c$E	401<471>LOC+6$E
To examine the mask used in searches (initially contains all ones)	$M/	$M/ -1
To insert another quantity n in the mask	n$M	777000777777$M

A.17 UNUSED FUNCTIONS

The following is unused:

$U

A.18 ZEROING MEMORY

The following are used for zeroing memory:

To zero memory, except DDT, locations 20-137, and the symbol table	$$Z
To zero memory locations FIRST through LAST inclusive	FIRST<LAST $$Z

A.19 SPECIAL CHARACTERS

The following special characters are used in DDT typeouts:

Breakpoint stops		
Break caused by conditional break instruction	>	
Break because proceed counter ≤ 0	>>	
Undefined symbol cannot be assembled	U	
Half-word type-outs	left,,right	401,,402
Unnormalized floating-point number	#1.234E+27	#1.234E+27
To indicate an integer is decimal. The decimal point is printed	$10R 77=63.	
Illegal command	?	
If all eight breakpoints have been assigned	?	
RUBOUT echo	XXX	

A.20 PAPER TAPE COMMANDS

The following commands are available only in EDDT:

	Type	Sample Output(s)
To punch a RIM10B loader	$L	
To punch checksummed data blocks where ADR1 is the first, and ADR2 is the last location of the data	ADR1 <ADR2 (TAPE)	
To punch data as above, except that more than two consecutive locations containing zeros are not punched.	ADR1 < ADR2 $ (TAPE) ((TAPE) is ↑R)	
To punch a one-word block to cause a transfer to adr after the preceding program has been loaded from paper tape	adr$J	

APPENDIX B
EXECUTIVE MODE DEBUGGING (EDDT)

A special version of DDT, called EDDT, is available for debugging programs in the executive mode of the PDP-10. EDDT also runs in user mode under the monitor and performs the same debugging functions as user-mode DDT. EDDT requires somewhat more memory space than DDT; therefore, it is normally used only with hardware diagnostics and the monitor. All of the paper tape commands are available in EDDT (those in DDT are marked by an asterisk in Chapter 5). The paper tape I/O routines in EDDT are optional at assembly time.

EDDT is used to debug monitor programs, diagnostic programs, and other executive (or privileged) programs. EDDT performs its own I/O on a Teletype and controls the Priority Interrupt system. It does <u>not</u> check JOBREL for boundary limits as DDT does.

In EDDT the symbol table pointer is in location 36 and the undefined-symbol table pointer is in location 32. If the NXM STOP switch is ON, the machine will hang up if nonexistent memory is referenced. If this happens, EDDT may be restarted by pressing START, or the CONTINUE switch may be pressed.

Stand-alone programs should initialize EDDT by placing the contents of .JBSYM (116) into location 36, and .JBUSY (117) into location 32.

The first address of EDDT is DDT; the last is DDTEND.

The $$Z command will not zero locations 20 through 37. (In the user mode version, $$Z does not zero locations 20 through 137. See Section 4.5.)

APPENDIX C
STORAGE MAP FOR USER MODE DDT

See Figure C-1. The permanent symbol table, which contains all PDP-10 instructions and monitor UUOs, is an integral part of DDT.

If the user's symbol table is overwritten DDT can still interpret all instructions and UUOs. It will not interpret I/O device mnemonics, internal $ symbols ($M, $I, $1B through $8B), DDT and DDTEND or the following:

JOV
JEN
HALT

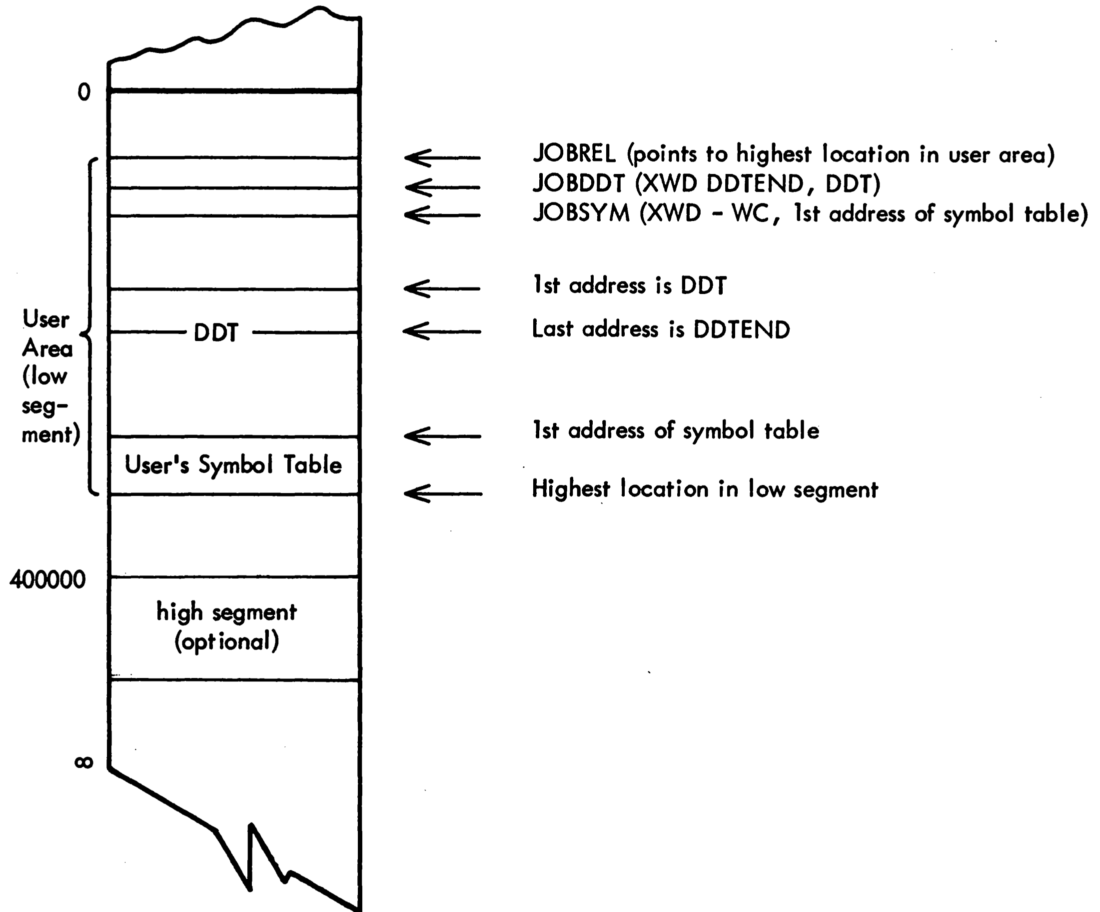


Figure C-1 Storage Map for User Mode DDT

APPENDIX D
OPERATING ENVIRONMENT

D.1 ENTERING AND LEAVING DDT

When control is transferred to DDT, the state of the machine is saved inside DDT:

a. The accumulators are saved.

b.[1] The status of the priority interrupt system (the result of a CONI PI, $I) is stored in the right half of register $I.

c. The central processor flags are saved in the left half of register $I.

d.[1] The PI channels are turned off (by a CONO PI, @$I+1) if they have a bit in register $I+1.

e.[1] The Teletype PI channel is saved in the right half of register $I+2. The Teletype buffer is saved in the left half of $I+2 but can never be restored. The character in the output buffer will have been typed on the Teletype.

f. If DDT was entered via ↑C↑C and the monitor DDT command, the old program counter word is saved in location JOBOPC.

When execution of a program is restarted, the following happens:

a. The accumulators are restored.

b.[1] Those PI channels which were on (when DDT was entered) and which have a bit equal to 1 in register $I+1 are turned on.

$$(C(\$I)_R \wedge C(\$I+1)_R) \vee 2000 \rightarrow \text{PI SYSTEM}$$

(logical AND (∧), logical OR (V))

c.[1] The Teletype PI channel is restored.

0 → TTI DONE → TTI BUSY → TTO BUSY

TTO done is set to 1 if either TTO busy or TTO done was on when DDT was entered. Otherwise 0 → TTO done.

d. The processor flags are restored from the left half of register $I.

e. To return to a program interrupted by ↑C, the user types:

`JRST 2, @ JOBOPC$X` to restore the PC and flags.

---

[1] Functions not available in the timesharing user mode.

D.2 LOADING AND SAVING DDT

Load and save DDT.SAV in 2K of core in the following manner:

	Instructions	Example[1]
(1)	Load DDT.	.R LOADER *DSK:DDT$ LOADER EXIT ↑C .ST 140
(2)	Enter DDT.	$$H JOBSYM/ -112,,5666
(3)	Type out, in halfword mode, the contents of JOBSYM.	
(4)	Open register 6, and put $(JOBSYM)_{RH}$ into left half of 6; put $((JOBSYM)_{RH}$ AND[2] 1777) + 2000 into right half of 6.	6! 5666,,3666
(5)	Perform a block transfer stopping at 3777.	BLT 6,3777$X
(6)	Open JOBSYM; leave the left half as is and change the right half to $((JOBSYM)_{RH}$ AND[2] 1777) + 2000.	JOBSYM/ -112,,5666 -112,,3666
(7)	Zero memory, except for DDT.	$$Z
(8)	Start over at 140 to initialize the new symbol table.	140$G
(9)	Open JOBSA and put DDTEND in the left half and DDT in the right half.	JOBSA! DDTEND,,DDT
(10)	Change back to symbol type-out mode.	$$S
(11)	Return to monitor.	↑C
(12)	Reduce core to 2K.	.CORE 2
(13)	Reenter DDT.	.DDT
(14)	Check that JOBREL is 2K.	JOBREL/ 3777
(15)	Return to monitor.	↑C
(16)	Save DDT.	.SAVE DSK DDT
(17)	Check start address.	.START ./ 3777

---

[1] ALTMODE is indicated by $.

[2] Logical AND.

D.3 EXPLANATION

The DDT saved file must be saved in 2K (minimum amount of core needed). Also, a starting address must be set up for DDT as location 140. To get DDT into 2K, the DDT symbol table must be moved down to the upper end of the first 2K of core. Any unused locations in DDT should be set to zero ($$Z) and JOBSYM should be set to the new location of the start of the DDT symbol table. Before saving the resulting file, a CORE 2 request should be given to the monitor to ensure that DDT is saved as a 2K core image.

UTILITIES

This section of the handbook includes documentation on the following software:

CREF	Version 47
FILCOM	Version 20
FUDGE2	Version 15
GLOB	Version 5A

These utilities are used by system programmers

1. To obtain cross-referenced listings for all operand-type symbols, user-defined symbols, and/or op codes and pseudo-op codes.

2. To compare two versions of a file and then output any differences.

3. To update files containing relocatable binary programs and manipulate programs within program files.

4. To obtain an alphabetical cross-reference listing of all global symbols encountered.

With the exception of the CREF writeup, the utilities have been reproduced from the DECsystem-10 Operating System Commands manual (DEC-10-MRDC-D).

CROSS-REFERENCE LISTING (CREF)

CREF produces a sequence-numbered assembly listing followed by one to three tables, one showing cross references for all operand-type symbols (labels, assignments, etc.), another showing cross references for all user-defined operators (macro calls, OPDEFs etc.), and another (if the proper switch is specified) showing the cross references for all op codes and pseudo-op codes (MOVE, XALL, etc.). A number sign (#) appears on the definition line of all symbols. The input to CREF is a modified assembly listing file created during a MACRO-10 assembly or FORTRAN IV compilation when the /C switch is specified in the command string.

CREF provides an invaluable aid for program debugging and modification.

1.0 REQUIREMENTS

Minimum Core:	2K pure, 1K impure
Additional Core:	Takes advantage of any additional core available, as necessary.
Equipment:	One input device (normally disk) which contains the modified assembly listing file; one output device (normally the line printer) for the listing.

2.0 INITIALIZATION

.R CREF↵	Loads the Cross-Reference Listing program into core.
*	The program is ready to receive a command.

NOTE

If CREF cannot initialize the terminal, it exits.

3.0 COMMANDS

3.1 Command Formats

a. output-dev:filename.ext[proj,prog]=input-dev:file1.ext[proj,prog],file2.ext,...

b. progname!

output-dev:filename.ext

The device on which the assembly listing and cross-reference tables are to be printed. If no output file name is specified, the default file name is the same as that specified for the first input file, but with the extension .LST. In such a case, the default device is LPT. However, if an output file name is specified, the default output device is DSK.

input-dev:filename.ext

The device on which the modified assembly listing was written during MACRO-10 assembly. DSK: is assumed if the device is not specified. When looking for the input file, CREF tries the following default extensions in the order listed: .CRF, .LST, .TMP, or a null extension. A missing input file name is given the name CREF. If the input file extension is .CRF or .LST and the /P switch is not included in the command string, the input file is deleted after the output file is successfully closed.

Multiple input files can be specified to be combined into a single CREF listing by separating the input files with commas. Switches affecting the entire listing (/K, /M, /O, and /S) must be specified before the terminator for the first input file. Switches affecting the positioning of an input file are specified with each file.

The ?CANNOT FIND FILE... message will be printed for each occurrence of a missing file. If the missing file is not part of a COMPIL-class command file (that is, if it was typed in directly), the command will be aborted, allowing the user to retype the command string. However, if the missing file is part of a COMPIL-class command file, processing will continue for the rest of the existing files in the command string. (Refer to section 4.0 of this document.) Note that if any file is in fact missing, no input file will be deleted.

[proj,prog]

The disk area on which the files are to be placed (output) or the disk area on which the source files reside (input). If omitted, the default is the user's disk area.

=

The output device and the input device are separated by an equal sign. If the equal sign is omitted, output defaults occur as described above. Any files specified by the user are for input.

progname!

The user can request CREF to run a system program by typing the program name followed by an exclamation point.

Examples of Commands:

.R MACRO	Load the MACRO-10 Assembler into core.
*PTP:,/C=DTA1:TXCALC THERE ARE NO ERRORS PROGRAM BREAK IS 003771	Assemble the program TXCALC from DTA1; writes the object program coding on the paper tape punch; writes a modified assembly listing on DSK: (assumed) and assigns it the filename CREF.LST.
7K CORE *↑C	Return to the monitor.
.R CREF	Load CREF into core.
*	Select the default assumptions of: output-dev: LPT: input-dev: DSK: input filename.ext CREF.CRF (.LST,.TMP) output filename.ext CREF.LST Equivalent to the command string LPT:CREF.LST=DSK:CREF.CRF
*↑C .	Return to the monitor.
.R CREF *OUTFIL=FIL1,FIL2,FIL3	Make single merged cross-reference file for three program files.
*LINK! *FIL1,FIL2,FIL3/G LINK: LOADING EXIT .	Run LINK10.

3.2 Switches

Switches are used to specify such options as magnetic tape control and list selection. All switches are preceded by a slash (/).

Examples of Switches:

.R CREF	Load CREF into core.
*/M=MTA1:/W	Rewind MTA1 and process the first file, listing only the cross references for operand-type symbols (labels, assignments, etc.).
*DTA5:SAVE1/Z=	Process the file named CREF.LST in the user's area of disk; write the program listing and operand-type cross references on DTA5 and call the file SAVE1.
*↑C	Return to monitor.

CREF Switch Options

Switch	Meaning
A	Advance magnetic tape reel by one file. /A may be repeated.
B	Backspace magnetic tape reel by one file. /B may be repeated.
H	Print help for running CREF.
K	Kill listing of references to basic symbols (labels, assignments, etc.).
M	Suppress listing of references to user-defined operators (Macro calls, OPDEFs, etc.).
O	Allow listing of references to machine and pseudo-operation codes (MOVE, XALL, etc.).
P	Preserve an input file with the extension .CRF or .LST, which is normally deleted.
R	Request the line number at which the listing is to Restart. CREF prints: RESTART LISTING AT LINE: at which time the user types the line number followed by a carriage return. (Such action might be necessary if the line printer ran out of paper, or jammed, etc.)
S	Suppress program listing (list only the selected tables).
T	Skip to logical end of magnetic Tape.
W	ReWind magnetic tape.
Z	Zero the DECtape directory (DECtape must be output only).

4.0 MONITOR COMMANDS

CREF-format listing files generated by COMPILE, LOAD, EXECUTE, and DEBUG commands (using the /CREF switch) can be printed on the line printer by typing

.CREF ↵

The CREF command will print out all listing files that are specified in the COMPIL-class command file, nnnCRE.TMP (where nnn is the user's job number). It will also transfer control to a system program if its name is present in the form "progname!". After completion of this operation, nnnCRE.TMP is deleted to prevent the listing files from being listed again by the next CREF command.

The CREF files may also be listed by an R CREF command and a response of "filename" to each asterisk (*) typed by CREF. It is important to note that, if the user uses the R CREF command to list files created by the monitor's COMPIL-class commands, the names of the files to be listed must be typed in response to the asterisks.

5.0 DIAGNOSTIC MESSAGES

CREF Diagnostic Messages

Message	Meaning
?dev NOT AVAILABLE	Device is assigned to another job.
?CANNOT ENTER FILE, n dev: file.ext	DTA or DSK directory is full; file cannot be entered; n indicates the cause of the failure and is obtained from the ENTER directory block.
?CANNOT FIND FILE, n dev: file.ext	The file cannot be found on the device specified; n indicates the cause of the failure and is obtained from the LOOKUP directory block.
?COMMAND ERROR--TYPE /H FOR HELP	Error in last command string entered. 1. Device name, file name, or extension consisted of non-alphanumeric characters. 2. The project-programmer number was not in standard format (i.e., it was not octal numbers in the form [proj, prog]). 3. An undefined switch was specified, switches in parentheses were not separated by commas, or the closing parenthesis was missing. 4. The /Z switch was used on the input side of the command string.
?COMMAND FILE INPUT ERROR, n dev:file.ext	Disk data error while reading nnnCRE.TMP; n is a six-digit (or less) octal number representing the file status word returned from the GETSTS UUO.
?DATA ERROR DEVICE dev:	READ or WRITE error.
IMPROPER INPUT DATA, CONTINUING	Input data not in CREF format. Output listing continues.
?INPUT BUFFERS TOO BIG	The monitor set up input buffers longer than 203_8. This is not a user error and hopefully will never occur.
?INPUT ERROR, n dev:file.ext	READ error has occurred on the device.
?INSUFFICIENT CORE	Additional core is required for execution but none is available from monitor.
?OUTPUT ERROR, n dev:file.ext	WRITE error has occurred on the device.

Function

The FILCOM program is used to compare two versions of a file and to output any differences. Generally, this comparison is line by line for ASCII files or word by word for binary files. FILCOM determines the type of comparison to use by examining either the switches specified in the command string or the extensions of the files. Switches always take precedence over file extensions.

Command Format

.R FILCOM ↵
*output dev:file.ext [directory] = input dev_1:file.ext [directory], input dev_2:file.ext [directory]

output dev:	= the device on which the differences are to be output.
input dev:	= the device on which an input file resides.

Defaults

1. If the entire output specification is omitted, the output device is assumed to be TTY. However, the equal sign must be given to separate the input and output specifications of the command string.
2. If an output filename is specified, the default output device is DSK.
3. If the output filename is omitted, the second input filename is used, unless it is null. In this case, the filename FILCOM is used.
4. If the output extension is omitted, .SCM is used on a source compare and .BCM is used on a binary compare.
5. If the [directory] is omitted (input or output side), the user's default directory is assumed.
6. If an input device is omitted, it is assumed to be DSK.
7. If the filename and/or extension of the second input file is omitted, it is taken from the first input file.
8. A dot following the filename of the second input is necessary to explicitly indicate a null extension, if the extension of the first input file is not null. For example, to compare FILE.MAC and FILE. (i.e., with null extension), use the following command string:

   .R FILCOM ↵
   *=FILE.MAC,FILE. ↵

Command Format (cont)

9. The second input file specification cannot be null unless a binary compare is being performed. In a binary compare, if the first input file is not followed by a comma and a second input file descriptor, the input file is compared to a zero file and is output in its entirety. This gives the user a method of listing a binary file. Refer to Example 4.

Switches

The following switches can appear in the command string, depending on whether a source compare or a binary source compare is being performed.

Binary Compare

/H	Type list of switches available (help text from device SYS:).
/nL	Specify the lower limit for a partial binary compare (n is an octal number). This switch, when used with the /nU switch, allows a binary file to be compared only within the specified limits.
/Q	When the files are different, print the message ?FILES ARE DIFFERENT, but do not list the differences. This switch is useful when BATCH control files want to test for differences but do not want the log file filled with these differences.
/nU	Specify the upper limit for a partial binary compare (n is an octal number). This switch, when used with the /nL switch, allows a binary file to be compared only within the specified limits.
/W	Compare files in binary mode without expanding the files first (refer to Appendix D). This switch is used to compare two binary files with ASCII extensions.
/X	Expand SAV files before comparing them in binary mode. This action removes differences resulting from zero compression (refer to Appendix D).

Source Compare

/A	Compare files in ASCII mode. This switch is used to force a source compare on two ASCII files.
/B	Compare blank lines. Without this switch, blank lines are ignored.
/C	Ignore comments (all text on a line following a semicolon) and spacing (spaces and tabs). This switch does not cause a line consisting entirely of a comment to become a blank line, which is normally ignored.
/H	Type list of switches available (help text from device SYS:).

(continued on next page)

Command Format (cont)

Source Compare (cont)

/nL Specify the number of lines that determine a match (n is an octal number). A match means that n successive lines in each input file have been found identical. When a match is found, all differences occurring before the match and after the previous match are output. In addition, the first line of the current match is output after the differences to aid in locating the place within each file at which the differences occurred. The default value for n is 3.

/Q Print the message ?FILES ARE DIFFERENT, when the files are different, but do not list the differences.

/S Ignore spaces and tabs.

/U Compare in update mode. This means that the output file consists of the second input file with vertical bars (or back slashes for 64-character printers) next to the lines that differ from the first input file. This feature is useful when updating a document because the changes made to the latest edition are flagged with change bars in the left margin. The latest edition of the document is the second input file.

If switches are not specified in the command string, the files are compared in the mode implied by the extension. The following extensions are recognized as binary and cause a binary compare if one or both of the input files have one of the extensions.

.BAC	.HGH	.RMT
.BIN	.LOW	.RTB
.BUG	.MSB	.SAV
.CAL	.OVR	.SFD
.CHN	.QUE	.SHR
.DAE	.QUF	.SVE
.DCR	.REL	.SYS
.DMP	.RIM	.UFD
		.XPN

Binary files are compared word by word starting at word 0 except for the following two cases:

1. Files with extensions .SHR and .HGH are assumed to be high segment files. Since the word count starts at 400000, upper and lower limits, if used, must be greater than (or equal to in the case of the lower limit) 400000.
2. Files with extensions .SAV, .LOW, and .SVE are assumed to be compressed core image files and are expanded before comparing.

Command Format (cont)

Conflicts are resolved by switches or defaults. If a conflict arises in the absence of switches, the files are assumed to be ordinary binary files.

Output

In most cases, headers consisting of the device, filename, extension, and creation date of each input file are listed before the differences are output. However, headers do not appear on output from the /U switch (update mode on source compare).

Source compare output - After the headers are listed, the following notation appears in the left column of the output

n)m

where

n is the number of the input file, and
m is the page number of the input file (see examples).

The right column lists the differences occurring between matches in the input files. Following the list of differences, a line identical to each file is output for reference purposes.

The output from the /U switch differs from the above-described output in that the output file created is the second input file with vertical bars in the left column next to the lines that are different from the first input file.

Binary compare output - When a difference is encountered between the two input files, a line in the following format appears on the output device:

octal loc. first file-word second file-word XOR of both words

If the exclusive OR (XOR) of the two words differs only in the right half, the third word output is the absolute value of the difference of the two right halves. This usually indicates an address that changed.

If one input file is shorter than the other, after the end of file is encountered on the shorter file, the remainder of the longer file is output.

Characteristics

The R FILCOM command:

Places the terminal in user mode.
Runs the FILCOM program, thereby destroying the user's core image.

Associated Messages

?2K CORE NEEDED AND NOT AVAILABLE

FILCOM needs 2K of core to initialize I/O devices and this core is not available from the monitor.

?BUFFER CAPACITY EXCEEDED AND NO CORE AVAILABLE

The buffer is not large enough to handle the number of lines required for looking ahead for matches, and additional core is not available.

?COMMAND ERROR

One of the following errors occured in the last command string typed.

1) There is no separator (←or =) between the output and input specifications.
2) The input specification is completely null.
3) The two input files are not separated by a comma.
4) A file descriptor consists of characters other than alphanumeric characters.
5) FILCOM does not recognize the specified switch.
6) The project-programmer number is not in standard format, i.e., [proj,prog].
7) The value of the specified switch is not octal.
8) The first input file is followed by a comma but the second input file is null.

?DEVICE dev: NOT AVAILABLE

Device is assigned to another job or does not exist.

?FILE n NOT IN SAV FORMAT

The user indicated via the /X switch that the file is to be expanded but the specified file is not in compressed file format. N is either 1 or 2 indicating the first file or the second file.

Associated Messages (Cont)

?FILE n READ ERROR

An error has occurred on either the first or second input device.

%FILES ARE DIFFERENT

The two input files specified in the command string are different (i.e., the two files are not two versions of the same file but are two different files).

?INPUT ERROR - file.ext FILE NOT FOUND

The specified file could not be found on the input device.

NO DIFFERENCES ENCOUNTERED

No differences were found between the two input files.

?OUTPUT DEVICE ERROR

An error has occurred on the output device.

?OUTPUT INITIALIZATION ERROR

The output device cannot be initialized for one of the following reasons:

1) The device does not exist or is assigned to another job.
2) The device is not an output device.
3) The file cannot be placed on the output device.

Examples

1. The user has the following two ASCII files on disk:

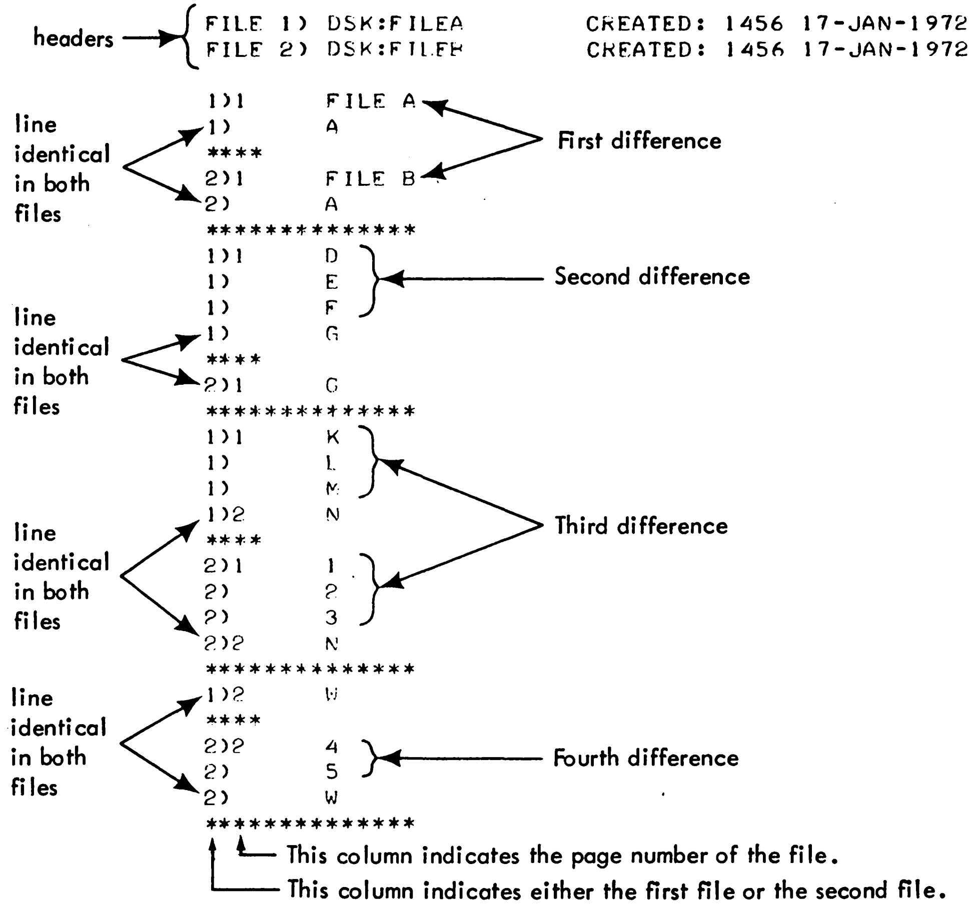
First File	Second File	
FILE A	FILE B	page 1
A	A	
B	B	
C	C	
D	G	
E	H	
F	I	
G	J	
H	1	
I	2	
J	3	
K		
L		
M		

First File	Second File	
N	N	page 2
O	O	
P	P	
Q	Q	
R	R	
S	S	
T	T	
U	U	
V	V	
W	4	
X	5	
Y	W	
Z	X	
	Y	
	Z	

To compare the two files and output the differences on the terminal, the following sequence is used:

`.R FILCOM`	Run the FILCOM program.
`*=FILEA,FILEB`	Compare the two files on disk and output the differences on the terminal. By default, three consecutive identical lines determine a match.

Examples (cont)



Examples (cont)

To compare the two files and output the differences on the line printer, the following commands are used. Note that in this example the number of successive lines that determines a match has been set to 4 with the /4L switch.

```
.R FILCOM
*LPT:/4L=FILEA,FILEB

        FILE 1) DSK:FILEA           CREATED: 1456 17-JAN-1972
        FILE 2) DSK:FILEB           CREATED: 1456 17-JAN-1972

        1)1     FILE A
        1)      A
        1)      B
        1)      C
        1)      D
        1)      E
        1)      F
        1)      G
        ****
        2)1     FILE B
        2)      A
        2)      B
        2)      C
        2)      G
        **************
        1)1     K
        1)      L
        1)      M
        1)2     N
        ****
        2)1     1
        2)      2
        2)      3
        2)2     N
        **************
        1)2     W
        ****
        2)2     4
        2)      5
        2)      W
        **************
```

These lines are listed as being different because the /4L switch specifies that 4 consecutive lines must be found identical in the two files before they are considered as a match.

(continued on next page)

Examples (cont)

To compare the two files so that the second input file is output with vertical bars in the left column next to the lines that differ from the first input file, use the following command sequence.

```
.R FILCOM
*LPT:/U=FILEA,FILEB

|       FILE B

        A
        B
        C
|       G
        H
        I
        J
|       1
|       2
|       3
        N
        O
        P
        Q
        R
        S
        T
        U
        V
|       4
|       5
        W
        X
        Y
        Z
```

The lines with vertical bars indicate the differences between the two files.

The lines with vertical bars indicate the differences between the two files.

2. To compare two binary files on the disk and output the differences on the terminal, use the following command sequence.

```
.R FILCOM
*TTY:←DSK:DIAL.REL,DIAL2
FILE 1) DSK:DIAL.REL     CREATED: 0000 23-DEC-1971
FILE 2) DSK:DIAL2.REL    CREATED: 0000 12-AUG-1971

000000  000004 000001   000004 000060          000057
000002  000000 054716   000311 372712   000311 326004
000003  000006 000001   017573 510354   017575 510355
000004  000000 000000   017573 513216   017573 513216
```

(continued on next page)

Examples (cont)

3. To compare two high segment files, the command sequence below is used. Note that the locations begin at 400000.

```
.R FILCOM
*TTY:←SYS:TABLE.SHR, TABLE.SHR
FILE 1) SYS:TABLE.SHR   CREATED: 2020 24-JAN-1972
FILE 2) DSK:TABLE.SHR   CREATED: 1829 30-NOV-1971

400000  001611 400010   001630 407157   000021 007147
400003  006675 000000   015024 407670   013651 407670
400004  005600 000070   004700 000113   001100 000163
400005  545741 444562   554143 625700   011602 261262
400010  634000 000000   260740 403516   454740 403516
400011  474000 000000   200000 414036   674000 414036
400012  402000 000156   202000 000720   600000 000676
400013  200040 406354   201000 000472   001040 406726
```

4. To list a binary file, use the following command sequence.

```
.R FILCOM
*TTY:←SYS:DOT.REL
000000  000004 000001
000001  000000 000000
000002  000000 054716
000003  000006 000001
000004  000000 000000
000005  000007 517716
000006  000001 000002
000007  000000 000000
  .
  .
  .
```

Note that the following sequence will not work because of the terminating comma.

```
*TTY:←SYS:DOT.REL,

?COMMAND ERROR
```

Examples (cont)

5. To compare two binary files between locations 150–160 (octal).

```
.R FILCOM
*TTY:/150L/160U←SYS:SYSTAT.SAV,SYS:SYSDPY.SAV
FILE 1) SYS:SYSTAT.SAV  CREATED: 0818 30-NOV-1971
FILE 2) SYS:SYSDPY.SAV  CREATED: 1642 29-NOV-1971

000150  200400 000137   200740 003217    000340 003320
000151  260740 004226   404500 004242    664240 000064
000152  260740 004253   661500 002000    401240 006253
000153  200040 005011   260740 002723    060700 007732
000154  260740 004063   200040 004243    060700 000220
000155  201041 777777   202040 003241    003001 774536
000156  047040 000042   200040 004241    247000 004203
000157  254000 000174   251040 004142    005040 004036
000160  476000 006774   211040 000144    667040 006630
```

6. To compare two .SAV files. Note that the files are expanded before the comparison.

```
.R FILCOM
*TTY:←SYS:TRY1.SAV,SYS:TRY.SAV
FILE 1) SYS:TRY1.SAV    CREATED: 2043 05-JAN-1972
FILE 2) SYS:TRY.SAV     CREATED: 0818 30-NOV-1971

000114  004000 000140   000000 000000    004000 000140
000116  777536 005536   000000 000000    777536 005536
000117  000000 005536   000000 000000           005536
000120  006000 000140   007222 000140    001222 000000
000121  000000 006000   000000 007222           001222
000130  010000 000005   000000 000000    010000 000005
000133  003727 005777   006643 007777    005164 002000
000137  003400 000070   046700 000004    045300 000074
000140  264000 001454   047000 000000    223000 001454
000141  260040 001773   200040 005075    060000 004706
000142  201240 001447   402000 006644    603240 007203
000143  542240 001634   251040 007221    713200 006415
000144  260040 002774   403000 000015    663040 002761
000145  621000 000010   476000 006715    257000 006705
000146  200240 003504   200740 006606    000500 005302
000147  251240 000012   051140 005076    200300 005064
000150  402000 003613   200400 000137    602400 003724
000151  201040 003730   260740 004226    061700 007516
000152  200260 003632   260740 004253    060520 007461
000153  321240 000164   200040 005011    121200 005175
```

Function

The FUDGE2 program is used to update files containing one or more reloctable binary modules and to manipulate the individual modules within these files. Relocatable binary modules are output by MACRO-10, FORTRAN-10, COBOL, ALGOL, and BLISS-10. A module can be a complete program or only a set of subroutines. One reason for collecting a group of relocatable modules into one file is to enable LINK-10 or LOADER to use the file as a library (refer to the LINK-10 or LOADER documentation). Three files are used in the updating process:

1. A master file containing the file to be updated.
2. A transaction file containing the modules to be used when updating.
3. An output file containing the updated file.

All three files can be on the same device if the device is DSK. The two input files can be on the same DECtape.

The desired function of FUDGE2 is specified by a command code at the end of the command string. Only one command code can be specified in each command string. Switches can also be used to position a magnetic tape and to zero a DECtape directory (zeroing a DECtape directory is equivalent to deleting all the files on the tape).

WARNING

For execution to occur, the command string must be terminated with an ALTmode, represented in this manual by a dollar sign ($), instead of the usual carriage return-line feed.

Command Format

```
.R FUDGE2↵
*output dev:file.ext=master dev:file.ext <modules>, transaction dev:file.ext <modules>
    (command)$
```

output dev: = the device on which the updated file is written. If omitted, DSK is assumed.

master dev: =the device containing the file to be updated. If omitted, the default is DSK. A comma is used to separate the master file and the transaction file.

transaction dev: = the device containing the modules to be used in the updating process. When more than one file is transferred from magnetic tape or paper tape, a colon must follow the device name for each file. For example,

MTA: : : Transfer 3 files

If the device is omitted, DSK is assumed.

file. ext =the filename and extension of each file. Filenames must be specified for directory devices, but the extension can be omitted. If the extension is not given, it is assumed to be .REL unless the /L switch appears in the command string. In this case, the output extension .LST is assumed.

Command Format (cont)

file.ext (cont)

Project-programmer numbers appearing after a filename apply to that file only. If the project-programmer number appears before the filename, it applies to all subsequent files until another device is specified.

The protection code of the master file is given to the output file.

The asterisk convention can be used with the input files (refer to Paragraph 1.4.2.4).

<modules>

=Names of modules (on DSK or DTA only) to be used in the updating process. They are grouped within angle brackets in the same order as they appear in the file and are separated by commas. When manipulating all the modules within a file, only the filename need be specified. Module names cannot appear for the output file.

(command)

=Code for the function to be performed. This code can be either preceded by a slash or enclosed in parentheses and must appear at the end of the command string. Each command results in the updated file being output to the output device. The command codes are as follows:

A Append the specified modules in the transaction file(s) to the master file.

C Compress the master file by deleting local symbols. These symbols are included in relocatable binary modules primarily because of their usefulness in debugging procedures. Large libraries of debugged routines, such as LIBOL, frequently have the local symbols deleted in order to save disk space and reduce the amount of I/O required during the loading process.

D Delete the specified modules from the master file.

E Extract the specified files and/or modules from the input files. The entire file is extracted if module names are not specified.

H Type the commands and switches available (help text from device SYS:).

I Insert modules from the specified transaction files into the master file. The modules from the transaction files are inserted immediately before the specified modules in the master file. A comma is used to separate the transaction files.

(continued on next page)

Command Format (cont)

(command) (cont)	L	List the names and lengths of all relocatable modules within a file. The length is in one of two forms: low segment break, high segment break or program break, absolute break The length of FORTRAN modules is not output. The default filename for spooled output is the name of the master file.
	R	Replace the specified modules in the master file with the specified modules in the transaction file. The number of replacing modules must be the same as the number of modules to replace.
	S	List all the entry points within a module. These entry points are listed across the page. The default filename for spooled output is the name of the master file.
	X	Write index blocks into a library file on DECtape or disk. Indexes cannot be written on magnetic tape. Index blocks are used in a direct access library search (refer to the LOADER documentation). This command implies a C command.

The method of numbering the blocks within a file is different on DECtape and disk. This can cause problems with indexed library files that are created on one device and loaded from the other. The index in an indexed library contains the name of each module in the library along with the block number within the library file of the beginning of that module. On disk, the blocks of a file are numbered relative to the beginning of the file; thus, an index references the same blocks properly no matter where the file is placed on the disk. However, on a DECtape, the block numbers are established relative to the beginning of the tape. Therefore, the area of the tape on which the file resides determines the block numbers that will be used in the index. When the LOADER references an indexed library on a device different from the one on which it was created, the block numbers in the index may no longer point to the correct location within the library. This problem can also arise when loading an indexed file that was created at one location on a DECtape and then was transferred to a different location on that tape or to another tape. To transfer indexed files to other devices and then to load them from that device, the index blocks should be deleted before transferring and recreated on the new device. Any FUDGE2 command which generates a new binary file deletes the index blocks and causes a warning message to be output. The /X switch writes the index blocks.

NOTE

An indexed library created on the disk will work properly no matter how many times it is transferred to and from other devices, such as DECtape and magnetic tape, as long as the library is restored to the disk for use by the LOADER or LINK-10.

(continued on next page)

Command Format (cont)

Comments are included on the FUDGE2 command string by preceding the comment with a semicolon. All characters after the semicolon, except for the ALTmode, are ignored until the next line feed, vertical tab, or form feed character is read.

File directories can be manipulated and magnetic tapes positioned by including switches in the command string. These switches can appear anywhere in the command string and are preceded by a slash or enclosed in parentheses. The following switches are available:

/B Backspace a magnetic tape one file.

/K Advance a magnetic tape one file.

/T Skip to the logical end of tape on a magnetic tape.

/W Rewind a magnetic tape.

/Z Clear the directory of the output DECtape.

Characteristics

The R FUDGE2 command:

Places the terminal in user mode.
Runs the FUDGE2 program, thereby destroying the user's core image.

Associated Messages

?CANNOT DO I/O AS REQUESTED

Input (or output) cannot be performed on one of the devices specified for input (output). For example, input may have been requested for a device that can only do output.

?COMMAND SWITCH REQUIRED

The given command string requires a FUDGE2 command code.

?DEVICE ERROR ON OUTPUT DEVICE

A write error has occurred on the output file.

?DIRECTORY FULL ON OUTPUT DEVICE

There is no room in the file directory on the output device to add the updated file (non-disk devices only).

Associated Messages (Cont)

?ENTER FAILURE n

The output filename is null; n is the error code for an illegal filename (non-disk devices only).

?ENTER FAILURE

The ENTER to write the disk file failed. This message is followed by a line explaining the reason for failure.

?ENTRY BLOCK TOO LARGE, PROGRAM name

The entry block of the named program is too large for the FUDGE2 entry table, which allows for 100 entry names. FUDGE2 can be reassembled with a larger table.

?FUDGE2 SYNTAX ERROR

An illegal command string was entered; for example, the left arrow was omitted or a program name was specified for the output file.

?ILLEGAL BLOCK TYPE dev:file.ext

The block type used is not in the range 0-77.

?ILLEGAL DATA MODE FOR dev

The data mode specified for a device in the user's program is illegal, such as dump mode for the terminal.

?(0)ILLEGAL FILENAME

A filename of zero was specified.

?INPUT ERROR ON DEVICE dev: STATUS (nnnnnn)

A data or device error occurred on input.

?x IS AN ILLEGAL {CHARACTER / SWITCH}

An illegal character or switch was encountered in the command string.

?LOOKUP FAILURE

The LOOKUP to read the disk file failed. This message is followed by a line explaining the reason for failure.

Associated Messages (Cont)

?dev:file.ext <> NO PROGRAM NAME SPECIFIED

The switch (/D or /R) used in the command string requires that a program name be given.

?dev NOT AVAILABLE

The specified device does not exist or is assigned to another user.

?NOT ENOUGH ARGUMENTS

An insufficient number of files of one type has been specified.

?dev file.ext program NOT FOUND

The file or the program was not found on the device or in the file specified. If a program name is printed, this message may indicate that the program names in the command string appear in a sequence different from their sequence within the file. Therefore, the program may actually exist but was missed because of the incorrect sequence in the command string.

?PROGRAM ERROR WHILE RESETTING MASTER DEVICE

FUDGE2 cannot find the master device or cannot find the program on the master device.

?TOO MANY FILENAMES OR PROGRAM NAMES

More than 40 program names or filenames were specified in the command string. The user should separate the job into several segments.

?TRANSMISSION ERROR ON INPUT DEVICE dev

A transmission error has occurred while reading data from the specified device.

?UNEQUAL NUMBER OF MASTER AND TRANSACTION PROGRAMS

On a replace request, the number of master programs (or files) does not equal the number of transaction programs (or files).

WARNING NO INDEX ON OUTPUT FILE-CONTINUING

The user has changed the structure of the indexed library file when deleting, appending, or inserting, thereby invalidating the index. The index has been removed from the new file. Reindexing is required.

Examples

```
.R FUDGE2
*LPT:=DTA1:LIB40(L)$
```

List all relocatable modules from the file LIB40 .REL, located on DTA1, on the line printer.

```
*DSK:LIB4BB=DTA2:LIB4AA <EXP.3,EXP.3C>,
DTA1:F4<EXP.3A,EXP.3B>(R)$
```

Replace modules EXP.3 and EXP.3C located in file LIB4AA.REL on DTA2, with modules EXP.3A and EXP.3B in File F4.REL on DTA1; write out the new LIB4AA file on disk and call it LIB4BB.REL.

```
*DTA1:NFILE=DSK:MFILE<M1,M2,M3,M4>,
DTA3:TFILEA<TA1,TA2>,
DTA4:TFILEB<TB1,TB2>/I$
```

Insert into MFILE the modules TA1 and TA2 from TFILEA, and TB1 and TB2 from TFILEB. Create NFILE with the following order:

TA1, M1, TA2, M2, TB1, M3, TB2, M4

Insertion is on a one-to-one basis. If there are more modules to be inserted than specified modules before which they are to be inserted, the extra files are ignored.

```
*DTA1:NFILE=DSK:MFILE<M1,M2,M3,M4>,
DTA3:TFILEA,
DTA4:TFILEB/I$
```

However, in this example (where TFILEA.REL and TFILEB.REL contain the modules TA1 and TA2 and TB1 and TB2, respectively) create an NFILE.REL with the following order:

TA1,TA2,M1,TB1,TB2,M2,M3,M4

```
*DTA2:TESTA=MTA1:(WK),MTA2: :(ZA)$
```

Clear the directory of DTA2; rewind MTA1 and advance the tape one file; append the first two program files from MTA2 to the second file on MTA1 and write out the resultant file on DTA2, calling it TESTA.REL.

```
*OUTPUT=LIBARY,DTA1:LIBARY<FILEY,FILEZ>/A$
```

Append the modules FILEY and FILEZ contained in the file LIBARY.REL on DTA1 to the end of the file LIBARY.REL on disk. Write the new file on disk and call it OUTPUT.REL.

```
*NEWFIL=OLDFIL<TEST,SUBTRC,MULTI>,BASFIL<PROG,
ROUTIN,ANSWER>,SUBFIL<MATH>(E)$
```

Extract the specified modules from the files OLDFIL, BASFIL, and SUBFIL and create a new output file called NEWFIL. The order of the modules in NEWFIL is as follows: TEST, SUBTRC, MULTI, PROG, ROUTIN, ANSWER, MATH.

(continued on next page)

Examples (cont)

```
*NEWF40=DTA2:OLDF40<SUBTLE,DATFIL,ROUTINE>/D$
```

Delete the modules SUBTLE, DATFIL, and ROUTNE from the file OLDF40.REL on DTA2 and create a new output file NEWF40.REL on disk containing the remainder of file OLDF40.

```
*NOIDX.REL=IDX.REL(A)$
```

Delete index blocks from the file IDX.REL and write the remainder of the file on the output file NOIDX.REL. The Append command (A) generates a new binary file and therefore removes the index blocks.

```
*↑C
```

Return to the monitor.

Function

The GLOB program reads multiple binary program files and produces an alphabetical cross-referenced list of all the global symbols (symbols accessible to other programs) encountered. This program also searches files in library search mode, checking for globals, if the program file was loaded by the LOADER in library search mode (refer to the LOADER documentation).

The GLOB program has two phases of operation; the first phase is to scan the input files and build an internal symbol table, and the second, to produce output based on the symbol table. Because of these phases, the user can input commands to GLOB in one of two ways. The first way is to specify one command string containing both the output and input specifications. (This is the command string format most system programs accept.) The second is to separate the command string into a series of input commands and output commands.

Command Formats

1. R GLOB

   outdev:file.ext [directory] = input dev:file.ext [directory], file.ext,...,dev:file.ext [directory] ($)

2. R GLOB

   followed by one or more input commands in the form

   dev:file.ext [directory],file.ext [directory],...,dev:file.ext [directory],...↵

   and then one or more output commands in the form

   outdev:file.ext [directory] =($)

When the user separates his input to GLOB into input commands and output commands (Command Format #2), the input commands contain only input specifications and the output commands, only output specifications. Each output command causes a listing to be generated; any number of listings can be printed from the symbol table generated from the current input files as long as no input commands occur after the first output command. When an input command is encountered after output has been generated, the current symbol table is destroyed and a new one begun.

Defaults

1. If the device is omitted, it is assumed to be DSK. However, if the entire output specification is omitted, the output device is TTY.

(continued on next page)

Command Format (cont)

Defaults (cont)

2. If the output filename is omitted, it is the name of the last input file on the line (Command Format #1) or is GLOB if the line contains only output commands (Command Format #2). The input filenames are required.
3. If the output extension is omitted, .GLB is used. If the input extension is omitted, it is assumed to be .REL unless the null extension is explicitly specified by a dot following the filename.
4. If the project-programmer number [proj,prog] is omitted, the user's default directory is used.
5. An ALTmode terminates the command input and signals GLOB to output the cross-referenced listing. In other words, a listing is not output until GLOB encounters an ALTmode. The ALTmode appears at the end of the command string shown in Command Format #1 or at the end of each output command shown in Command Format #2.

Switches

Switches control the types of global listings to be output. Each switch can be preceded by a slash, or several switches can be enclosed in parentheses. Only the most recently specified switch (except for L, M, P, Q, and X, which are always in effect) is in effect at any given time. If no switches are specified, all global symbols are output. The following switches are available.

/A	Output all global symbols. This is the default if no switches are specified.
/E	List only erroneous (multiple defined or undefined) symbols.
/F	List nonrelocatable (fixed) symbols only.
/H	List the switches available (help text) from SYS:GLOB.HLP.
/L	Scan programs only if they contain globals previously defined and not yet satisfied (library search mode).
/M	Turn off library search mode scanning resulting from a /L switch.
/N	List only symbols which are never referenced.
/P	List all routines that define a symbol to have the same value. The routine that defines the symbol first is listed followed by a plus (+) sign. Subsequent routines that define the symbol are listed preceded by a plus sign.
/Q	Suppress the listing of subsequent definers that result from the /P switch.

(continued on next page)

Command Format (cont)

Switches (cont)

/R List only relocatable symbols.

/S List symbols with non-conflicting values that are defined in more than one program.

/X Do not print listing header when output device is not the terminal, and include listing header when it is the terminal. Without this switch, the header is printed on all devices except the terminal. The listing header is in the following format:

FLAGS SYMBOL OCTAL VALUE DEFINED IN REFERENCED IN

Symbols listed are in alphabetical order according to their ASCII code values. The octal value is followed by a prime (') if the symbol is relocatable. The value is then relative to the beginning of the program in which the symbol is defined. Flags preceding the symbol are shown below.

M Multiply defined symbol (all values are shown).

N Never referred to (i.e., was not declared external in any of the binary programs).

S Multiply specified symbol (i.e., defined in more than one program but with non-conflicting values). The name of the first program in which the symbol was encountered is followed by a plus sign.

U Undefined symbol.

Characteristics

The R GLOB command:

Places the terminal in user mode.
Runs the GLOB program, thereby destroying the user's core image.

Associated Messages

?COMMAND SYNTAX ERROR
TYPE/H FOR HELP

An illegal command string was entered.

?DESTINATION DEVICE ERROR

An I/O error occurred on the output device.

Associated Messages (cont)

?ENTER ERROR n
?DIRECTORY FULL

No additional files can be added to the directory of the output device; n is the disk error code.

?ILLEGAL SWITCH

A non-recognizable switch was used in the command string.

?LOOKUP ERROR n
?file.ext FILE NOT FOUND

The named file cannot be found in the directory on the specified device.

?dev NOT AVAILABLE

The requested device does not exist or is assigned to another job.

?TABLE OVERFLOW - CORE UUO FAILED TRYING TO EXPAND TO xxx

The GLOB program requested additional core from the monitor, but none was available.

Examples

.R GLOB	Run the GLOB program.
*LPT:=MAIN,DTA2:SUB40,SUB50 $	All global symbols in the programs MAIN (on DSK), SUB40, and SUB50 (on DTA2) are listed on the line printer. Along with the symbol is listed its value, the program in which it is defined, all programs in which it is referenced, and any error flags.
*DTA4:BATCH.REL,DATA,DTA6:NUMBER.REL,CLASS	
*DSK:MATH.REL,LIBARY.	The programs to be scanned are BATCH.REL, DATA.REL on DTA4; NUMBER.REL, CLASS.REL on DTA6; and MATH.REL, LIBARY.null on DSK.
*LPT:=/F $	List only nonrelocatable symbols on the line printer.
*DSK:SYMBOL=/R $	List only relocatable symbols in the file named SYMBOL in the user's default directory.
*TTY:=/E $ U EXTSYM SUBRTE	Print all erroneous symbols on the terminal. EXTSYM is an undefined symbol appearing in the program SUBRTE.
*↑C	Return to monitor mode.

INDEX

A, 24, (SYSTEM REF.)
A+1, 24, (SYSTEM REF.)
Absolute address, 360, (MONITOR CALLS)
Absolute address mode, 205, 277, (MACRO)
Absolute binary programs, 288, (MACRO)
Absolute symbols, 695, (LINK-10)
AC, 24, (SYSTEM REF.)
Access block, 559, 642, (MONITOR CALLS)
Access protection, 554, (MONITOR CALLS)
Access time, 13, (SYSTEM REF.)
Accumulators, 360, (MONITOR CALLS)
Accumulators, 7, (SYSTEM REF.)
Action switches,(LINK-10)
  Delayed, 684, 695, 700, 708, 713, 717, 720, 726, 730, 731, 741, 743, 745, 748, 749, 755 762, 765, 771
  Immediate, 693, 697, 698, 703, 706, 718, 735, 739, 746, 754, 769
ACTIVATE UUO, 378, (MONITOR CALLS)
Active search list, 562, (MONITOR CALLS)
Active swapping list, 607, (MONITOR CALLS)
ADD, 49, (SYSTEM REF.)
Address assignments, 223, (MACRO)
Address break, 102, (SYSTEM REF.)
Addressing, 7, (SYSTEM REF.)
Addressing by monitor, 550, (MONITOR CALLS)
Address, (LINK-10)
  ignoring start, 738
  specifying start, 758
  start, 666, 677, 689, 713, 738, 758
Address mapping, 361, (MONITOR CALLS)
Address mode, relocatable or absolute, 227, (MACRO)
Addresses, symbolic, 212, (MACRO)
  elements, 280
  operands, 212
  operators, 211
ALGLIB, 740, (LINK-10)
Algorithms, switch, 679, (LINK-10)
Algorithms, 605, (MONITOR CALLS)
Algorithms, 179, (SYSTEM REF.)
  fixed point, 150
    addition, 150
    division, 183
    multiplication, 181
    subtraction, 181
Algorithms, (Cont.)
  floating point, 185
    addition, 186
    division, 188
    double precision division, 190
    multiplication, 188
    scaling, 188
    subtracting, 186
Allocating COMMON, 692, (LINK-10)
Allocating core, 360, 401, (MONITOR CALLS)
Allocating disk space, 709, (LINK-10)
Allocating patching space, 744, (LINK-10)
Altering a monitor location, 417, (MONITOR CALLS)
AND, 42, (SYSTEM REF.)
ANDCA, 42, (SYSTEM REF.)
ANDCB, 43, (SYSTEM REF.)
ANDCM, 42, (SYSTEM REF.)
Angle brackets (<>), 213, 261, 266, 327, (MACRO)
  coding, 252
AOBJN, 63, (SYSTEM REF.)
AOBJP, 63, (SYSTEM REF.)
AOJ, 66, (SYSTEM REF.)
AOS, 67, (SYSTEM REF.)
Appending a file, 586, (MONITOR CALLS)
APR, 95, 101, 105, (SYSTEM REF.)
APRENB UUO, 373, 387, (MONITOR CALLS)
APR trapping, 387, (MONITOR CALLS)
AR, 6, (SYSTEM REF.)
Area, (LINK-10)
  COMMON, 692
  DSK disk, 667
  NEW disk, 667
  OLD disk, 667
  SYS disk, 667
  expanding, 716
Arithmetic operations, 219, (MACRO)
Arithmetic shifting, 53, 163, (SYSTEM REF.)
Arithmetic testing, 63, 163, (SYSTEM REF.)
Arranging symbol table, 765, (LINK-10)
ARRAY pseudo-op, 242, (MACRO)
AS, 5, (SYSTEM REF.)
ASCII, 6-bit, 240, (MACRO)
ASCII, 7-bit, 240, (MACRO)
ASCII interpretation, 268, (MACRO)
ASCII card codes, 627, (MONITOR CALLS)
ASCII statement, 240, (MACRO)
ASCIZ statement, 240, (MACRO)
ASH, 53, (SYSTEM REF.)
ASHC, 54, (SYSTEM REF.)
Assembler control statements, 253, (MACRO)
Assembler interpretation, 327, (MACRO)
Assembler, MACRO, 205, (MACRO)

Assembly, 900, (DDT)
Assembly listing, 205, 283, (MACRO)
Assigning values, 705, (LINK-10)
Assignment delay, 652, (LINK-10)
Assignment statements, direct, 215, (MACRO)
Assignments, 215, (MACRO)
Assignments, address, 223, (MACRO)
ASUPPRESS pseudo-op, 248, (MACRO)
ATTACH UUO, 379, (MONITOR CALLS)
@(at) character, 207, 224, (MACRO)
Automatically, Using LINK-10, 656, 659, (LINK-10)
Auxiliary output, 651, 654, (LINK-10)

B (binary radix), 218, (MACRO)
Back Slash (\), 268, 328, (MACRO)
/BACKSPACE, 686, 691. (LINK-10)
Backspacing tapes, 691, (LINK-10)
Bad address subtable, 439 (MONITOR CALLS)
Bad allocation table, 612 (MONITOR CALLS)
BCD (binary coded decimal) code, 639, (MONITOR CALLS)
Binary files, 651, (LINK-10)
Binary files, loading, 360, (MONITOR CALLS)
Binary program output, 284, 285, (MACRO)
Binary shifting, 221, (MACRO)
Binary value interpretation, 903, (DDT)
Bit assignments, in-out, 195, (SYSTEM REF.)
Blank character, 331, (MACRO)
Blank COMMON, 692, (LINK-10)
Blank field, 233, (MACRO)
BLIST, 36, (SYSTEM REF.)
BLKI, 92, (SYSTEM REF.)
BLKO, 92, (SYSTEM REF.)
Block 10, 92, (SYSTEM REF.)
Block allocation, DECtape, 540, (MONITOR CALLS)
Block mode, 444, (MONITOR CALLS)
BLOCK statements, 243, (MACRO)
Block transfer, 32, (SYSTEM REF.)
Block transfer instructions, 239, (MACRO)
Block types, 290, 291, 292, (MACRO)
BLT, 32, (SYSTEM REF.)
Boolean functions, 39, 159, (SYSTEM REF.)
BR, 7, (SYSTEM REF.)
Brackets, angle, see angle brackets
Breakpoints, 874, 891, 911, (DDT)
  checking status, 893, 912
  conditional, 893, 912
  proceeding from, 876, 892, 894, 912
Breakpoints (Cont.)
  reassigning and removing, 876, 891, 912
  restrictions, 875, 892
  setting, 875, 891, 911
  type outs, 875, 907, 909
Buffer, 466, (MONITOR CALLS)
Buffered data mode, 465, 476, (MONITOR CALLS)
Buffer header, 465, (MONITOR CALLS)
Buffer initialization, 468, (MONITOR CALLS)
Buffer ring, 467, (MONITOR CALLS)
Buffer ring header block, 466, (MONITOR CALLS)
Busy, 93, (SYSTEM REF.)
Byte interrupt, 77, (SYSTEM REF.)
Byte manipulation, 37, 160, (SYSTEM REF.)
Byte manipulation, 237, 238, (MACRO)
Byte pointer, 37, (SYSTEM REF.)
Byte pointer, 225, (MACRO)

CAI, 64, (SYSTEM REF.)
CALL and CALLI operations, 371, (MONITOR CALLS)
  table of, 372
CAM, 65, (SYSTEM REF.)
Card codes, 627, (MONITOR CALLS)
Card codes, 172, (SYSTEM REF.)
Card punch, 494, (MONITOR CALLS)
Card reader, 496, (MONITOR CALLS)
Carries, 48, (SYSTEM REF.)
Carriage return, 332, (MACRO)
Carry 0, 77, (SYSTEM REF.)
Carry 1, 77, (SYSTEM REF.)
CDB constants table, 433 (MONITOR CALLS)
CDB variables table, 434, (MONITOR CALLS)
Central processor specification, 700, (LINK-10)
Central processor flags, 387, (MONITOR CALLS)
Change of unit pointer, 642, (MONITOR CALLS)
Changing defaults, 678, 703, (LINK-10)
Changing local radix, 235 (MACRO)
Changing magnetic tape modes, 509, (MONITOR CALLS)
Changing switches, (LINK-10)
  status, 684
Changing the logical station, 412, (MONITOR CALLS)
Channel command chaining, 611, (MONITOR CALLS)
Channel interrupt routines, 613, (MONITOR CALLS)
Character handling, 327, (MACRO)
Character set, radix 50, 210, (MACRO)

Characters, summary of special,
315 through 3-17, (MACRO)
Checking file access, 588, (MONITOR CALLS)
CHGPPN UUO, 378, (MONITOR CALLS)
CHKACC UUO, 379, 588, (MONITOR CALLS)
↑C intercept, 389, (MONITOR CALLS)
CLEAR, 30, (SYSTEM REF.)
Clearing DECtapes, 774, (LINK-10)
Clearing directory, 774, (LINK-10)
Clearing initial symbol table,
734, (LINK-10)
Clearing the write-protect bit,
403, (MONITOR CALLS)
CLK, 113, (SYSTEM REF.)
Clock, (SYSTEM REF.)
flag, 102
line frequency, 102
real time DK10, 113
operation, 116
CLOCK function, 443, (MONITOR CALLS)
CLOSE UUO, 371, 481, (MONITOR CALLS)
Cluster count, 641, (MONITOR CALLS)
Clusters, 550, 607, (MONITOR CALLS)
COBDDT, 701 (LINK-10)
COBOL programs, (LINK-10)
loading, 803, 805
Code, (LINK-10)
impure, 753
pure, 753
relocatable, 651
Colon, double (::), 246 (MACRO)
Colon (:) as label terminator
210, 246, (MACRO)
.COMM., 692, (LINK-10)
Comma usage, 328, (MACRO)
Commands, (LINK-10)
COMPIL-class, 656
DEBUG, 656, 660
EXECUTE, 654, 656, 660
GET, 654, 748
LOAD, 656, 659
R LINK, 656, 657, 675
RUN, 654
SAVE, 659, 668
SSAVE, 659, 668
START, 654, 659
Command file, 404 (MONITOR CALLS)
Command format, (LINK-10)
COMPIL-class, 660
Command strings, 676, 681, (LINK-10)
Comment, 208, 213, (MACRO)
Comments, 675 (LINK-10)
COMMENT statement, 251, (MACRO)
/COMMON, 686, 692, (LINK-10)
COMMON, (LINK-10)
allocating, 692
blank, 692
COMMON area, 692, (LINK-10)
COMMON symbols, 695, (LINK-10)
Comparing files,(UTILITY)
FILCOM, 931
Comparison of disk-like devices,
637, (MONITOR CALLS)
COMPIL program, 656, (LINK-10)
/COMPIL switch, 257, (MACRO)
COMPIL switches, 662, 664, (LINK-10)
COMPIL-class command, 656, (LINK-10)
format, 660
Complement, 8, (SYSTEM REF.)
Compressed file pointer, 552,
(MONITOR CALLS)
Concatenation, 263, 329, (MACRO)
Conditional assembly, 252, (MACRO)
Conditional break instruction, 894,
912, (DDT)
Conditions in, 93, (SYSTEM REF.)
see status
Conditions out, 93, (SYSTEM REF.)
clock, 114
interrupt, 98
processor, 98
Configuration information,(MONITOR CALLS)
LIGHTS, 442
SWITCH, 442
Configuration table, 423, (MONITOR CALLS)
CONI, 89, (SYSTEM REF.)
CONO, 90, (SYSTEM REF.)
CONSO, 92, (SYSTEM REF.)
Console, 95, (SYSTEM REF.)
Console operator panel, 106, (SYSTEM REF.)
Constants table, 433, (MONITOR CALLS)
CONSZ, 91, (SYSTEM REF.)
/CONTENTS, 687, 694, (LINK-10)
Contents of map file, 694, (LINK-10)
Continuation lines, 675, 681,
(LINK-10)
Continued MFD, 551,(MONITOR CALLS)
Continued SFD, 552, (MONITOR CALLS)
Continued UFD, 552, (MONITOR CALLS)
Control left arrow (CTRL←), 209 (MACRO)
/CORE, 687, 697, (LINK-10)
Core, (LINK-10)
free, 716
Core allocation resource, 397,
(MONITOR CALLS)
Core allocation unit, 360, (MONITOR CALLS)
Core control, 393, (MONITOR CALLS)
CORE, 401
definitions, 393
LOCK, 394
SETUWP, 403
UNLOK.,397
Core image, 654, 659, 668, 750, 757,
(LINK-10)
Core image file, (LINK-10)
expanded, 690, 772
Core memory, 652, (LINK-10)

Core storage, 360, (MONITOR CALLS)
CORE UUO, 372, 401, (MONITOR CALLS)
CORMAX, 394, (MONITOR CALLS)
CORMAX, 716, (LINK-10)
CORMIN, 393, (MONITOR CALLS)
/COUNTER, 687, 698, (LINK-10)
Counters, Relocation, 686, 689, 698, 754, (LINK-10)
CPA, 101, (SYSTEM REF.)
/CPU, 687, 700, (LINK-10)
Created symbols, 262, (MACRO)
Creating a file, 563, (MONITOR CALLS)
Creating a saved file, 805, (LINK-10)
Creating XPN file, 772, (LINK-10)
CREF command, 928,(UTILITY)
CREF program, 925, (UTILITY)
Cross-referenced listings (UTILITY)
  CREF, 925
  GLOB, 951
CTLJOB UUO, 378, 520, (MONITOR CALLS)

D (decimal radix), 218, (MACRO)
/D, 687, (LINK-10)
DAEFIN UUO, 379, (MONITOR CALLS)
DAEMON UUO, 379, 442, (MONITOR CALLS)
/DATA, 687, (LINK-10)
Data channel, 463, (MONITOR CALLS)
Data error, 612, (MONITOR CALLS)
DATAI, 91, (SYSTEM REF.)
DATAO, 91, (SYSTEM REF.)
Data modes, 464, (MONITOR CALLS)
  card punch, 494
  card reader, 497
  DECtape, 536
  disk, 549
  display, 499
  line printer, 502
  magnetic tape, 503
  paper-tape punch, 512
  paper-tape reader, 513
  plotter, 515
  terminal, 522
Data transmission, 474, (MONITOR CALLS)
DATE UUO, 373, 415, (MONITOR CALLS)
.DCORE function, 442, (MONITOR CALLS)
DDT, 660, 664, 701, (LINK-10)
/DDT, 664, (LINK-10)
DDTGT UUO, 372, (MONITOR CALLS)
DDTIN UUO, 372, 525, (MONITOR CALLS)
DDTOUT UUO, 372, 525, (MONITOR CALLS)
DDTRL UUO, 372, (MONITOR CALLS)
DDT submode, 524, (MONITOR CALLS)
DEACTIVATE UUO, 378, (MONITOR CALLS)
Dead reckoning, 548, (MONITOR CALLS)
/DEBUG, 684, 687, 701, (LINK-10)
DEBUG command, 656, 660, (LINK-10)
Debugging, 654, 722, 723, 727, (LINK-10)
Debugging program, 660, 686, 690, (LINK-10)
  loading a, 701, 766
DEC statement, 234, (MACRO)
Decimal numbers,(MACRO)
  fixed point, 222
  floating point, 222
Decimal print routine, 87,(SYSTEM REF.)
DECsystem-1040, 368, (MONITOR CALLS)
DECtape, 536, (MONITOR CALLS)
  block allocation, 540
  directory format, 537
  file format, 539
  format, 536
  I/O programming, 540
DECtape compatibility, 621, (MONITOR CALLS)
DECtapes, clearing, 774, (LINK-10)
DEC-029 card codes, 624, (MONITOR CALLS)
DEC-026 card codes, 630 (MONITOR CALLS)
/Default, 678, 687, 703, (LINK-10)
Default values, (LINK-10)
  initial, 677, 678, 686
Defaults, (LINK-10)
  changing, 678, 703
  input, 677
  output, 677
/DEFINE, 687, 705 (LINK-10)
Defined operator, 214, (MACRO)
DEFINE operator, 259, (MACRO)
Defining save file, 757, (LINK-10)
Defining symbol file, 759, (LINK-10)
Definition of macros, 259, (MACRO)
Delay, (LINK-10)
  assignment, 652
Delayed action switches, 684, 695, 700, 708, 713, 717, 720, 726, 730, 731, 741, 743, 745, 748, 749, 755, 762, 765, 771, (LINK-10)
Deleted symbols, 217, (MACRO)
Delimiter, 663, (LINK-10)
Delimiters, (MACRO)
  =:, 216
  parentheses, 237
DEPHASE statement, 229, (MACRO)
Determining buffer sizes, 490 (MONITOR CALLS)
Determining properties of devices, 489, (MONITOR CALLS)
Determining two-segment capability, 403, (MONITOR CALLS)

DEVCHR UUO, 372, 488, (MONITOR CALLS)
DEVGEN UUO, 378, (MONITOR CALLS)
Device errors, 388, 586, (MONITOR CALLS)
Device information, 487, (MONITOR CALLS)
  DEVCHR, 488
  DEVNAM, 491
  DEVSIZ, 490
  DEVSTS, 487
  DEVTYP, 489
  WHERE, 491
Device initialization, 463, (MONITOR CALLS)
Device name, 660, 677, (LINK-10)
Device name, 463, (MONITOR CALLS)
Device optimization, 609, (MONITOR CALLS)
Device reassignment, 483, (MONITOR CALLS)
Devices, 493, 535, (MONITOR CALLS)
  card punch, 494
  card reader, 496
  DECtape, 536
  disk, 549
  display, 499
  line printer, 502
  magnetic tape, 503
  paper-tape punch, 512
  paper-tape reader, 513
  plotter, 515
  pseudo-TTY, 516
  terminal, 521
Device selection, 462, (MONITOR CALLS)
Device status bits, 631, (MONITOR CALLS)
Device switches, 680, 691, 733, 747, 756, 768, (LINK-10)
Device termination, 483, (MONITOR CALLS)
DEVLNM UUO, 380, 484, (MONITOR CALLS)
DEVNAM UUO, 378, 491, (MONITOR CALLS)
DEVPPN UUO, 377, 593, (MONITOR CALLS)
DEVSIZ UUO, 379, 490, (MONITOR CALLS)
DEVSTS UUO, 377, 487, (MONITOR CALLS)
DEVTYP UUO, 377, 489, (MONITOR CALLS)
DFN, 59 , (SYSTEM REF.)
Diagnostic messages (UTILITY)
  CREF, 929
  FILCOM, 935
  FUDGE2, 946
  GLOB, 953
Differences, (LINK-10)
  LOADER and LINK-10, 843
Direct addressing, 11, (SYSTEM REF.)
Directory algorithms, 613, (MONITOR CALLS)
Direct assignment statements, 215, (MACRO)
Directly, (LINK-10)
  using LINK-10, 657, 675
Directory, 660, 677, (LINK-10)
  clearing, 774
Directory devices, 463, 535, (MONITOR CALLS)
Directory path, 552, 577, 580, (MONITOR CALLS)
Disk, 549, (MONITOR CALLS)
  access protection, 554
  file directories, 551
  file structure names, 560
  job search list, 562
  packs, 602
  quotas, 559
  status, 601
  structure of files, 549
  user programming, 563
  UUOs, 577
Disk area, (LINK-10)
  DSK, 667
  NEW, 667
  OLD, 677
  SYS, 667
Disk file organization, 551, 553, (MONITOR CALLS)
Disk overflow, 729 (LINK-10)
Disk packs, 602, (MONITOR CALLS)
Disk parameters, 429, (MONITOR CALLS)
Disk priority, 599, (MONITOR CALLS)
Disk quotas, 391, 412, 559, (MONITOR CALLS)
Disk space, (LINK-10)
  allocating, 709
Disk unit offline, 390, (MONITOR CALLS)
DISK.UUO, 381, 599, (MONITOR CALLS)
Disk writing, types of, 563, (MONITOR CALLS)
Dismissing an interrupt, 97, (SYSTEM REF.)
Dismissing a real-time interrupt, 449, (MONITOR CALLS)
Display, 499, (MONITOR CALLS)
DIV, 51, (SYSTEM REF.)
DK10, 113, (SYSTEM REF.)
Done, 93, (SYSTEM REF.)
Dormant segment, 607, (MONITOR CALLS)
Double colon (::), 246, (MACRO)
Double equal sign ==, 217, (MACRO)
Double pound sign (##), 247, (MACRO)
Double precision floating point, 89, (SYSTEM REF.)
DPB, 38, (SYSTEM REF.)
DS, 5, (SYSTEM REF.)
DSK disk area, 667, (LINK-10)
DSKCHR UUO, 376, 597, (MONITOR CALLS)
Dump mode, 464, 475, (MONITOR CALLS)

Dummy symbols, 264, (MACRO)
DVRST.UUO, 381, (MONITOR CALLS)
DVURS.UUO, 381, (MONITOR CALLS)

E, 11, 23, (SYSTEM REF.)
/E, 687, (LINK-10)
Effective address calculation, 11, (SYSTEM REF.)
End block type 5, 287, (MACRO)
END statements, 243, 290, (MACRO)
End-of-file card, 494, 497, (MONITOR CALLS)
ENTER, 371, 471, (MONITOR CALLS)
  DECtape, 542
  disk, 565, 571
  error codes, 635
/ENTRY, 687, 706, (LINK-10)
Entry block type 4, 287, (MACRO)
Entry name symbols, 695, (LINK-10)
Entry points, 706, 713, 746, (LINK-10)
ENTRY statement, 706, (LINK-10)
ENTRY statement, 247, (MACRO)
Entry symbols, 667, (LINK-10)
Environmental information, 414, (MONITOR CALLS)
  configuration, 442
  job status, 416
  monitor, 416
  timing, 414
EOF, magnetic tape, 503, 504, (MONITOR CALLS)
Equal signs == (double), 217, (MACRO)
EQV, 45, (SYSTEM REF.)
Error codes, 635, (MONITOR CALLS)
  naming conventions, 384
Error code printing, 284, (MACRO)
Error codes, single-letter, 269 through 274, (MACRO)
Error intercepting, 388, (MONITOR CALLS)
/ERRORLEVEL, 687, 708, 775, (LINK-10)
Error messages, 275 through 281, (MACRO)
Error messages, 877, 886, (DDT)
Error messages, (UTILITY)
  CREF, 929
  FILCOM, 935
  FUDGE2, 946
  GLOB, 953
Error recovery, RUN, 405, (MONITOR CALLS)
Errors, total number, 284, (MACRO)
Ersatz devices, 593, (MONITOR CALLS)
/ESTIMATE, 687, 709, (LINK-10)
Evaluation of expression, 214, 219, (MACRO)
Evaluation of statement, 214, (MACRO)
Examining storage words, 871, 879, 908, (DDT)
Examining the monitor, 416, (MONITOR CALLS)
Examples, 669, 803, (LINK-10)
Examples (MONITOR CALLS)
  ↑C intercept, 389, 390
  device initialization, 473
  dump output, 475
  expanding core, 403
  file reading, 485, 486
  file writing, 485, 486
  GETTAB subtables, 436
  inputting one character, 476
  outputting one character, 478
  paper-tape input, 534
  pointer list, display, 500
  reading search list, 591
  reading UFD, 595
  real-time trapping, 449, 454
  SFD's, 581
  terminating a file, 483
  testing high segments, 409
  TRPSET, 456
  user generated buffers, 469
  writing reentrant programs, 623
Examples, (UTILITIES)
  FILCOM, 937
  FUDGE2, 949
  GLOB, 954
Excess-128 code, 9, (SYSTEM REF.)
EXCH, 31, (SYSTEM REF.)
Exceeded time limit, 391, (MONITOR CALLS)
/EXCLUDE, 687, 711, (LINK-10)
Exec mode program, 734, (LINK-10)
Executable version, 651, 653, 654, (LINK-10)
/EXECUTE, 687, 713, (LINK-10)
EXECUTE command, 654, 656, 660, (LINK-10)
Execution, (LINK-10)
  specifying, 701, 713, 718, 766
Execution control, 385, (MONITOR CALLS)
  starting, 385
  stopping, 385
  suspending, 391
  trapping, 387
Executive mode, 368, (MONITOR CALLS)
Executive mode debugging,(EDDT), 915, (DDT)
Executive mode trapping, RTTRP, 453
Exhausted disk quota, 391, 412, (MONITOR CALLS)
Exit condition switch, 676, (LINK-10)
EXIT UUO, 373, 386, (MONITOR CALLS)
Expanded core image file, 690, 772, (LINK-10)
Expanding areas, 716, (LINK-10)
Expanding core, 402 (MONITOR CALLS)

Experimental SYS, 579, (MONITOR
CALLS)
Expressions, 874, (DDT)
Expressions, 207, MACRO)
evaluation of, 214, 219
nesting of, 220
Expression evaluation, 902, (DDT)
Expressions, nesting of, 220, (MACRO)
literals, 225
macros, 265
Extended arguments for LOOKUP,ENTER,
RENAME, 571, (MONITOR CALLS)
Extended instruction statements,
256, (MACRO)
Extended RIB's 585, (MONITOR CALLS)
Extension, (LINK-10)
filename, 660, 677
EXTERN statement, 246, (MACRO)
External symbols, 692, (LINK-10)

F (fixed point decimal fractions),
218, (MACRO)
FAD, 60, (SYSTEM REF.)
FADR, 57, (SYSTEM REF.)
Fast block mode, 444, (MONITOR CALLS)
Fast memory, 7, (SYSTEM REF.)
Fatality of messages, (LINK-10)
specifying, 755
FDV, 62, (SYSTEM REF.)
FDVR, 58, (SYSTEM REF.)
Feature table, 439, (MONITOR CALLS)
Feature test switches, 368, 439,
(MONITOR CALLS)
Fields, 214, (MACRO)
blank, 253
Field separators, 901, 911, (DDT)
FILCOM program, 931, (UTILITY)
File, (LINK-10)
creating a saved, 805
creating XPN, 772
defining save, 757
defining symbol, 759
expanded core image, 690, 772
log, 683, 688, 723, 725
map, 688, 727, 731
save, 689, 750, 757
saving XPN, 772
specifying log, 723
specifying map, 727
symbol, 689, 690, 759
XPN, 772
File dependent switches, 681,
712, 721, 722, 736, 737, 738,
752, 753, 758
File directories, 551 (MONITOR CALLS)
File format, (MONITOR CALLS)
DECtape, 539
disk, 554
Filename, 660, 677, (LINK-10)
Filename extension, 660, 677,
(LINK-10)
File reading, example, 485, (MONITOR
CALLS)
File retrieval pointers, 641, (MONITOR
CALLS)
Files, (LINK-10)
binary, 651
output, 677, 679, 682, 703, 718
Files, 461, 552, (MONITOR CALLS)
File selection, 470, (MONITOR CALLS)
File specification, 660, 676, 703,
(LINK-10)
File specification switches, (LINK-10)
implicit, 702, 714, 715, 764, 766
File status, 464, 479, (MONITOR CALLS)
card punch, 496
card reader, 498
DECtape, 547
disk, 601
display, 501
line printer, 502
magnetic tape, 511
paper-tape punch, 513
paper-tape reader, 514
plotter, 516
pseudo-TTY, 518
terminal, 533
File status checking, 480, (MONITOR
CALLS)
File status setting, 480, (MONITOR
CALLS)
File structure names, 560, (MONITOR
CALLS)
File termination, 481, (MONITOR CALLS)
File writing, example, 485, (MONITOR
CALLS)
FILSER, 549, 609, (MONITOR CALLS)
Fixed point arithmetic, 48, 160,
(SYSTEM REF.)
Fixed point decimal numbers, 222,
(MACRO)
Fixed point numbers, 8, (SYSTEM REF.)
double length, 48
Flag restoration, 81, (SYSTEM REF.)
Flags, 76, (SYSTEM REF.)
Floating overflow, 77, 102, (SYSTEM
REF.)
Floating point arithmetic, 54, 160,
(SYSTEM REF.)
Floating point decimal numbers, 222,
(MACRO)
Floating point numbers, 9, (SYSTEM
REF.)
double length, 10
double precision, 89
FMP, 62, (SYSTEM REF.)
FMPR, 58, (SYSTEM REF.)
Folded checksum, 641, (MONITOR CALLS)
FORLIB, 740, (LINK-10)
Format, (LINK-10)
COMPIL-class command, 660
triplet, 759

Formats, 193, (SYSTEM REF.)
Format for macro calls, 261, (MACRO)
Format of instruction word, 206, (MACRO)
Format, primary instruction, 207, (MACRO)
/FOROTS, 664, 688, 714, (LINK-10)
FOROTS, (LINK-10)
  loading, 714
/FORSE, 664, 688, 715, (LINK-10)
FORSE, (LINK-10)
  loading, 715
Forward tapes, (LINK-10)
  spacing, 756
FRCUUO UUO, 379, (MONITOR CALLS)
FRECHN UUO, 377, (MONITOR CALLS)
/FRECOR, 688, 716, (LINK-10)
Free core, 716, (LINK-10)
FSB, 61, (SYSTEM REF.)
FSBR, 57, (SYSTEM REF.)
FSC, 56, (SYSTEM REF.)
FUDGE2 program, 737, (UTILITY)
Full file structure, 391, 412, (MONITOR CALLS)
Full word data transmission, 31, 161, (SYSTEM REF.)
FUNCTION statement, 706, (LINK-10)
Functions, (LINK-10)
  performing magnetic tape, 732

/G, 688, (LINK-10)
General form of statements, 206, (MACRO)
GET command, 654, 748, (LINK-10)
GETCHR UUO, 372, (MONITOR CALLS)
GETLIN UUO, 374, 529, (MONITOR CALLS)
GETPPN UUO, 374, 416, (MONITOR CALLS)
GETSEG UUO, 375, 407, (MONITOR CALLS)
GETSTS UUO, 370, 480, (MONITOR CALLS)
GETTAB tables, 419, (MONITOR CALLS)
  .GTADR, 419
  .GTCM2, 421
  .GTCNF, 419, 423
  .GTCNO, 420
  .GTCOM, 420
  .GTCOR, 420
  .GTCRS, 421
  .GTCØC, 422, 433
  .GTC2C, 422
  .GTC3C, 422
  .GTC4C, 422
  .GTC5C, 422
  .GTCØV, 422, 434
  .GTC1V, 422
  .GTC2V, 422
  .GTC3V, 422
  .GTC4V, 422
  .GTC5V, 422
  .GTDBS, 420
  .GTDEV, 420
  .GTFET, 422, 439
  .GTISC, 421
  .GTKCT, 419
  .GTLDV, 419, 429
  .GTLIM, 421
  .GTLOC, 420
  .GTNM1, 420
  .GTNM2, 420
  .GTNSW, 419, 427
  .GTODP, 419, 429
  .GTOSC, 421
  .GTPPN, 419
  .GTPRG, 419
  .GTPRV, 419, 423
  .GTQJB, 421
  .GTQQQ, 421
  .GTRCT, 420
  .GTRSP, 421
  .GTRTD, 420
  .GTSDT, 419, 429
  .GTSGN, 419
  .GTSLF, 420, 431
  .GTSPL, 420, 432
  .GTSPS, 422
  .GTSSC, 421
  .GTSTS, 419
  .GTSWP, 419
  .GTSYS, 421
  .GTTDB, 420
  .GTTIM, 419
  .GTTMP, 420
  .GTTRQ, 421
  .GTTTY, 419
  .GTWCH, 420, 432
  .GTWCT, 420
  .GTWHY, 421
  .GTWSN, 420, 431
  naming of, 383
  subtables, 436
GETTAB UUO, 375, 417, (MONITOR CALLS)
GLOB program, 951, (UTILITY)
Global requests, 667, 746, (LINK-10)
  undefined, 752, 763, 767
Global symbol table, 688, (LINK-10)
  initial, 734
Global symbols, 694, 695, 769, (LINK-10)
  typing in, 769
  undefined, 705
Global symbols, listing of, 951, (GLOB)
Global symbols, 216, (MACRO)
Globals, (LINK-10)
  typing undefined, 767
/GO, 688, 718, (LINK-10)
GOBSTR UUO, 378, 591, (MONITOR CALLS)
Group pointer, 641, (MONITOR CALLS)
Guidelines for locking jobs, 396, (MONITOR CALLS)

Half word data transmission, 24, 161, (SYSTEM REF.)
HALT, 386, (MONITOR CALLS)
HALT, 81, (SYSTEM REF.)
Hard error, 612, (MONITOR CALLS)
Hardware detected errors, 612, (MONITOR CALLS)
Hash table, 719, (LINK-10)
/HASHSIZE, 688, 719, (LINK-10)
Hashsize, (LINK-10)
  recommended, 719
Header card, 494, 496, (MONITOR CALLS)
Hardware read-in format, 288, (MACRO)
HIBER UUO, 378, 391, 517, (MONITOR CALLS)
.HIGH., 698, 754, (LINK-10)
High priority queues, 457, (MONITOR CALLS)
High segment table, 419, (MONITOR CALLS)
HISEG pseudo-op, 232, (MACRO)
HLL, 25, (SYSTEM REF.)
HLLE, 26, (SYSTEM REF.)
HLLO, 26, (SYSTEM REF.)
HLLZ, 25, (SYSTEM REF.)
HLR, 29, (SYSTEM REF.)
HLRE, 30, (SYSTEM REF.)
HLRO, 30, (SYSTEM REF.)
HLRZ, 30, (SYSTEM REF.)
HPQ UUO, 378, 457, (MONITOR CALLS)
HRL, 26, (SYSTEM REF.)
HRLE, 27, (SYSTEM REF.)
HRLO, 27, (SYSTEM REF.)
HRLZ, 27, (SYSTEM REF.)
HRR, 28, (SYSTEM REF.)
HRRE, 29, (SYSTEM REF.)
HRRO, 29, (SYSTEM REF.)
HRRZ, 28, (SYSTEM REF.)
Hyphen, 675, (LINK-10)

I, 11, (SYSTEM REF.)
IBP, 38, (SYSTEM REF.)
IBUF, 465, (MONITOR CALLS)
IDIV, 51, (SYSTEM REF.)
Idle segment, 607, (MONITOR CALLS)
IDPB, 38, (SYSTEM REF.)
IF statement, 252, 253, (MACRO)
Ignoring start address, 738, (LINK-10)
ILDB, 38, (SYSTEM REF.)
Illegal instructions, 385, (MONITOR CALLS)
Illegal operation codes, 383, (MONITOR CALLS)
Image, (LINK-10)
  core, 659, 668, 750, 757
Image file, (LINK-10)
  expanded core, 690, 772
Immediate action switches, 693, 697, 698, 703, 718, 735, 739, 746, 754, 769, (LINK-10)
Implicit file specifications switches, 684, 702, 714, 715, 764, 766, (LINK-10)
Impure code, 753, (LINK-10)
Impure segment, 359, (MONITOR CALLS)
IMUL, 50, (SYSTEM REF.)
INBUF UUO, 370, 468, (MONITOR CALLS)
/INCLUDE, 688, 721, (LINK-10)
Index registers, 7, (SYSTEM REF.)
Indexing, 207, 244, (MACRO)
Indicator panels, 182, (SYSTEM REF.)
Indicators, 106, (SYSTEM REF.)
Indirect addressing, 11, (SYSTEM REF.)
Indirect addressing, 224, (MACRO)
Inhibiting loading of modules, 711, (LINK-10)
Inhibiting searching, 740, (LINK-10)
Initial conditions, 93, (SYSTEM REF.)
Initial default values, 677, 678, 686, (LINK-10)
Initial file status, 464, (MONITOR CALLS)
Initial global symbol table, 734, (LINK-10)
Initial low segment size, 697, (LINK-10)
Initial symbol table, (LINK-10)
  clearing, 734
  size, 719
Initialization of LINK-10, 656 (LINK-10)
Initializing devices, 464, (MONITOR CALLS)
Initializing the high segment, 407, (MONITOR CALLS)
INIT UUO, 369, 463, (MONITOR CALLS)
In-out, 82, 163, (SYSTEM REF.)
In-out bit assignments, 180, (SYSTEM REF.)
In-out devices, 156, 180, (SYSTEM REF.)
Input, buffered, 476, (MONITOR CALLS)
Input defaults, 677, (LINK-10)
Input specifications, 676, 703, (LINK-10)
Input-output, 82, 163, (SYSTEM REF.)
Input spooling, 604, (MONITOR CALLS)
Input to LINK-10, 651, (LINK-10)
Input UUO, 370, 474, (MONITOR CALLS)
Instruction format, 10, (SYSTEM REF.)
Instruction times, 23, (SYSTEM REF.)
Instruction word format, 206, (MACRO)
Instructions, 157, (SYSTEM REF.)
  arithmetic testing, 63, 163
  Boolean functions, 40, 159
  byte, 38, 160
  fixed point, 49, 160
  floating point, 56, 160
    without rounding, 59
    with rounding, 57
  full word, 31, 161
  half word, 25, 161

Instructions (Cont.)
  in-out, 90, 163
  jump, 78, 163
  logic, 40, 159
  logical testing, 70, 165
  move, 33, 161
  pushdown, 35, 84, 163
  rotate, 47, 163
  shift, 47, 53, 163
INTEGER pseudo-op, 242, (MACRO)
Interactive options, 656, 683, (LINK-10)
Intercept,↑C, 389, (MONITOR CALLS)
Internal request, block type 10, 288, (MACRO)
Internal symbols, 692,(LINK-10)
INTERN statement, 247, (MACRO)
Inter-program communication, 412, (MONITOR CALLS)
Interrupt, 95, (SYSTEM REF.)
Interrupt chains, 614, (MONITOR CALLS)
Interrupt level use of RTTRP, 447, (MONITOR CALLS)
Interrupt requests, 96, (SYSTEM REF.)
Introduction to LINK-10, 651, (LINK-10)
IO, 82, (SYSTEM REF.)
I/O error messages, 278, (MACRO)
I/O monitor facilities, 209, (MACRO)
I/O organization, 461, (MONITOR CALLS)
I/O programming, 461, (MONITOR CALLS)
  DECtape, 540
  disk, 563
I/O transfer words, 239, (MACRO)
IOR, 43, (SYSTEM REF.)
IOWD, 475 (MONITOR CALLS)
IOWD statement, 239, (MACRO)
IR, 6, (SYSTEM REF.)
Item types, (LINK-10)
  LINK, 815

JACCT bit, 404, (MONITOR CALLS)
.JB41, 362, (MONITOR CALLS)
.JBAPR, 363, (MONITOR CALLS)
.JBBLT, 362, (MONITOR CALLS)
.JBCN6, 362, (MONITOR CALLS)
.JBCHN, 364, (MONITOR CALLS)
.JBCNI, 363, (MONITOR CALLS)
.JBCOR, 364, (MONITOR CALLS)
.JBCST, 364, (MONITOR CALLS)
.JBDA, 364, (MONITOR CALLS)
.JBDDT, 362, (MONITOR CALLS)
.JBERR, 362, (MONITOR CALLS)
.JBFF, 363, (MONITOR CALLS)
.JBH41, 365, (MONITOR CALLS)
.JBHCR, 365, (MONITOR CALLS)
.JBHDA, 365, (MONITOR CALLS)
.JBHNM, 365, (MONITOR CALLS)
.JBHRL, 362, (MONITOR CALLS)
.JBHRN, 365, (MONITOR CALLS)
.JBHSA, 365, (MONITOR CALLS)
.JBHSM, 365, (MONITOR CALLS)
.JBHVR, 365, (MONITOR CALLS)
.JBINT, 364, 388, (MONITOR CALLS)
.JBOPC, 363, (MONITOR CALLS)
.JBOPS, 364, (MONITOR CALLS)
.JBPFI, 362, (MONITOR CALLS)
.JBREL, 362, (MONITOR CALLS)
.JBREN, 363, (MONITOR CALLS)
.JBSA, 363, (MONITOR CALLS)
.JBSET.UUO, 380, (MONITOR CALLS)
.JBSYM, 363, (MONITOR CALLS)
.JBTPC, 363, (MONITOR CALLS)
.JBUSY, 363, (MONITOR CALLS)
.JBUUO, 362, (MONITOR CALLS)
.JBVER, 364, (MONITOR CALLS)
JCRY, 79, (SYSTEM REF.)
JCRYO, 79, (SYSTEM REF.)
JCRY1, 79, (SYSTEM REF.)
JEN, 81, (SYSTEM REF.)
JFCL, 79, (SYSTEM REF.)
JFFO, 78, (SYSTEM REF.)
JFOV, 79, (SYSTEM REF.)
Jiffy, 415, 419, (MONITOR CALLS)
JLOG bit, 404, (MONITOR CALLS)
Job, 605, (MONITOR CALLS)
JOBDAT, 360, 362, (MONITOR CALLS)
Job data area, 360, 362, (MONITOR CALLS)
  vestigial, 365
JOBDAT, 722, (LINK-10)
Job's current name, 668, (LINK-10)
Job I/O initialization, 461, (MONITOR CALLS)
JOBPEK UUO, 379, (MONITOR CALLS)
Job privilege table, 423, (MONITOR CALLS)
Job search list, 562, (MONITOR CALLS)
Job state codes, 431, (MONITOR CALLS)
Job status information, (MONITOR CALLS)
  GETPPN, 416
  OTHUSR, 416
  PJOB, 416
  RUNTIME, 416
JOBSTR UUO, 376,590, (MONITOR CALLS)
JOBSTS UUO, 377, 519,(MONITOR CALLS)
JOV, 79, (SYSTEM REF.)
JRA, 83, (SYSTEM REF.)
JRST, 80, (SYSTEM REF.)
JRST 4, 386, (MONITOR CALLS)
JRSTF, 81, (SYSTEM REF.)
JSA, 83, (SYSTEM REF.)
JSP, 80, (SYSTEM REF.)
JSR, 79, (SYSTEM REF.)
JUMP, 65, (SYSTEM REF.)

Key, (LINK-10)
  RETURN, 675
Keys, 109, (SYSTEM REF.)
KT10 Option, 360, (MONITOR CALLS)

/L, 688, (LINK-10)
↑L (qualifier), 218, (MACRO)
Label field, 214, (MACRO)
Label of statement, 206, (MACRO)
Labels, 209, 210, 215, (MACRO)
LALL statement, 250, (MACRO)
Language, MACRO-10 statements, 206, (MACRO)
Latency time, 609, (MONITOR CALLS)
LDB, 38, (SYSTEM REF.)
Left arrow shifting, 222, (MACRO)
Level, (LINK-10)
  message, 708, 775
  severity, 755, 776
LIB40, 740, (LINK-10)
LIBOL, 740, (LINK-10)
Library, (LINK-10)
  modules, 653
  search, 653, 666, 763
  search mode, 666, 688, 689, 737, 752
  search symbols, 686, 706, 746
  suppression of search mode, 737
  system, 653, 666, 667, 689, 690, 740, 752, 763
  system search, 718
  user, 667
/LIBRARY, 665, (LINK-10)
Library subroutines, 248, (MACRO)
LIGHTS UUO, 372, 442, (MONITOR CALLS)
Line of program, 213, (MACRO)
Line printer, 502, (MONITOR CALLS)
Line printer listings, 250, (MACRO)
Lines, (LINK-10)
  continuation, 675, 681
LINK block, 257, (MACRO)
  formats, 284, 285
/LINK, 665, (LINK-10)
LINK item types, 815, (LINK-10)
LINK-10, (LINK-10)
  command strings, 676
  differences between LOADER and, 843
  initialization of, 656
  input to, 651
  introduction to, 651
  loading, 806
  messages, 775
  message suppression, 708, 725
  output from, 653
  switch algorithms, 679
  switches, 663, 685, 686, 691
Linking loader, 284, (MACRO)
Linking modules, 652, (LINK-10)
Linking subroutines, 246, (MACRO)
Linking symbols, 653, (MACRO)
List, (LINK-10)
  search, 667
LIST statement, 250, (MACRO)
Listing control statements, 250, (MACRO)
Listing relocation counters, 698, (LINK-10)
Literals, 255, (MACRO)
  nested, 255
LIT statements, 244, (MACRO)
/LMAP, 665, (LINK-10)
LOAD command, 656, 659, (LINK-10)
LOADER and LINK-10, (LINK-10)
  differences, 843
LOADER switches, 663, 846, (LINK-10)
Loading, 806, (LINK-10)
  COBOL programs, 803, 805
  debugging program, 701,766
  FOROTS, 714
  FORSE, 715
  inhibition of module loading, 711
  local symbols, 722
  MACRO programs, 804
  monitor, 811
  object time system, 742
  specified modules, 721
  symbols, 761
  termination, 718
Loading (MONITOR CALLS)
  binary files, 360
  core area, 360
Loading procedure, 869, 920, (DDT)
Local symbol mode, 701, (LINK-10)
Local symbols, 665,688, 695, 701, 736, 759
  loading, 722
Local symbols, 216, (MACRO)
/LOCALS, 688, 722, (LINK-10)
LOCATE UUO, 378, 412, (MONITOR CALLS)
Location counter, 224, (MACRO)
LOC pseudo-op, 227, (MACRO)
Lock UUO, 377, 394, (MONITOR CALLS)
Locking jobs, 394, 398, (MONITOR CALLS)
Logged-in quota, 555, (MONITOR CALLS)
Logged-out quota, 555, (MONITOR CALLS)
/LOG, 683, 688, 723, (LINK-10)
LOG, (LINK-10)
  logical name, 683, 723
Log file, 683, 688, 723, 725, (LINK-10)
Logic, 39, 159, (SYSTEM REF.)
Logical device names, suppression of, 382, (MONITOR CALLS)
Logical operations, 219, (MACRO)
Logical shifting, 46, 163, (SYSTEM REF.)
Logical station, 412, (MONITOR CALLS)
Logical testing and modification, 63, 165, (SYSTEM REF.)
Logical unit names, 560 (MONITOR CALLS)
LOGIN UUO, 373, (MONITOR CALLS)
/LOGLEVEL, 688, 725, 775, (LINK-10)
LOGOUT UUO, 373, (MONITOR CALLS)
LOOKUP UUO, 371, 470, (MONITOR CALLS)
  DECtape, 541
  disk, 567, 571
  error codes, 635
Lookup errors, 277, 278, (MACRO)
Low segment size, (LINK-10)
  initial, 697
  maximum, 729
.LOW., 698, 754, (LINK-10)
LSH, 47, (SYSTEM REF.)
LSHC, 47, (SYSTEM REF.)

/M, 688, (LINK-10)
MA, 6, (SYSTEM REF.)
Machine instruction formats, 206, (MACRO)
Machine mnemonics, summary, 309, (MACRO)
Macros, definition of, 259, (MACRO)
MACRO, (MACRO)
  calls, 260, 261
  capabilities, 205
  instruction, 256
  name, 211
  nesting and redefinition, 266
  program examples, 295
  statements, 209
  table, 213
MACRO program, loading, 804, (LINK-10)
Magnetic tape, 503, (MONITOR CALLS)
  codes, 639
  format, 504
  9-channel, 507
  use of MTAPE, 506
  UUOs, 505
Magnetic tape functions, 732, (LINK-10)
Manipulating file strucures, 588, (MONITOR CALLS)
MANTIS, 660, 701, (LINK-10)
Map, 666, 671, 679, (LINK-10)
/MAP, 665, 688, 694, 727, (LINK-10)
Map file, 688, 727, 731, (LINK-10)
  contents, 694
MAP program, 672, (MONITOR CALLS)
Mapping, 359, 360, (MONITOR CALLS)
Margin check panel, 107, (SYSTEM REF.)
Masks, naming of, 383, (MONITOR CALLS)
Master file directory, 551 (MONITOR CALLS)
MATRIX subroutine, 247, 248, (MACRO)
MAXCOR, 697, 716, (LINK-10)
/MAXCOR, 688, 729, (LINK-10)
Maximum low segment size, 729, (LINK-10)
Meddling, 404, 409, (MONITOR CALLS)
Memory, (LINK-10)
  core, 652
Memory, 12, (SYSTEM REF.)
  access time, 13
  allocation, 13
  protection, 102, 103
  stop, 108
Memory parity error recovery, 618, (MONITOR CALLS)
Memory protection and relocation, 359, (MONITOR CALLS)
Memory protection register, 360, (MONITOR CALLS)
Memory relocation register, 359, (MONITOR CALLS)
Message level, 708, 775, (LINK-10)
Messages, 775, (LINK-10)
  specifying fatality of, 755
  supressing, 708, 725
METER.UUO, 380, 458, (MONITOR CALLS)
MFD, 551, (MONITOR CALLS)
MI, 6, (SYSTEM REF.)
Miscellaneous commands, 897, (DDT)
Miscellaneous features, 655, (LINK-10)
Mnemonics, 14, 147, (SYSTEM REF.)
  alphabetic, 152
  derivation, 148
  device, 156
  numeric, 149
Mnemonics, summary of machine, 309, (MACRO)
Mnemonic table, 213, (MACRO)
Mode, (LINK-10)
  library search, 688, 689, 737, 752
  local symbol, 701
  search, 721
  supressing library search, 737
Modes, 23, (SYSTEM REF.)
  arithmetic testing, 64
  fixed point, 49
  floating point, 56, 60
  half word, 25
  logic, 39
  logical testing, 69
  move, 33
Modifying shared segments, 409, (MONITOR CALLS)
Modifying storage words, 872, 880, (DDT)
Modules, (LINK-10)
  inhibiting loading of, 711
  library, 653
  linking, 652
  loading specified, 721
  object, 651, 659
Monitor, (LINK-10)
  loading the, 811
Monitor commands, 233, (MACRO)
  summary, 307, 308
Monitor error handling, 612, (MONITOR CALLS)
Monitor examination, (MONITOR CALLS)
  GETTAB, 417
  PEEK, 416
  POKE, 417
  SPY, 416
Monitor generated buffers, 468, (MONITOR CALLS)
Monitor I/O facilities, 209, (MACRO)
Monitor programming, 105, (SYSTEM REF.)
Monitor symbols, naming of, 383, (MONITOR CALLS)
Monitor UUOs, 369 (MONITOR CALLS)
  restrictions in reentrant programs, 382
MONRT., 386, (MONITOR CALLS)
MOVE, 33, (SYSTEM REF.)

MOVM, 34, (SYSTEM REF.)
MOVS, 33, (SYSTEM REF.)
/MPSORT, 688, 731, (LINK-10)
MQ, 7, (SYSTEM REF.)
MSTIME UUO, 373, 415, (MONITOR CALLS)
/MTAPE, 688, 732, (LINK-10)
MTAPE UUO, 371, 505, 546, (MONITOR CALLS)
MTCHR.UUO, 380, 507, (MONITOR CALLS)
MUL, 50, (SYSTEM REF.)
Multiprogram assemblies, 244, (MACRO)

/N, 688, (LINK-10)
Name, (LINK-10)
  device, 660, 677
  program, 749
Name block, type 6, 287, (MACRO)
Naming of monitor symbols, 383, (MONITOR CALLS)
Naming programs, 230, (MACRO)
Nesting of, (MACRO)
  expressions, 220
  literals, 225
  macros, 266
Nested subroutines, 84, (SYSTEM REF.)
/NEW, 667, (LINK-10)
NEW disk area, 667, (LINK-10)
Nine-channel magtape, 507, (MONITOR CALLS)
/NOINITIAL, 688, 734, (LINK-10)
/NOLOCAL, 688, 736, (LINK-10)
Nondirectory devices, 462, 493, (MONITOR CALLS)
Nonexistent Memory, 102,(SYSTEM REF.)
Non-I/O UUOs, 385, (MONITOR CALLS)
Nonsharable, 668, (LINK-10)
Non-standard I/O, DECtape, 547, (MONITOR CALLS)
Nonswapping data table, 427, (MONITOR CALLS)
No-ops, 76, (SYSTEM REF.)
Normal block mode, 444, (MONITOR CALLS)
/NOSEARCH, 666, 688, 737, (LINK-10)
/NOSTART, 689, 738, (LINK-10)
NOSYM statement, 250, (MACRO)
/NOSYMBOL, 689, 739, (LINK-10)
/NOSYSLIB, 689, 740, (LINK-10)
Null arguments, 264, (MACRO)
Number, (LINK-10)
  prime, 719
Numbers, naming of, 383, (MONITOR CALLS)
Number sign (#) usage, 242,(MACRO)
Number system, 8, (SYSTEM REF.)
  fixed point, 8
  floating point, 9
Numeric terms, 220, (MACRO)

O (octal radix), 218, (MACRO)
Object modules, 651, 659, (LINK-10)
Object time system, OTS, 742, (LINK-10)
OBUF, 465, (MONITOR CALLS)
Octal codes, summary, 309, (MACRO)
Octal-to-decimal conversion, 87, (SYSTEM REF.)
OCT statement, 234, (MACRO)
Off-line disk unit, 390, (MONITOR CALLS)
Offset, 363, (MONITOR CALLS)
/OLD, 667, (LINK-10)
OLD disk area, 667, (LINK-10)
Once-only disk parameters, 429, (MONITOR CALLS)
One-segment machine, 286, 287, (MACRO)
Ones complement, 8,(SYSTEM REF.)
Op-code table, 213, (MACRO)
OPDEF statement, 254, (MACRO)
OPEN UUO, 370, 463, (MONITOR CALLS)
Operand field, 214, (MACRO)
Operands, 212, (MACRO)
  in primary statements, 206
Operating environment, 919, (DDT)
Operating instructions, 331, (MACRO)
Operating keys, 109, (SYSTEM REF.)
Operating switches, 111, (SYSTEM REF.)
Operation, (SYSTEM REF.)
  clock, 116
  processor, 106
Operator, defined, 214, (MACRO)
Operator field, 214, (MACRO)
Operators, 211, (MACRO)
  arithmetic, 219
  logical, 219
  unary, 222
Optimization, 583, (MONITOR CALLS)
Options, (LINK-10)
  interactive, 683
OR, 43, (SYSTEM REF.)
ORCA, 43, (SYSTEM REF.)
ORCB, 44, (SYSTEM REF.)
ORCM, 44, (SYSTEM REF.)
OTHUSR UUO, 379, 416, (MONITOR CALLS)
/OTS, 689, 742, (LINK-10)
OUTBUF UUO, 370, 468, (MONITOR CALLS)
Output, 653, (LINK-10)
  auxiliary, 651, 654
  defaults, 677
  files, 677, 679, 682, 703, 716
  specification, 676, 703
  switches, 682, 710, 724, 728, 751, 757, 760, 773, 774
Output, 283, (MACRO)
Output, (MONITOR CALLS)
  buffered, 477
  spooling, 604
  unit selection, 561

OUTPUT UUO, 371, 474, (MONITOR CALLS)
OUT UUO, 370, 474, (MONITOR CALLS)
Overdrawn amount, 555, (MONITOR CALLS)
Overflow, 48, 55, 77, 102, (SYSTEM REF.)
Overflow, (LINK-10)
  disk, 729
Overlay facility, 654, 706, (LINK-10)
Owner of files, 555, (MONITOR CALLS)

Page mapped, 359, (MONITOR CALLS)
PAGE statement, 250, (MACRO)
Paper-tape control, 905, 914, (DDT)
Paper-tape input/output at TTY, 534, (MONITOR CALLS)
Paper-tape punch, 512, (MONITOR CALLS)
Paper-tape reader, 513, (MONITOR CALLS)
Parentheses delimiters, 237, (MACRO)
Parity subtable, 438, (MONITOR CALLS)
PASS2 statements, 245, (MACRO)
Passive search list, 562, (MONITOR CALLS)
PAT.., 744, 761, (LINK-10)
Patching space, 689, (LINK-10)
  allocation, 744,
/PATCHSIZE, 689, 744, (LINK-10)
PATH.UUO, 380, 577, (MONITOR CALLS)
PC, 5, (SYSTEM REF.)
PC word, 76, (SYSTEM REF.)
PEEK UUO, 374, 416, (MONITOR CALLS)
Percent sign (%) symbol, 262, (MACRO)
Performing magnetic tape functions, 732, (LINK-10)
Period (.) symbol, see point symbol
Permanent switches, 662, 681, (LINK-10)
PHASE statement, 299 (MACRO)
Physical address, 360, (MONITOR CALLS)
Physical controller names, 560, (MONITOR CALLS)
Physical-only bit (UU.PHS), 382, 462, (MONITOR CALLS)
Physical-only I/O, 382, 462, (MONITOR CALLS)
Physical unit names, 560, (MONITOR CALLS)
PI, 95, 98, (SYSTEM REF.)
PI ON, 108, (SYSTEM REF.)
PJOB UUO, 374, 416, (MONITOR CALLS)
Plotter, 515, (MONITOR CALLS)
POINT statement, 237, (MACRO)
Point (.) symbol, 224, (MACRO)
Pointer, (SYSTEM REF.)
  byte, 37
  IO block, 92
Points, (LINK-10)
  entry, 706, 713, 746
POKE.UUO, 380, 417, (MONITOR CALLS)
POP, 35, (SYSTEM REF.)
POPJ, 85, (SYSTEM REF.)
Position-done interrupt, 611, (MONITOR CALLS)
Pound sign , double (##), 247, (MACRO)
Preserved files, 556, (MONITOR CALLS)
PRGEND pseudo-op, 244, (MACRO)
Primary instruction format, 207, (MACRO)
Primary statement operands, 206, (MACRO)
Prime number, 719, (LINK-10)
PRINTX MESSAGE statement, 251, (MACRO)
Priority interrupt, 95, (SYSTEM REF.)
Priority interrupt routines, 613, (MONITOR CALLS)
Privilege word (.GTPRV), 423, (MONITOR CALLS)
Proceed counter, 894, 912, (DDT)
Processing of statement, 213, (MACRO)
Processor conditions, 100, (SYSTEM REF.)
Processor, (MONITOR CALLS)
  constants table, 433
  flags, 387
  modes, 367
  variable table, 434
Program (MACRO)
  break, 284
  line, 213
  listing, 249
  listing output, 283
  origin, 231
  subtitles, 231
Program and profile identification, 410, (MONITOR CALLS)
  LOCATE, 412
  SETNAM, 410
  SETUUO, 410
Program control, 76, 163, (SYSTEM REF.)
Program examples, 293, (MACRO)
Programmed operator, 223, (MACRO)
Programmed operators, 368, (MONITOR CALLS)
  tables, 369, 372
Programming conventions, 14, (SYSTEM REF.)
Program name, 749, (LINK-10)
Programs, (LINK-10)
  COMPIL, 656
  debugging, 660, 686, 690, 701, 766
  exec mode, 734

Programs (Cont.)
  loading COBOL, 803, 805
  loading debugging, 701, 766
  loading MACRO, 804
Program Stop, 108, (SYSTEM REF.)
Program version number, 364, (MONITOR CALLS)
Project-programmer number word, 419, (MONITOR CALLS)
Protection, 103, (SYSTEM REF.)
Protection, 554, (MONITOR CALLS)
Protection address, 359, (MONITOR CALLS)
Protection and relocation, 359, (MONITOR CALLS)
Pseudo-operation code, 211, (MACRO)
Pseudo-ops, 227, (MACRO)
  summary, 311, 312
Pseudo-TTY, 516, (MONITOR CALLS)
Pure code, 753, (LINK-10)
Pure segment, 359, (MONITOR CALLS)
PURGE statement, 245, (MACRO)
PUSH, 35, (SYSTEM REF.)
Pushdown list, 34, 163, (SYSTEM REF.)
  defined, 35
  subroutines, 85
Pushdown overflow, 102, (SYSTEM REF.)
PUSHJ, 84, (SYSTEM REF.)

Qualifier ↑L, 218, (MACRO)
Quantum time, 606, (MONITOR CALLS)
Queuing strategy, 610, (MONITOR CALLS)
Quotas, disk, 391, 412, 559, (MONITOR CALLS)
Quotation mark, single ('), 249, 264, (MACRO)

R LINK command, 656, 657, 675 (LINK-10)
Radix changing, 889, 907, (DDT)
Radix changing, 218, 235, (MACRO)
Radix 50 character set, 210, (MACRO)
Radix 50 representation, 325, (MACRO)
Radix 50 representation, 759, (LINK-10)
RADIX statements, 233, 236, (MACRO)
Read in, 109, (SYSTEM REF.)
Reading directory paths, 577, (MONITOR CALLS)
Readin mode, 94, (SYSTEM REF.)
Real-time programming, 444, (MONITOR CALLS)
  HPQ, 457
  RTTRP, 444
  TRPSET, 455
  UJEN, 457
Real-time trapping, 444, (MONITOR CALLS)
  data block, 446
  dismissing interrupt, 449
  examples, 449, 454
  exec mode, 453
  interrupt level use, 447
  removing devices, 449
  restrictions, 448
  returns, 447
REASSI UUO, 373, 484, (MONITOR CALLS)
Recommended hashsize, 719, (LINK-10)
Redefinition of macros, 266, (MACRO)
Reentrant program, 359, (MONITOR CALLS)
Relative address, 360, (MONITOR CALLS)
RELEAS UUO, 371, 483, (MONITOR CALLS)
RELOC pseudo-op, 227, (MACRO)
Relocatability of an expression, 279, (MACRO)
Relocatable address mode, 227, (MACRO)
Relocatable code, 651, (LINK-10)
Relocatable object program, 279, (MACRO)
Relocatable symbols, 694, 695, (LINK-10)
Relocation, 103, (SYSTEM REF.)
Relocation address, 359, (MONITOR CALLS)
Relocation and protection word, 419, (MONITOR CALLS)
Relocation counters, 686, 689, 698, 754, (LINK-10)
  listing, 698
REMAP UUO, 375, 408, (MONITOR CALLS)
REMARK COMMENTS statement, 251, (MACRO)
Remembering arguments on RUN, 405, (MONITOR CALLS)
Removing devices from PI channel, 449, (MONITOR CALLS)
RENAME UUO, 370, 472, (MONITOR CALLS)
  DECtape, 543
  disk, 568, 571
  error codes, 635
REPEAT statement, 253, (MACRO)
Representation, 759, (LINK-10)
Requests, global, 667, 746, (LINK-10)
  undefined, 752, 763, 767
/REQUIRE, 689, 746, (LINK-10)
RESDV.UUO, 381, 483, (MONITOR CALLS)
Reserving a single location, 242, (MACRO)
RESET UUO, 370, 461, 587, (MONITOR CALLS)
Response subtable, 436, (MONITOR CALLS)
Restore, 81, (SYSTEM REF.)
Restricted devices, 462, (MONITOR CALLS)
Restrictions on monitor UUO's, 382, (MONITOR CALLS)
Restrictions on real-time trapping, 448, (MONITOR CALLS)

Retrieval information block, 554, (MONITOR CALLS)
Retrieval pointers, 641, (MONITOR CALLS)
RETURN key, 675, (LINK-10)
Reverse slash (\), 267, (MACRO)
/REWIND, 689, 747, (LINK-10)
Rewinding tapes, 747, 768, (LINK-10)
RIB, 554, (MONITOR CALLS)
RIM format, 290, (MACRO)
RIM10 format, 289, (MACRO)
RIM10B format, 292, (MACRO)
Ring buffers, 466, (MONITOR CALLS)
ROT, 47, (SYSTEM REF.)
Rotate, 46, 163, (SYSTEM REF.)
ROTC, 48, (SYSTEM REF.)
Rounding, 56 (SYSTEM REF.)
RSW, 95, (SYSTEM REF.)
RTTRP UUO, 379, 444, (MONITOR CALLS)
RUN, 107, (SYSTEM REF.)
RUN command, 654, (LINK-10)
/RUNAME, 689, 749, (LINK-10)
/RUNCOR, 689, 748, (LINK-10)
RUNTIM UUO, 374, 416, (MONITOR CALLS)
RUN UUO, 374, 403, (MONITOR CALLS)

/S, 689, (LINK-10)
SALL statement, 250, (MACRO)
SAT blocks, 550, (MONITOR CALLS)
/SAVE, 689, 750, (LINK-10)
SAVE command, 659, 668, (LINK-10)
Save file, 689, 750, (LINK-10)
  creating, 805
  defining, 757
Saving XPN file, 772, (LINK-10)
Scaling, 55, (SYSTEM REF.)
Scan switch, 577, (MONITOR CALLS)
Scheduling, 605, (MONITOR CALLS)
Searches, 895, 912, (DDT)
/SEARCH, 666, 689, 752, (LINK-10)
Search list, job, 562, (MONITOR CALLS)
Search list, 667, (LINK-10)
Search mode, 721, (LINK-10)
  library, 666, 688, 689, 737, 752
  surpressing library, 737
SEARCH name, 258, (MACRO)
SEARCH pseudo-op, 258, (MACRO)
Search symbols, (LINK-10)
  library, 686, 706, 746
Searching, (LINK-10)
  inhibiting, 740
  system library, 718
Searching, 609, (MONITOR CALLS)
Searching libraries, 763, (LINK-10)
SEEK UUO, 337, 587, (MONITOR CALLS)
Segment control, 403, (MONITOR CALLS)
  GETSEG, 407
  modifying shared segments, 409
  REMAP, 408
Segment Control (Cont.)
  RUN, 403
  testing, 408
/SEGMENT, 689, 753, (LINK-10)
Segment size, (LINK-10)
  initial low, 697
  maximum low, 729
Segments, 359, (MONITOR CALLS)
/SELF, 667, (LINK-10)
Semicolon (;) as terminator, 212, 328, (MACRO)
Semi-standard I/O, DECtape, 547, (MONITOR CALLS)
Sequence of operations, RUN, 406, (MONITOR CALLS)
/SET, 689, 754, (LINK-10)
SETA, 40, (SYSTEM REF.)
SETCA, 41, (SYSTEM REF.)
SETCM, 41, (SYSTEM REF.)
SETDDT UUO, 372, 385, (MONITOR CALLS)
SETM, 41, (SYSTEM REF.)
SETNAM UUO, 375, 410, (MONITOR CALLS)
SETO, 40, (SYSTEM REF.)
SETPOV UUO, 374, (MONITOR CALLS)
SETSTS UUO, 370, 480, (MONITOR CALLS)
Setting directory paths, 577, (MONITOR CALLS)
Setting disk priority, 599, (MONITOR CALLS)
Setting job parameters, 410, (MONITOR CALLS)
Setting logical names, 484, (MONITOR CALLS)
Setting write-up bit, 403, (MONITOR CALLS)
SETUUO, 378, 410, (MONITOR CALLS)
SETUWP UUO, 375, 403, (MONITOR CALLS)
SETZ, 40, (SYSTEM REF.)
/SEVERITY, 689, 775, 776, (LINK-10)
Severity levels, 775, 776, (LINK-10)
SFD, 551, (MONITOR CALLS)
SFD privileges, 556, (MONITOR CALLS)
Sharable, 668, 757, (LINK-10)
Shift, 163, (SYSTEM REF.)
  arithmetic, 53
  logical, 46
Shifting, binary, 221, (MACRO)
Simultaneous access, 560, (MONITOR CALLS)
Simultaneous supersede/update, 600, (MONITOR CALLS)
Single mode, 444, (MONITOR CALLS)
Single quote mark ('), 263, (MACROO
SIXBIT statement, 240, (MACRO)
Size, (LINK-10)
  initial low segment, 697
  initial symbol table, 719
  maximum low segment, 729
/SKIP, 689, 756, (LINK-10)
SKIP, 66, (SYSTEM REF.)
SLEEP UUO, 374, 391, (MONITOR CALLS

Soft error, 612, (MONITOR CALLS)
Software detected errors, 613, (MONITOR CALLS)
SOJ, 67, (SYSTEM REF.)
Sorting symbol table, 731, (LINK-10)
SOS, 68, (SYSTEM REF.)
Space, (LINK-10)
  allocating disk, 709
  allocating patching, 744
  patching, 689
Spacing forward tapes, 756, (LINK-10)
Special symbols, 903, 910, (DDT)
Specification, (LINK-10)
  file, 660, 676, 703
  switch, 663, 686
Specification switches, (LINK-10)
  implicit file, 702, 714, 715, 764, 766
Specifications, Input/Output, 676, 703, (LINK-10)
Specified modules, loading of, 721, (LINK-10)
Specifying, (LINK-10)
  central processor, 700
  disk areas, 667
  execution, 701, 713, 718, 766
  fatality of messages, 755
  free core, 716
  log file, 723
  map file, 727
  start address, 758
Spool bits, 411, 432, (MONITOR CALLS)
Spooling, 603, (MONITOR CALLS)
SPY UUO, 375, 416, (MONITOR CALLS)
Square bracket ([]), used as terminator, 226,(MACRO)
SQUOZE mnemonic, 236, (MACRO)
/SSAVE, 689, 757, (LINK-10)
SSAVE command, 659, 668, (LINK-10)
Standard listing operations, 250, (MACRO)
/START, 689, 758, (LINK-10)
Start address, 666, 677, 689, 713, (LINK-10)
  ignoring, 738
  specifying, 758
START command, 654, 659, (LINK-10)
Starting address, block type 7, 288, (MACRO)
Starting a program, 385, (MONITOR CALLS)
Starting the program, 876, 883, 912, (DDT)
State codes, 431, (MONITOR CALLS)
Statement, 209, (MACRO)
  elements, 206
  evaluation, 214
  formats, 249
  label, 206
  processing, 213
  general form of, 206
  operands in primary, 206
Statement, (LINK-10)
  entry, 706, 746
  function, 706
  subroutine, 706
STATO UUO, 370, 480, (MONITOR CALLS)
Status changing switches, 684, (LINK-10)
Status, 93, (SYSTEM REF.)
  clock, 114
  interrupt, 99
  processor, 101
Status checking and setting, 479, 480, (MONITOR CALLS)
Status word, 419, (MONITOR CALLS)
STATZ UUO, 370, 480, (MONITOR CALLS)
Stopping a program, 385, (MONITOR CALLS)
Storage allocation, 319, (MACRO)
Storage allocation table, 550, (MONITOR CALLS)
Storage map for user mode, 917, (DDT)
Storage, reserving, 242, (MACRO)
Storage words, (DDT)
  examining, 871, 879, 908
  modifying, 872, 880
String, (LINK-10)
  command, 675, 676, 681
Structure of disk files, 549, (MONITOR CALLS)
STRUUO, 376, 588, (MONITOR CALLS)
SUB, 49, (SYSTEM REF.)
Sub-file directories, 551, 577, (MONITOR CALLS)
Subroutines, 82, (SYSTEM REF.)
SUBROUTINE statement, 706, (LINK-10)
Subtables, GETTAB, 436, (MONITOR CALLS)
SUBTTL pseudo-op, 231, (MACRO)
Suffix ##, 216, (MACRO)
Suffixes K, M, and G, 219, (MACRO)
Superseding a file, 563, (MONITOR CALLS)
Superseding a sharable program, 409, (MONITOR CALLS)
Super-USETI/USETO, 587, (MONITOR CALLS)
Suppression of logical device names, 382, (MONITOR CALLS)
SUPPRESS pseudo-op, 248, (MACRO)
Suppression of, (LINK-10)
  library search mode, 737
  messages, 708, 725,
  symbol table, 739
Suspending, 391, (MONITOR CALLS)
Swapping, 607, (MONITOR CALLS)
Swapping classes, 608, (MONITOR CALLS)
Swapping data table, 429, (MONITOR CALLS)
Swapping parameter word, 419, (MONITOR CALLS)

Switch, (LINK-10)
  exit condition, 676
Switch algorithms, 679, (LINK-10)
Switch options, summary, 335, (MACRO)
Switch specification, 663, 686, (LINK-10)
Switches, 111,(SYSTEM REF.)
Switches, (LINK-10)
  COMPIL, 662, 664
  delayed action, 684, 695, 700, 708, 713, 717, 720, 726, 730, 731, 741, 743, 745, 748, 749, 755, 762, 765, 771
  device, 680, 691, 733, 747, 756, 768
  file dependent, 681, 712, 721, 722, 736, 737, 738, 752, 753, 758
  immediate action, 693, 697, 698, 703, 706, 718, 735, 739, 746, 754, 769
  implicit file specification, 684, 702, 714, 715, 764, 766
  LINK-10, 663, 685, 686, 691
  LOADER, 663, 846
  output, 682, 710, 724, 728, 751, 757, 760, 773, 774
  permanent, 662, 681
  status changing, 684
  temporary, 662, 681
  type-out, 683
SWITCH UUO, 373, 442, (MONITOR CALLS)
/SYMBOL, 689, 759, (LINK-10)
Symbol:! (suppressed local symbol), 218, (MACRO)
Symbol;:!(suppressed internal symbol), 218 (MACRO)
Symbol delimiter, 210, (MACRO)
Symbol evaluation, 902, (DDT)
Symbol expression, 248, (MACRO)
Symbol file, 689, 690, (LINK-10)
  defining, 759
Symbolic, (MACRO)
  addresses, 210, 212
  elements, 280
    operands, 212
    operators, 211
Symbols, 873, 883, (DDT)
  defining, 899, 910
  deleting, 900, 910
Symbols, 652, (LINK-10)
  absolute, 695
  COMMON, 695
  entry, 695
  entryname, 695
  external, 692
  global, 694, 695, 769
  internal, 692
  library search, 686, 706, 746
  linking, 653
  loading, 761
Symbols, (Cont.)
  local, 665, 688, 695, 701, 722, 736, 759
  relocatable, 694, 695
  typing in global, 769
  undefined, 656, 681, 767
  undefined global, 705
  zero length, 695
Symbols, 209, (MACRO)
  block type 2, 286
  block type 3, 286
  generated, 262
  global, 216
  local, 216
Symbol table, 660, 679, 761, (LINK-10)
  arranging, 765
  clearing initial, 734
  global, 688
  initial global, 734
  initial size, 719
  sorting, 731
  suppressing, 739
/SYMSEG, 690, 761, (LINK-10)
Synchronization of buffered I/O 478, (MONITOR CALLS)
SYN statement, 255, (MACRO)
SYS device, 554, (MONITOR CALLS)
SYS disk area, 667, (LINK-10)
/SYSLIB, 690, 763, (LINK-10)
/SYSORT, 690, 765, (LINK-10)
SYSPHY UUO, 376, 593, (MONITOR CALLS)
SYSSTR UUO, 376, 592, (MONITOR CALLS)
System, Object Time (OTS), (LINK-10)
  loading, 742
System libraries, 653, 666, 667, 689 690, 740, 752, 763
  searching, 718
System library, 554, (MONITOR CALLS)

Table, (LINK-10)
  arranging symbol, 765
  clearing initial symbol, 734
  global symbol, 688
  hash, 719
  initial global symbol, 734
  size of initial symbol, 719
  sorting symbol, 731
  suppressing symbol, 739
  symbol, 660, 679, 731, 761, 765
Tape functions, (LINK-10)
  performing magnetic, 732
TAPE statement, 251, (MACRO)
Tapes, (LINK-10)
  backspacing, 691
  rewinding, 747 768
  spacing forward, 756
  unloading, 768
TDC, 73, (SYSTEM REF.)

TDN, 72, (SYSTEM REF.)
TDO, 73, (SYSTEM REF.)
TDZ, 73, (SYSTEM REF.)
Temporary files, 405, 412, (MONITOR CALLS)
Temporary switches, 662, 681, (LINK-10)
Terminating loading, 718, (LINK-10)
Terminals, 521, (MONITOR CALLS)
Terminator, (MACRO)
  colon (:), 210
  right square bracket (]), 226
  semicolon (;), 212
Terms, numeric, 220, (MACRO)
/TEST, 690, 766, (LINK-10)
Testing of sharable segments, 408, (MONITOR CALLS)
Test instructions, (SYSTEM REF.)
  arithmetic, 63, 163
  logical, 70, 165
Text codes, summary, 323, (MACRO)
Text input, 240, (MACRO)
Time limit exceeded, 391, (MONITOR CALLS)
TIMER UUO, 373, 415, (MONITOR CALLS)
Time sharing, 103, (SYSTEM REF.)
Timing, 23, 178, (SYSTEM REF.)
  clock, 115
  interrupt, 99
Timing information, 414, (MONITOR CALLS)
  DATE, 415
  MSTIME, 415
  TIMER, 415
TITLE pseudo-op, 230, (MACRO
TLC, 72, (SYSTEM REF.)
TLN, 71, (SYSTEM REF.)
TLO, 72, (SYSTEM REF.)
TLZ, 71, (SYSTEM REF.)
TMPCOR UUO, 375, 412, (MONITOR CALLS)
Total user core, 393, (MONITOR CALLS)
Transfer-done interrupt, 611, (MONITOR CALLS)
Transferring program control, 403, (MONITOR CALLS)
Trapping, 387, (MONITOR CALLS)
Traps, 13, (SYSTEM REF.)
TRC, 70, (SYSTEM REF.)
Triplet format, 759, (LINK-10)
TRMNO. UUO, 381, 529, (MONITOR CALLS)
TRMOP. UUO,381, 530, (MONITOR CALLS)
TRN, 70, (SYSTEM REF.)
TRO, 71, (SYSTEM REF.)
TRPJEN UUO, 374, (MONITOR CALLS)
TRPSET UUO, 374, 455, (MONITOR CALLS)
TRZ, 70, (SYSTEM REF.)
TSC, 74, (SYSTEM REF.)
TSN, 74, (SYSTEM REF.)
TSO, 75, (SYSTEM REF.)
TSZ, 74, (SYSTEM REF.)
TTCALL UUO, 370, 525, (MONITOR CALLS)
Type-in modes, 873, 909, (DDT)
Type-out modes, 871, 883, 889, 907, (DDT)
Type-out switches, 683, (LINK-10)
Typing errors, 876, 886, (DDT)
Typing in, 884, 909, (DDT)
  arithmetic expressions, 886, 911
  numbers, 885, 909
  symbolic instructions, 885, 909
  text characters, 885, 910
Typing, (LINK-10)
  global symbols, 769
  library search symbols, 706
  undefined globals, 767
Two half-words of data, entering, 239, (MACRO)
Twos complement, 8, (SYSTEM REF.)
Two-segment machine, 286, (MACRO)
TWOSEG pseudo-op, 232, 286, (MACRO)

/U, 690, (LINK-10)
UFA, 59, (SYSTEM REF.)
UFD, 551, (MONITOR CALLS)
UFD privileges, 556, (MONITOR CALLS)
UGETF UUO, 371, 545, (MONITOR CALLS)
UJEN UUO, 371, 457, (MONITOR CALLS)
Unary operators, 222, (MACRO)
Unbuffered data modes, 465, 475, (MONITOR CALLS)
/UNDEFINED, 690, 767, (LINK-10)
Undefined, (LINK-10)
  global requests, 752, 763, 767
  global symbols, 705,
  typing of globals, 767
  undefined symbols, 656, 681, 767
Unimplemented op codes, 383, 385, (MONITOR CALLS)
Unimplemented operations, 86, (SYSTEM REF.)
Unit of core allocation, 360, (MONITOR CALLS)
Unit selection on output, 561, (MONITOR CALLS)
Unit state codes, 610, (MONITOR CALLS)
Universal date-time standard, 415, (MONITOR CALLS)
Universal I/O index, 530, (MONITOR CALLS)
UNIVERSAL name, 257, (MACRO)
UNIVERSAL pseudo-op, 257, (MACRO)
.UNIV symbol, 258, (MACRO)
/UNLOAD, 690, 768, (LINK-10)
Unloading tapes, 768, (LINK-10)
Unlocking jobs, 397, (MONITOR CALLS)
UNLOK.UUO, 381, 397, (MONITOR CALLS)

Unrestricted devices, 462, (MONITOR CALLS)
Updating a file, 564, (MONITOR CALLS
Upper and lower case, 887, (DDT)
Use of MTAPE operator, 506, (MONITOR CALLS)
User, 77, (SYSTEM REF.)
User address mapping, 361, (MONITOR CALLS)
User core storage, 360, (MONITOR CALLS)
User-defined operator, 245, (MACRO)
User-defined symbols, 215, (MACRO)
User file directories, 551, (MONITOR CALLS)
User generated buffers, 469, (MONITOR CALLS)
User in-out, 78, 102, (SYSTEM REF.)
User I/O mode, 367, (MONITOR CALLS)
User library, 667, (LINK-10)
User mode, 359, 367, (MONITOR CALLS)
User mode, 108, (SYSTEM REF.)
User programming, 104, (SYSTEM REF.)
User programming, introduction, 367, (MONITOR CALLS)
User programming, disk, 563, (MONITOR CALLS)
User program name word, 419, (MONITOR CALLS)
User symbol table, 213, (MACRO)
User UUOs, 369, (MONITOR CALLS)
User virtual address space, 652, (LINK-10)
USETI UUO, 371, 545, 584, (MONITOR CALLS)
USETO UUO, 371, 545, 584, (MONITOR CALLS)
Using, (LINK-10)
  automatically, 656, 659
  directly, 657, 675
UTPCLR, 373, 546, (MONITOR CALLS)
UUO, 91, (SYSTEM REF.)
UUOs, 368, (MONITOR CALLS)
  APRENB, 387
  CHKACC, 588
  CLOSE, 481
  CORE, 401
  CTLJOB, 520
  DAEMON, 442
  DATE, 415
  DDTIN, 525
  DDTOUT, 525
  DEVCHR, 488
  DEVLNM, 484
  DEVNAM, 491
  DEVPPN, 593
  DEVSIZ, 490
  DEVSTS, 487
  DEVTYP, 489
  DISK., 599
  DSKCHR, 597

UUOs, (Cont.)
  ENTER, 471, 542, 565
  EXIT, 386
  GETLIN, 529
  GETPPN, 416
  GETSEG, 407
  GETSTS, 480
  GETTAB, 417
  GOBSTR, 591
  HIBER, 391
  HPQ, 457
  IN, 474
  INBUF, 468
  INIT, 465
  INPUT, 474
  JOBSTR, 590
  JOBSTS, 519
  LIGHTS, 442
  LOCATE, 412
  LOCK, 394
  LOOKUP, 541, 567, 470
  METER., 458
  MSTIME, 415
  MTAPE, 505, 546
  MTCHR., 507
  OPEN, 463
  OTHUSR, 416
  OUT, 474
  OUTBUF, 468
  OUTPUT, 474
  PATH., 577
  PEEK, 416
  PJOB, 416
  POKE., 417
  REASSIGN, 484
  RELEASE, 483
  REMAP, 408
  RENAME, 472, 543, 568
  RESDV., 483
  RESET, 461, 587
  RTTRP, 444
  RUN, 403
  RUNTIM, 416
  SEEK, 587
  SETDDT, 385
  SETNAM, 410
  SETSTS, 480
  SETUUO, 410
  SETUWP, 403
  SLEEP, 391
  SPY, 416
  STATO, 480
  STATZ, 480
  STRUUO, 588
  SWITCH, 442
  SYSPHY, 593
  SYSSTR, 592
  TIMER, 415
  TMPCOR, 412
  TRMNO., 529
  TRMOP., 530

UUOs, (Cont.)
TRPSET, 455
TTCALL, 525
UGETF, 545
UJEN, 457
UNLOK, 397
USETI, 545, 584
USETO, 545, 584
UTPCLR, 546
WAIT, 478
WAKE, 392
WHERE, 491
UU.PHS, 382, 462, (MONITOR CALLS)

/VALUE, 769, (LINK-10)
Values, (LINK-10)
assigning, 705
initial default, 677, 678, 686
VAR statements, 243, (MACRO)
Variables tables, 434, (MONITOR CALLS)
Verbosity, 770, 771, (LINK-10)
/VERBOSITY, 690, 770, 775, 777, (LINK-10)
Verification, 609, (MONITOR CALLS)
Version number, 364, (MONITOR CALLS)
Vestigial job data area, 365, (MONITOR CALLS)
Virtual address, 360, (MONITOR CALLS)

WAIT UUO, 372, 478, (MONITOR CALLS)
Wake-enable bits, 392, (MONITOR CALLS)
WAKE UUO, 378, 392, (MONITOR CALLS)
WATCH bits, 411, 432, (MONITOR CALLS)
WHERE UUO, 377, 491, (MONITOR CALLS)
Word format, 193, (SYSTEM REF.)
Write protect bit, 403, 409, (SYSTEM REF.)
Writing reentrant user programs, 623, (MONITOR CALLS)

X, 11, (SYSTEM REF.)
XALL statement, 250, (MACRO
XCT, 78, (SYSTEM REF.)
XLIST statement, 250, (MACRO)
XOR, 44, (SYSTEM REF.)
/XPN, 690, 772, (LINK-10)
XPN file, creating and saving, 772, (LINK-10)
XPUNGE pseudo-op statement, 245, (MACRO)
XWD, 475, (MONITOR CALLS)
XWD statement, 239, (MACRO)

Y, 11, (SYSTEM REF.)

Z statement, 236, (MACRO)
/ZERO, 690, 774, (LINK-10)
Zero length symbols, 695, (LINK-10)
Zero-compression, 750, (LINK-10)
Zero word, 236, (MACRO)

DIGITAL EQUIPMENT CORPORATION digital WORLD-WIDE SALES AND SERVICE

MAIN OFFICE AND PLANT

146 Main Street, Maynard, Massachusetts, U.S.A. 01754 • Telephone: From Metropolitan Boston: 646-8600 • Elsewhere: (617)-897-5111
TWX: 710-347-0212 Cable: DIGITAL MAYN Telex: 94-8457

UNITED STATES

NORTHEAST

REGIONAL OFFICE
275 Wyman Street, Waltham, Massachusetts 02154
Telephone: (617)-890-0320/0330 TWX: 710-324-6919

WALTHAM
15 Lunda Street, Waltham, Massachusetts 02154
Telephone: (617)-891-1030 TWX: 710-324-6919

CAMBRIDGE/BOSTON
899 Main Street, Cambridge, Massachusetts 02139
Telephone: (617)-491-6130 TWX: 710-320-1167

ROCHESTER
130 Allens Creek Road, Rochester, New York 14618
Telephone: (716)-461-1700 TWX: 710-253-3078

CONNECTICUT
240 Pomeroy Ave., Meriden, Conn. 06450
Telephone: (203)-237-8441/7466 TWX: 710-461-0054

MID-ATLANTIC — SOUTHEAST

REGIONAL OFFICE:
U.S. Route 1, Princeton, New Jersey 08540
Telephone: (609)-452-2940 TWX: 510-685-2338

NEW YORK
95 Cedar Lane, Englewood, New Jersey 07631
Telephone: (201)-871-4984, (212)-594-6955, (212)-736-0447
TWX: 710-991-9721

NEW JERSEY
1259 Route 46, Parsippany, New Jersey 07054
Telephone: (201)-335-3300 TWX: 710-987-8319

PRINCETON
U.S. Route 1
Princeton, New Jersey 08540
Telephone: (609) 452-2940 TWX: 510-685-2338

LONG ISLAND
1 Huntington Quadrangle
Suite 1SO7 Huntington Station, New York 11746
Telephone: (516)-694-4131, (212)-895-8095

PHILADELPHIA
Station Square Three, Paoli, Pennsylvania 19301
Telephone: (215)-647-4900/4410 Telex: 510-668-8395

WASHINGTON
Executive Building
6811 Kenilworth Ave., Riverdale, Maryland 20840
Telephone: (301)-779-1600/752-8797 TWX: 710-826-9662

DURHAM/CHAPEL HILL
2704 Chapel Hill Boulevard
Durham, North Carolina 27707
Telephone: (919)-489-3347 TWX: 510-927-0912

ORLANDO
Suite 130, 7001 Lake Ellenor Drive, Orlando, Florida 32809
Telephone: (305)-851-4450 TWX: 810-850-0180

ATLANTA
2815 Clearview Place, Suite 100,
Atlanta, Georgia 30340
Telephone: (404)-451-3734/3735/3736 TWX: 810-757-4223

MID-ATLANTIC — SOUTHEAST (cont.)

KNOXVILLE
6311 Kingston Pike, Suite 21E
Knoxville, Tennessee 37919
Telephone: (615)-588-6571 TWX: 810-583-0123

CENTRAL

REGIONAL OFFICE:
1850 Frontage Road, Northbrook, Illinois 60062
Telephone: (312)-498-2500 TWX: 910-686-0655

PITTSBURGH
400 Penn Center Boulevard
Pittsburgh, Pennsylvania 15235
Telephone: (412)-243-9404 TWX: 710-797-3657

CHICAGO
1850 Frontage Road, Northbrook, Illinois 60062
Telephone: (312)-498-2500 TWX: 910-686-0655

ANN ARBOR
230 Huron View Boulevard, Ann Arbor, Michigan 48103
Telephone: (313)-761-1150 TWX: 810-223-6053

INDIANAPOLIS
21 Beachway Drive — Suite G
Indianapolis, Indiana 46224
Telephone: (317)-243-8341 TWX: 810-341-3436

MINNEAPOLIS
Suite 111, 8030 Cedar Avenue South,
Minneapolis, Minnesota 55420
Telephone: (612)-854-6562-3-4-5 TWX: 910-576-2818

CLEVELAND
Park Hill Bldg., 35104 Euclid Ave.
Willoughby, Ohio 44094
Telephone: (216)-946-8484 TWX: 810-427-2608

ST. LOUIS
Suite 110, 115 Progress Pky., Maryland Heights,
Missouri 63043
Telephone: (314)-878-4310 TWX: 910-764-0831

DAYTON
3101 Kettering Blvd., Dayton, Ohio 45439
Telephone: (513)-299-7377 TWX: 810-459-1676

MILWAUKEE
8531 W. Capitol Drive, Milwaukee, Wisconsin 53222
Telephone: (414)-463-9110 TWX: 910-262-1199

DALLAS
8855 North Stemmons Freeway
Dallas, Texas 75247
Telephone: (214)-638-4880 TWX: 910-861-4000

HOUSTON
3417 Milam Street, Suite A, Houston, Texas 77002
Telephone: (713)-524-2961 TWX: 910-881-1651

CENTRAL (cont.)

NEW ORLEANS
3100 Ridgelake Drive, Suite 108
Metairie, Louisiana 70002
Telephone: 504-837-0257

WEST

REGIONAL OFFICE
310 Soquel Way, Sunnyvale, California 94086
Telephone: (408)-735-9200

ANAHEIM
801 E. Ball Road, Anaheim, California 92805
Telephone: (714)-776-6932/8730 TWX: 910-591-1189

WEST LOS ANGELES
1510 Cotner Avenue, Los Angeles, Californa 90025
Telephone: (213)-479-3791/4318 TWX: 910-342-6999

SAN DIEGO
3444 Hancock Street
San Diego, California 92110
Telephone: (714)-298-0591, 0593 TWX: 910-335-1230

SAN FRANCISCO
1400 Terra Bella
Mountain View, California 94040
Telephone: (415)-964-6200 TWX: 910-373-1266

PALO ALTO
560 San Antonio Rd., Palo Alto, California 94306
Telephone: (415)-969-6200 TWX: 910-373-1266

OAKLAND
7850 Edgewater Drive
Oakland, California 94621
Telephone: (415)-635-5453/7830 TWX: 910-366-7238

ALBUQUERQUE
6303 Indian School Road, N.E.
Albuquerque, N.M. 87110
Telephone: (505)-296-5411/5428 TWX: 910-989-0614

DENVER
2305 South Colorado Blvd., Suite #5
Denver, Colorado 80222
Telephone: (303)-757-3332/758-1656/758-1659
TWX: 910-931-2650

SEATTLE
1521 130th N.E., Bellevue, Washington 98005
Telephone: (206)-454-4058/455-5404 TWX: 910-443-2306

SALT LAKE CITY
431 South 3rd East, Salt Lake City, Utah 84111
Telephone: (801)-328-9838 TWX: 910-925-5834

PHOENIX
4358 East Broadway Road
Phoenix, Arizona 85040
Telephone: (602) 268-3488 TWX: 910-950-4691

PORTLAND
Suite 168
5319 S.W. Canyon Court, Portland, Ore. 97221
Telephone: (503) 297-3761/3765

INTERNATIONAL

EUROPEAN HEADQUARTERS

Digital Equipment Corporation International Europe
81 Route de l'Aire
1211 Geneva 26, Switzerland
Telephone: 42 79 50 Telex: 22 683

FRANCE

Equipment Digital S.A.R.L.

PARIS
327 Rue de Charenton, 75 Paris 12 ème, France
Telephone: 344-76-07 Telex: 21339

GRENOBLE
10 rue Auguste Ravier, F-38 Grenoble, France
Telephone: (76) 87 87 32 Telex: 32 882 F (Code 212)

GERMANY

Digital Equipment GmbH

MUNICH
8 Muenchen 13, Wallensteinplatz 2
Telephone: 0811-35031 Telex: 524-226

COLOGNE
5 Koeln, Bismarckstrasse 7,
Telephone: 0221-522181 Telex: 888-2269
Telegram: Flip Chip Koeln

FRANKFURT
6078 Neu-Isenburg 2
Am Forsthaus Gravenbruch 5-7
Telephone: 06102-5526 Telex: 41-76-82

HANNOVER
3 Hannover, Podbielskistrasse 102
Telephone: 0511-69-70-95 Telex: 922-952

AUSTRIA

Digital Equipment Corporation Ges.m.b.H

VIENNA
Mariahilferstrasse 136, 1150 Vienna 15, Austria
Telephone: 85 51 86

UNITED KINGDOM

Digital Equipment Co., Ltd.

U.K. HEADQUARTERS
Arkwright Road, Reading, Berks.
Telephone: 0734-583555 Telex: 84327

READING
The Evening Post Building, Tessa Road
Reading, Berks.

BIRMINGHAM
29/31, Birmingham Road, Sutton Coldfield, Warwicks.
Telephone: (0044) 21-355 5501 Telex: 337 060

MANCHESTER
13 Upper Precinct, Walkden, Manchester M28 5AZ
Telephone: 061-790-8411 Telex: 668666

LONDON
Bilton House, Uxbridge Road, Ealing, London W.5.
Telephone: 01-579-2334 Telex: 22371

EDINBURGH
Shiel House, Craigshill, Livingston,
West Lothian, Scotland
Telephone: 32705 / Telex: 727113

NETHERLANDS

THE HAGUE
Digital Equipment N.V.
Sir Winston Churchilllaan 370
Rijswijk/The Hague, Netherlands
Telephone: 070-995-160 Telex: 32533

BELGIUM

BRUSSELS
Digital Equipment N.V./S.A.
108 Rue D'Arlon
1040 Brussels, Belgium
Telephone: 02-139256 Telex: 25297

SWEDEN

Digital Equipment Aktiebolag

STOCKHOLM
Vretenvagen 2, S-171 54 Solna, Sweden
Telephone: 98 13 90 Telex: 170 50
Cable: Digital Stockholm

NORWAY

Digital Equipment

OSLO
c/o Firma Service
Waldenmarthranesgate 84-B-86
Oslo 1, Norway
Telephone: 37 19 85, 37 02 30 Telex: 166 43

DENMARK

Digital Equipment Corporation

COPENHAGEN
Vesterbrogade 140, 1620 Copenhagen V

SWITZERLAND

Digital Equipment Corporation S.A.

GENEVA
81 Route de l'Aire
1211 Geneva 26, Switzerland
Telephone: 42 79 50 Telex: 22 683

ZURICH
Scheuchzerstrasse 21
CH-8006 Zurich, Switzerland
Telephone: 01/60 35 66 Telex: 56059

ITALY

Digital Equipment S.p.A.

MILAN
Corso Garibaldi 49, 20121 Milano, Italy
Telephone: 872 748 694 394 Telex: 33615

SPAIN

MADRID
Ataio Ingenieros S.A., Enrigue Larreta 12, Madrid 16
Telephone: 215 35 43 / Telex: 27249

BARCELONA
Ataio Ingenieros S.A., Ganduxer 76, Barcelona 6
Telephone: 221 44 66
Digital Equipment Corporation Ttd.

AUSTRALIA

Digital Equipment Australia Pty. Ltd.

SYDNEY
P.O. Box 491, Crows Nest
N.S.W. Australia 3065
Telephone: 439-2566 Telex: AA20740
Cable: Digital, Sydney

MELBOURNE
60 Park Street, South Melbourne, Victoria, 3205
Telephone: 696-142 Telex: AA40616

PERTH
643 Murray Street
West Perth, Western Australia 6005
Telephone: 214-993 Telex: AA92140

BRISBANE
139 Merivale Street, South Brisbane
Queensland, Australia 4101
Telephone 444-047 Telex: AA40616

ADELAIDE
6 Montrose Avenue
Norwood, South Australia 5067
Telephone: 631-339 Telex: AA82825

NEW ZEALAND

Digital Equipment Corporation Ltd.

AUCKLAND
Hilton House, 430 Queen Street, Box 2471 A,
Auckland, New Zealand
Telephone: 75-533

CANADA

Digital Equipment of Canada, Ltd.

CANADIAN HEADQUARTERS
150 Rosamond Street, Carleton Place, Ontario
Telephone: (613)-257-2615 TWX: 610-561-1651

OTTAWA
120 Holland Street, Ottawa 3, Ontario K1Y 0X7
Telephone: (613)-725-2193 TWX: 610-562-8907

TORONTO
230 Lakeshore Road East, Port Credit, Ontario
Telephone: (416)-274-1241 TWX: 610-492-4306

MONTREAL
9675 Cote de Liesse Road
Dorval, Quebec, Canada 760
Telephone: 514-636-9393 TWX: 610-422-4124

EDMONTON
5531 - 103 Street
Edmonton, Alberta, Canada
Telephone: (403)-434-9333 TWX: 610-831-2248

VANCOUVER
Digital Equipment of Canada, Ltd.
2210 West 12th Avenue
Vancouver 9, British Columbia, Canada
Telephone: (604)-736-5616 TWX: 610-929-2006

ARGENTINA

BUENOS AIRES
Coasin S.A.
Virrey del Pino 4071, Buenos Aires
Telephone: 52-3185 Telex: 012-2284

VENEZUELA

CARACAS
Coasin S.A. (Sales only)
Apartado 50939
Salana Grande No. 1, Caracas
Telephone: 72-9637 Cable: INSTRUVEN

CHILE

SANTIAGO
Coasin Chile Ltda. (sales only)
Casilla 14588, Correo 15, Santiago
Telephone: 396713 Cable: COACHIL

JAPAN

TOKYO
Rikei Trading Co., Ltd. (sales only)
Kozato-Kaikan Bldg.
No. 18-14, Nishishimbashi 1-chome
Minato-Ku, Tokyo, Japan
Telephone: 5915246 Telex: 781-4208
Digital Equipment Corporation International
Kowa Building No. 17, Second Floor
2-7 Nishi-Azabu 1-Chome
Minato-Ku, Tokyo, Japan
Telephone: 404-5894/6 Telex: TK-6428

PHILIPPINES

Stanford Computer Corporation
P.O. Box 1608
416 Dasmarinas St., Manila
Telephone: 49-68-96 Telex: 742-0352

INDIA

H.S. Sonawala Mg Director (Sales Only)
HINDITRON SERVICES PUT LTD.
69/A Nepean Sea Road
Bombay, India

digital

decsystem10 handbook series