R83 ASSEMBLY LANGUAGE

PICK®
SYSTEMS

Technical
Reference
Manual

R83®
Assembly Language

Technical Reference Manual

Version 3.1

First edition, 1984
Second edition, 1987
Third edition, 1990

Pick Systems
1691 Browning, Irvine, CA 92714
(714)261-7425 FAX: (714)250-8187

This entire document and any support materials or programs have been provided under a confidentiality and non-disclosure agreement. Any additional parties having access to such information also need to supply an accepted, signed copy of this statement to Pick Systems prior to gaining access.

Confidentiality and Non-Disclosure Agreement

The undersigned recognizes that in the course of business dealings with Pick Systems, it has become necessary for Pick Systems to provide certain trade secrets, know-how and other proprietary data and material information.

This may include, but is not limited to: documentation, engineering specifications, test procedures, maintenance documentation, schematics and logic diagrams. Pick Systems considers all such information and material, and any derivatives thereof, as proprietary to and a trade secret of Pick Systems.

The undersigned recognizes and agrees with such classification and agrees to treat **ALL** such information and material received from Pick Systems as proprietary to and trade secrets of Pick Systems.

Furthermore, the undersigned agrees not to disclose, divulge, copy or otherwise make available to any third party, either directly or indirectly, any information or material received from Pick Systems without prior written approval.

Date _____

Company _____

Title _____

Name _____

Signature _____

Caution on Using Pick Assembly Language

The Pick Operating System functions as multiple, concurrent processes, each executing within one of many, intricately blended software components; and, at any one point in time, executing at one of several levels. The relationships and interdependencies of these components and/or levels are often implied, rather than explicit. Thus, one may perceive many seemingly transparent operations. Individuals versed in the theory of operating systems tend to be more cognizant of these implied relationships.

On first exposure to the Pick Assembly Language, these implied relationships are often overlooked by the uninitiated programmer, resulting in severe consequences, such as loss of system/data integrity, within the virtual machine. As with most assembly languages, a serious commitment to understanding the underlying architecture is required *prior* to executing any resulting code in a production environment.

The kernel or *monitor* level of the virtual machine traps most violations of virtual addressing continuity (such as handling boundary conditions on linked structures and illegal address ranges) within the total virtual addressing space. However, **the virtual programmer is allowed a tremendous power:**

> *Any and all virtual memory resources, including file space, process workspaces, control blocks, etc., are available to the virtual assembly language programmer. It is only through strict adherence -- to interfacing correctly with system routines, to observing programmer-controlled boundary checking, to using interprocess semaphores, and to using proper elements -- that resources are both defined and preserved.*

The Pick implementation makes extensive use of *global structures* such as control blocks, tables, and/or memory resources, the very nature of which requires broad-based knowledge. There also exists many seamless interfaces between components and levels. Finally, implicit operations may take place right down at the virtual instruction level.

The scope of this manual is not to act as a tutorial guide to utilizing the Pick Virtual Assembly Language. Rather, the aim is to present a knowledge base of interface specifications, which, taken collectively, may be utilized to create highly efficient functionality within the Pick Operating Environment.

The forseen learning curve, and subsequent requirements of attention to programming details, necessitate that the Pick Assembly Language be an unsupported product.

* * * * *

Although Pick Systems may provide access to its Assembly environment on various hardware implementations (and always under a Confidentiality and Non-Disclosure Agreement), the fact that **Pick Systems cannot support any user whose system is running with user-written (or third party supplied) assembly language code** should be a major consideration when electing to delve into the Pick Assembler.

Format Conventions

MD	Words in all UPPERCASE are Pick terms.
MLIST	Words or characters in **boldface** must be entered as shown in uppercase. These are commands, file names, and options.
file.name	Words in *italics* are parts of the command which must be replaced by an actual name, word or number. For example, *file.name* might be replaced by **ORDERS**.
{(*options*)}	Information displayed within {} (braces) is optional.
`description`	Text printed `in this type` represents an actual word, prompt, or listing that is output by the system.

Table of Contents

System Debugger

Appendix

Index

Figures

Tables

Introduction

The Pick Assembly Language

Computer languages can be divided into levels according to the sophistication of their instructions. The lowest level is machine level, which is a series of binary digits. Ultimately, all languages must be reduced to machine level. The next level consists of assembly languages. These use mnemonic instructions that are translated into machine language by a translation program known as an assembler. Higher level languages approach a natural human language syntax. Pick/BASIC, C, PASCAL, ADA, FORTRAN, and COBOL are examples of higher level computer languages.

High level languages are also translated into machine language by a translation program known as a compiler. However, the translation of high level languages into executable machine code does not produce as efficient a code as does the translation of assembly language programs. Because it translates in a one-to-one correspondence, an assembly language program shows a two- to three-time improvement in performance as compared to an equivalent higher level language program.

One advantage of a higher level language is that, theoretically, it can be run on any computer marketed. Once a compiler or interpreter for one of these languages has been implemented on a specific computer, an application written in the language runs the same on that computer as it would on different computers. Most assembly languages are tied to specific hardware; programs written in a particular assembly language cannot be run on other hardware. However, one assembly language that can run on many types of hardware is the PICK Assembly Language. PICK is not tied to any particular hardware, but rather applies the assembler process to many machines.

In use, the PICK Assembler is associated with a particular target computer. Assembly language modules, also called source items and modes, are created and modified as lines of source statements. These source items are assembled and loaded with other object code items into an executable machine code item. The actual bit pattern of this machine code item differs from machine type to machine type. For example, the machine code item for the IBM PC/AT is totally different from that of the IBM RT, in pattern as well as size. The generation of target object code is done by assembling the PICK source code with associated files that define the actual target machine. The code that is the PICK System is contained within 1024 of these assembly language modes.

Installing

Floppy diskettes 1 and 2 contain all the necessary files to create, assemble and load PICK Assembler code for the IBM PC/AT 3.1 or beyond. Ensure that you have at least 700 frames of disk space available. Follow these instructions to install the account:

1. At the TCL prompt (>),
   type: **LOGTO SYSPROG**

2. To verify that there is no ASSEMBLER account on the system already,
   type: **LIST ONLY SYSTEM "ASSEMBLER"**

3. Insert floppy #1 in drive A.
   To set the diskette drive and prepare for the restore,
   type: **SET-FLOPPY (SA**
   T-REW

4. Type: **ACCOUNT-RESTORE ASSEMBLER**

5. When prompted, `'ACCOUNT NAME ON TAPE'`,
   type: **ASSEMBLER**

6. Load floppy #2 in the drive when requested; and, to continue,
   type: **C**

7. The following files are restored with the ASSEMBLER account:

BP	Pick/BASIC Assembly utilities
NAT.OSYM	Native Object Symbols file
NAT.PSYM	Native Permanent Symbol file
NAT.SM	Native Source Modes
NAT.TSYM	Native Temporary Symbol file
OPT.ERRS	Optimization errors
VIR.OSYM	Virtual Object Symbols file
VIR.PSYM	Virtual Permanent Symbol file
VIR.SM	Virtual Source mode file
VIR.TSYM	Virtual Temporary Symbol file

8. When the restore completes, you are returned to the TCL prompt.
   Type: **LOGTO ASSEMBLER**

9. To define the type of assembly (XT or AT), type for:
   ATs, **SET-AT**
   XTs, **SET-XT**

   The assembled code is stored according to assembly type
   for ATs, in the file: **NAT.SMS,AT**
   for XTs, in the file: **NAT.SMS,XT**

The **SET** creates a Q-pointer, **NAT.SM**, which simplifies addressing the appropriate data area.

Assembling and Loading

The PICK Assembler for the IBM PC-AT/XT uses the same source code as all other PICK Systems. Use the following instructions to assemble and load your code:

1. Create your source code in the file, **VIR.SM**, using the editor, **ED**, with the **A** option. The format for all Pick Assembly language programs is demonstrated in the following sample program.

   Type: **ED VIR.SM** *item-name* **(A**
   I

   Enter your source lines in a similar format:

```
001                FRAME   1023
002  * SAMPLE PROGRAM
003  * (Other
004  *     descriptive
005  *          information
006  *               lines)
007                EP      !GET.CHAR   ENTRY POINT 0
008                EP      !PUT.CHAR   ENTRY POINT 1
009                NEP     *           ENTRY POINT 2
010                NEP     *           ENTRY POINT 3
011                NEP     *           ENTRY POINT 4
012                NEP     *           ENTRY POINT 5
013                NEP     *           ENTRY POINT 6
014                NEP     *           ENTRY POINT 7
015                NEP     *           ENTRY POINT 8
016                NEP     *           ENTRY POINT 9
017                NEP     *           ENTRY POINT 10
018                NEP     *           ENTRY POINT 11
019                NEP     *           ENTRY POINT 12
020                NEP     *           ENTRY POINT 13
021                NEP     *           ENTRY POINT 14
022                NEP     *           ENTRY POINT 15
023  !GET.CHAR     EQU     *
024                MOV     TSBEG,TS
025                INC     TS
026                READ    TS
027                MCI     X'FF',TS
028                ENT     CONVEXIT
029  !PUT.CHAR     EQU     *
030                MOV     TSBEG,TS
031                INC     TS
032                WRITE   TS
033                MCI     X'FF',TS
034                ENT     CONVEXIT
035                END     *
```

   Press the return key twice at the end of the last line; and, to file,
   type: **FI**

2. To assemble,
   type: **AS** *item-name*

3. To check for Translation (1st pass) assembly errors,
   type: **LIST VIR.SM** *item(s)*

   The 'asm err' column will show any errors.

4. To check for Optimization and Native (2nd pass) assembly errors,
   type: **LIST NAT.SM** *item(s)*

   The 'opt err' and 'asm err' columns will show any errors. It is important to check the 'obj siz dec' column to insure the final assembled object is not over the maximum frame size of 2048 bytes. Object exceeding this size will be truncated by **MLOAD**.

5. If there are no errors, to load the assembled code into the operating system,
   type: **MLOAD NAT.SM** *item(s)*

   The Native assembly is loaded into the ABS area identified in the **FRAME** statement on line 1 of the source mode.

Assembler Environment

This chapter describes the verbs necessary to manipulate assembly language source modes and the syntax needed to create those modes.

Verbs

The verbs described in this section are TCL (Terminal Control Language) verbs used exclusively in the Assembler environment.

The following options are legal with most verbs:

option

F	suppresses footers.
H	suppresses headers.
I	suppresses the item.id.
N	no pause; suppresses the pause at end of page on display to the terminal.
P	sends output to the printer.
S	suppresses the "number of items" message.

For brevity, the above options are explained once, here. Additional options or exceptions will be given under the specific verb.

Item.lists are shown as optional -- enclosed in braces {} -- since a select list can be activated in the preceding statement. An *item.list*, therefore, can be an explicit list of *item.id*s, an active select list or, in many cases, an asterisk (*) indicating all of the items.

AS *item.id*

The PICK Assembler is invoked by the Proc, **AS**. This proc reads the mode to be assembled from the source file, VIR.SM, and writes the generated object code in the destination file, NAT.SM.

item.id names the source mode to be assembled.

The Assembler stores assembly errors as a subvalue in the source line causing the error. Undefined references are also appended to the mode.

ERROR	DESCRIPTION
OPCD?	missing opcode
OPRND REQD	missing operand(s)
ILGL OPCD: x	invalid opcode
MUL-DEF	symbol already defined
REF: UNDEF	symbol not defined
TRUNC	operand value truncated
UNDEF: x {,x}	undefined symbols

WARNING: The Assembler account can only be used by one user at a time.

CREF *file.name {item.list} {(options)}*

A cross-index is created in the CSYM file of external references or operand symbols by selecting elements of type b, c, d, e, f, h, n, r, s or t.

file.name	names the file containing the modes to be cross-indexed.
item.list	names the modes to be cross-indexed.

DUMP *n{-m} {(options)}*

The **DUMP** verb displays data in a frame in either character or hexadecimal format. There are two types of frames: ABS and file. ABS frames may be object code (Assembly or Pick/BASIC compiled), buffers or workspace required by the system. ABS frames contain 2048 bytes and are not linked.

File frames contain 512 byes; 500 for data, 12 as link fields. Linked frames are used to define data areas that are greater than 1 frame in length. The groups in data files may expand as more data is placed in the group, so when the end of a frame is reached, another frame is obtained from the system overflow and linked to the end of the group.

n	frame number (FID) of first frame. May be expressed in decimal or hexadecimal. To indicate hexadecimal, precede the number with a period (.).
m	FID of last frame to display.
options	
C	Displays ABS frame; dump begins with byte 0 of the frame and continues for 2048 bytes.
G	Group; specifies that the data starting at frame *n* is to be dumped, and that the dump continue following either the forward or backward links (depending on whether the **u** option is specified). The dump terminates when the last frame in the logical chain has been found.
L	Links; specifies that the dump be confined to the links of the frames concerned. No data is displayed.
U	Upward trace. The data or links are traced logically upward. That is, the backward links are used to continue the display.
X	Hexadecimal. The frames are dumped in hexadecimal with ASCII representation along the right side of the display.

The format of the linked fields is shown in the appendix. Linkage information displayed is meaningful only for linked frames. Unlinked frames have no specified format. All 512 bytes may be used by the system.

MAP *file.name {(options)}*

Displays PCB (Primary Control Block), SCB (Secondary Control Block), DCB (Debug Control Block), QCB (Quadrenary Control Block) symbols in a chart layout, giving symbol name, location and size.

file.name names the file containing the symbols. Defaults to VIR.PSYM.

MLIST *file.name {item.list} {(options)}*

This verb prints or displays assembly language mode listings. The format includes columns for the statement number, location counter, object code and the label, op-code, operand and comment fields of the source code. Macro expansions are shown as source code with the operation codes prefixed by a plus sign (+). Any error messages appear in the location counter/object code area.

file.name names the file containing the modes to be listed.
item.list names the modes to be listed.
options
- *n{-m}* restricts the listing to line number *n* or the range, *n* through *m*, inclusive.
- **E** error lines only; suppresses pagination and enters EDIT at the end of the listing.
- **M** shows the macro-expansion of source statements. For translation assemblies, shows the translation expansions.
- **S** suppresses the listing of the object code column.

MLOAD *file.name {item.list} {(options)}*

Once modes have been assembled, they can be loaded into the ABS area for execution using the **MLOAD** verb. The assembled mode is loaded into the frame specified by the FRAME op-code statement.

file.name names the file containing the assembled modes to be loaded.
item.list names the modes to be loaded.
options
- **N** returns the check-sum data without loading item.
- **V** verifies mismatches and errors only.

The mode will not load correctly if its size exceeds 2048 bytes, or if FRAME is not the first statement assembled in the mode. In either case, a message indicates the error.

If the load is successful, this message is returned:

[216] MODE '*item.id*' LOADED; FRAME = *nnnn* SIZE = *sss* CKSUM = *cccc*

where

nnnn is the 4-digit decimal number of the frame where the mode was loaded.

sss is the number of bytes of object code loaded into the frame, expressed in hexadecimal notation.

cccc is the byte check-sum for the object code in the loaded mode.

MVERIFY *file.name {item.list} {(options)}*

After assembling and loading a program, the verb **MVERIFY** can be used to check the object code in the file of the assembled program against the code loaded into the ABS area. Verification consists of comparing the check-sum for the assembled mode to the check-sum of the code loaded into the ABS area. If there are no errors, the frame id, size and the check-sum are displayed. If there are errors, the frame name, frame id and number of mismatches are displayed.

file.name names the file containing the assembled modes.

item.list names the modes to be verified; may be a select list or explicitly named modes.

options

A outputs a columnar listing of all bytes which mismatch. Each value in the assembled code which does not match is listed, followed by the value in the executable frame.

E lists errors only. Only statistics for modes with verify errors are listed.

SET-SYM *file.name*

SET-SYM defines which global symbol file the System Debugger will use. VIR.PSYM is typically the file used. There is no system default.

STRIP-SOURCE *file.name {item.list} {(options)}*

Removes the source code from Assembly Language programs. This frees large amounts of disc space by returning the frames back to overflow. Modes with the source stripped out can still be verified against the ABS. The first six lines of the source item will be copied without source code stripping. Standard PICK Systems convention for source modes is for line 1 to be the "FRAME" statement and lines 2 through 6 to contain other descriptive information.

file.name names the file that contains the source mode. After the verb is invoked, the user is prompted for the name of the destination file where the stripped object code is to be stored.

item.list names the modes to be stripped; may be a select list or explicitly named modes.

XREF *file.name {item.list} {(options)}*

The **XREF** verb is used to build a cross reference in the XSYM file of symbols in the cross-indexed file, normally CSYM.

file.name names the file that contains the cross-indexed symbols.

item.list names the modes to be cross referenced; may be a select list or explicitly named modes.

Assembly Source Format

A PICK assembly language source program, called a mode, is a sequence of symbolic statements existing as a data item in a file. A source mode can be created or modified using the editor with the A option. Each source mode statement is an attribute in the source mode item. The assembler assumes the first statement is a comment and ignores it.

Each source statement contains the following fields:

```
{label} opcode {operand1{,operand2{,operand3}}} {comments}
```

A statement beginning with an asterisk (*) is treated as a comment. Every source mode must begin with at least five comment statements. Usually several are employed to summarize the purpose of the mode.

Unless a statement is a comment, it must contain an operator field, and may potentially contain a label field, an operand field, and a comment field. Fields within statements are separated by one or more spaces.

Labels

A label gives a symbolic name to a particular location and must be unique within the mode in which it appears. Labels are optional for most statements but are required by some due to the nature of the operation the statement signifies. If present, a label begins in the first character position of the source statement and continues until it is terminated by a space.

A label may consist of any number of alphanumeric characters. To avoid problems when referencing labels and listing modes, a label should be reasonable in length, begin with an alphabetic character, and not contain an asterisk (*), plus sign (+), minus sign (-), apostrophe ('), comma (,), percent sign (%), at sign (@), or equal sign (=).

Operators

The operator is the "action word" of the assembly language statement. Every statement, except those beginning with an asterisk, must have one. The operator field begins after the spaces terminating the label, or after one or more initial spaces, if a label is not present, and continues until it is terminated by a space.

An operator is a mnemonic either for a Pick Assembly Language instruction or for a directive to the assembler. Some operators refer to macros which expand into multiple assembly instructions. An operator can cause one or several machine instructions to be generated as object code. Valid operators are limited to those defined in the VIR.OSYM file being used by the Assembler for the mode being assembled.

Operands

Operands are the things operated upon by an operator. The operand field begins after the spaces terminating the operator and continues until it, in turn, is terminated by a space. The number of operands required in a statement is a function of the operator: some operators require several and some require none. Multiple operands, if present, are separated by commas.

An operand may be a symbolic reference to a label, variable, constant, register, or storage register which is locally defined, or defined in the Processor Symbol File, VIR.PSYM. An operand may also specify a literal value which gets assembled into the object code of the instruction. A literal may be a positive or negative decimal number, a hexadecimal number, which must be preceded by X and enclosed in single quotation marks (for example, X'aa'), or an ASCII character, which must be preceded by C and enclosed in single quotation marks (for example, C'a'). Plus (+) and minus (-) arithmetic operators may be used to combine multiple terms to produce a single literal value.

The asterisk (*) has special significance when used as an operand. When used with the equate operator, an asterisk refers to the current value of the program location counter. When used in the operand field of a statement that requires no operands, it simply causes a comment in that statement to align with the comments of surrounding statements.

Comments

Any assembly language statement may include commentary information following the spaces terminating the last required field of the statement. Comments cause no object code to be generated and do not affect instruction execution.

Syntax

Braces are used to indicate something optional (for example, {l} means the label l is optional). An ellipsis is used to indicate continuation (for example, n,n,... means the sequence n,n, may continue).

Whenever two arithmetic elements are named as operands in an instruction (such as: a,a or n,a), both of the elements are implied to be, and are required to be, equal in size (that is, both half tallies, tallies, double tallies, or triple tallies).

The instruction code is listed in the first column, the operands associated with it (if any) are listed in the second column, and a full description of how the instruction operates is in the third column.

A list of valid operand forms is shown with each instruction definition.

EXAMPLE:

```
INC  a
     a,n
     r,a
```

means that the INC instruction may be used with any of three different operand forms. That is,

```
INC   a
INC   a,n
INC   r,a
```

are all valid instruction forms.

The following symbols are used to represent the system delimiters in examples and illustrations:

Symbol	Delimiter Name	Abbreviation	Hexadecimal Value
[	start buffer mark	SB	X'FB'
\	subvalue mark	SVB	X'FC'
]	value mark	VM	X'FD'
^	attribute mark	AM	X'FE'
_	segment mark	SM	X'FF'

Table 1 System Delimiters

Instruction Set

This chapter defines the instructions provided by the Pick Assembly Language. The instructions are first summarized by category, then listed alphabetically with details of functional operation.

Some virtual assembly instructions expand into more than one low-level instruction, and many instructions use elements not explicitly named in the instruction. In particular, the accumulator and R15 are often used in many instruction expansions.

Instruction Categories

The following sections describe these categories of instructions used in the Assembler:

Arithmetic
Arithmetic Compare and Branch
Assembler Directives
Bit
Branch
Character Compare and Branch
Character String
Conversion
Logical
Process Synchronization
Register
Other

Each instruction that belongs to the category is listed. Detailed information about each instruction follows the categories in an alphabetical listing.

Arithmetic Instructions

Arithmetic instructions, which operate on signed binary integers, may require one or two operands. Each operand specifies a literal, or an 8-, 16-, 32-, or 48-bit arithmetic element (that is, half tally, tally, double tally, or triple tally respectively).

The value of a literal is placed in the generated object code. It is stored as two, four, or six bytes of object code. The number of bytes normally is determined by the size of the destination element. For example, `INC T2,20` would use two object bytes to hold the decimal literal "20" (that is, x'0014') because the destination element T2 is a tally. The number of bytes allocated to literal storage for multiply and divide instructions is determined by the size of the multiplicand or dividend. Four bytes are provided for MUL and DIV. Six bytes are provided for MULX and DIVX. Specifying a literal greater than the capacity of the object space provided usually results in an assembly error. For example, `INC T2,65536` would be reported as an error.

The following arithmetic instructions operate on specified memory locations:

ZERO	Zero
ONE	One
INC	Increment
DEC	Decrement
NEG	Negate
MOV	Move

Another class of arithmetic instructions operate between a specified memory location and a fixed location called the accumulator. The accumulator is a data element in the PCB consisting of 8 contiguous bytes. Each byte of the accumulator is directly addressable using the PCB elements. D0, the low-order four bytes, is the element used in most operations. D1, the accumulator extension, is used for the remainder in DIV operations. FP0, the 6-byte accumulator, is a triple tally used for extended mathematical functions. 1-byte and 2-byte operands are sign extended to form a double word value before the operation is performed. Storage operands may not cross frame boundaries.

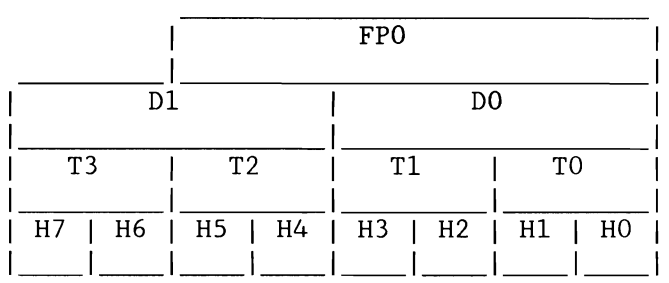


Figure 1 Accumulator Layout

For the following arithmetic instructions, if an operand specifies a triple tally such as FP0, the Assembler automatically generates object code for the extended form of the instruction; (for example, `LOADX FP1`). For an element less than 32 bits in size, the sign bit of the element is propogated, or sign extended, through the remainder of D0.

LOAD	Load
LOADX	Load extended
STORE	Store
ADD	Add
ADDX	Add extended
SUB	Subtract
SUBX	Subtract extended
MUL	Multiply
MULX	Multiply extended
DIV	Divide
DIVX	Divide extended

Arithmetic Compare and Branch Instructions

These instructions compare two operands and determine whether to branch depending on the results. Arithmetic comparisons include the sign bits of the elements. In all comparisons, the second operand is conceptually subtracted from the first; i.e., the result is not stored and the operands are not changed.

BE	Branch if equal
BU	Branch if unequal
BH	Branch if high
BHE	Branch if high or equal
BL	Branch if low
BLE	Branch if low or equal

The following instructions compare a named operand to an implied operand of zero:

BZ	Branch if zero
BNZ	Branch if not zero
BHZ	Branch if higher than zero
BHEZ	Branch if high or equal to zero
BLZ	Branch if less than zero
BLEZ	Branch if less than or equal zero

The branch decrementing instructions are used for looping. For the *a,l* operand form, the element named by the first operand is decremented by one. In the other formats, the first operand is decremented by the value of the literal or the element named by the second operand. The first operand is then tested against zero. If the specified condition is met, a branch is taken to the local label named by the last operand.

BDZ	Branch decrementing zero
BDNZ	Branch decrementing not zero
BDHZ	Branch decrementing higher than zero
BDHEZ	Branch decrementing higher than or equal to zero
BDLEZ	Branch decrementing less than or equal to zero
BDLZ	Branch decrementing less than zero

Assembler Directives

An assembler directive is an instruction for the Assembler to perform a specific function at assembly time. These directives do not generate object code.

%x	Define element
ADDR	Address
ALIGN	Align
CHR	Character
CMNT	Comment
DEFx	Define element
DTLY	Double tally
END	End program
EQU	Equate
FRAME	Frame
FTLY	Triple tally
HTLY	Half tally
INCLUDE	Include source
MTLY	Define double tally address
ORG	Originate
TEXT	Text
TLY	Tally

Bit Instructions

A bit (**bi**nary-dig**it**), the smallest unit of storage in any digital computer, can have two states. In the Pick System, set and zero are used to describe the bit states. Each bit instruction shown here allows setting, zeroing, or testing a specific bit.

SB	Set bit
ZB	Zero bit
MOV	Move bit
BBS	Branch if bit set
BBZ	Branch if bit zero

Branch Instructions

Program flow can branch as a result of a data comparison, a specific condition, or unconditionally. Certain unconditional branch instructions push the location of the next sequential instruction onto a return stack. The destination of any branch is always a label, which, in a source program, may consist of any number of alphanumeric characters. A label must not, however, contain colons (:), percent signs (%), plus signs (+), minus signs (-), asterisks (*), or slashes (/). The following are unconditional branch instructions:

B	Branch
BSL	Branch and stack location counter
BSLI	Branch and stack location counter indirectly, using T0
BSL*	Branch and stack location counter indirectly
ENT	Enter External Mode
ENTI	Enter Indirect, using T0
ENT*	Enter Indirect
RTN	Return from Subroutine

Character Compare and Branch Instructions

The following instructions compare a character to a character range and determine whether to branch depending on the results. The character range may be alphabetic, numeric, or hexadecimal. The ranges for some instructions may be restricted and are described, where applicable, in the descriptions listed alphabetically later in this chapter.

BCA	Branch if character alphabetical
BCNA	Branch if character not alphabetical
BCN	Branch if character numeric
BCNN	Branch if character not numeric
BCX	Branch if character hexadecimal
BCNX	Branch if character not hexadecimal

The following instructions compare two operands and determine whether to branch depending on the results. The comparison is logical, which means the characters are treated as 8-bit, unsigned fields with the lowest possible character being x'00' and the highest being x'FF'. The character addressed by the first operand is compared to the character addressed by the second operand. If the condition specified by the instruction is met, a branch is taken to the label.

BCE	Branch if characters equal
BCU	Branch if characters unequal
BCL	Branch if character less than
BCLE	Branch if character less than or equal
BCH	Branch if character higher than
BCHE	Branch if character high than or equal

The following instructions compare two strings and determine whether to branch depending on the results. A literal operand is used to specify the string delimiter. Any character that is logically greater than or equal to the literal terminates a string. The strings do not have to be terminated by the same delimiter.

BSTE	Branch if strings equal
BSTU	Branch if strings unequal

A three-way branch on a string comparison can be implemented by following the Branch Strings instruction with a Branch Character instruction. For example:

```
        BSTE    r,r,n,EQUAL     Branch if strings are equal
        BCH     r,r,HIGH        Branch if string 1 > string 2
        EQU     *               String 1 < string 2
          .
EQUAL   EQU     *
          .
HIGH    EQU     *
          .
```

Character String Instructions

The character string instructions allow scanning or moving character strings. A character string consists of any number of logically adjacent characters which can cross linked frame boundaries. The transition of frame boundaries during instruction execution is handled automatically and is transparent to the user. The XMODE facility is relevant to many of these instructions. These are the Character String Instructions.

XCC	Exchange characters
MCC	Move character to character
MCI	Move character incrementing
MIC	Move incrementing character
MII	Move incrementing incrementing
MIIR	Move incrementing incrementing to R15
MIIT	Move incrementing incrementing to count
MIID	Move incrementing incrementing through delimiter
MIIDC	Move incrementing incrementing through delimiter counting
MIIDT	Move incrementing incrementing through delimiter or to count
SIT	Scan incrementing to count
SID	Scan incrementing through delimiter
SIDC	Scan incrementing through delimiter counting
SICD	Scan incrementing counting delimiters
SITD	Scan incrementing through delimiter or to count

Several of the character string instructions contain a literal, known as a "variant byte" or "scan mask." The variant byte controls byte-by-byte matching against preset delimiters. The format of the variant byte literal for all character string instructions, except SICD, is shown in the following table:

Bit	Meaning
0 (most significant)	1 = Stop on match 0 = Stop on mismatch
1	Compare to x'FF' (SM)
2	Compare to x'FE' (AM)
3	Compare to x'FD' (VM)
4	Compare to x'FC' (SVM)
5	Compare to character in SC0
6	Compare to character in SC1
7 (least significant)	Compare to character in SC2

Table 2 Variant Byte Format

The most-significant bit determines whether the instruction stops on a "match" condition (bit is set) or on a mismatch condition (bit is zero). Only those characters whose corresponding bits are set are checked for match or mismatch. The first four characters are the system delimiters. The last three are variable and reside in the user's PCB.

Some examples of variant bytes and their respective match conditions follow:

Mask	Condition
x'01'	Match on non-blank (if there is a blank in SC2).
x'C0'	Match on SM.
x'E0'	Match on SM or AM.
x'F0'	Match on SM, AM, or VM.
x'A0'	Match on AM.
x'A4'	Match on AM or the character in SC0.

The SICD instruction has a different variant byte format. This is due to the ready for position nature of this instruction. Since an SICD is used to ready a register for string insertion in a data structure, the variant byte is based on the number of delimiters rather than just the delimiter. The format for the SICD variant byte is:

Bit	Meaning
0 (most significant)	0 = count is not pre-decremented. 1 = count is pre-decremented before instruction is started.
1	0 = scan terminates when a character is greater than the delimiter set by bits 2-7. 1 = scan terminates only when a character is found which is greater than the character contained in SC2.
2	Scan delimiter is x'FE'. (AM)
3	Scan delimiter is x'FD'. (VM)
4	Scan delimiter is x'FC'. (SVM)
5	Scan delimiter is contained in SC0.
6	Scan delimiter is contained in SC1.
7 (least significant)	Scan delimiter is contained in SC2.

Table 3 SICD Variant Byte Format

The most-significant bit of the variant byte is for ordinal positioning. Prior to the instruction, the register is pointing to the byte before attribute 1 of the string. The pre-decrement feature of the SICD variant byte allows the attributes of the string to be referenced by their logical numbering scheme.

The second bit is used when scanning for system level delimiters. Logical character compares are used: x'FE' is > x'20'.

Conversion Instructions

The following instructions provide the means to convert decimal and hexadecimal string numbers to binary numbers and vice versa.

MBD	Move binary to decimal
MBX	Move binary to hexadecimal
MBXN	Move binary to hexadecimal
MDB	Move decimal to binary
MXB	Move hexadecimal to binary
MSDB	Move string decimal to binary
MFD	Move string decimal to binary
MSXB	Move string hexadecimal to binary
MFX	Move string hexadecimal to binary

Logical Instructions

The following truth table defines the operations performed by the AND, OR, and XOR instructions. The operations are performed on each bit of the elements specified and the result replaces the destination operand. Unlike most Pick instructions, the destination operand comes first in logical instructions

	1	1	0	0	Value 1
	1	0	1	0	Value 2
AND	1	0	0	0	
OR	1	1	1	0	Results
XOR	0	1	1	0	

Figure 2 AND, OR, and XOR Truth Table

These are the logical instructions:

AND	And
OR	Or
XOR	Exclusive or
SHIFT	Shift

Process Synchronization Instructions

These instructions are used to synchronize execution of processes.

LOCK	Lock
RQM	Release time quantum

Register Instructions

An address register contains the address of a character (that is, points to a character). The register instructions operate directly on registers as opposed to instructions that operate on data pointed to by registers. Register instructions which increment or decrement the register value change the address where the register points.

BE	Branch if equal
BU	Branch if unequal
DEC	Decrement register
INC	Increment register
LAD	Load absolute difference
MOV	Move register to register
SETUP	Setup register
SETUP0	Setup register to byte 0
SETUP1	Setup register to byte 1
SR	Set register to FID
SRA	Set register to address
XRR	Exchange registers

Other Instructions

These instructions do not conveniently fit into any of the previous categories. Most of them relate to the operation of the system.

ECHO.ON	Echo off
ECHO.OFF	Echo on
EP	Entry point
FRM.LOCK	Frame lock
FRM.UNLOCK	Frame unlock
GET.TIME	Get time
HALT	Halt
MSG#	Message number
NEP	No entry point
NOP	No operation
READ	Read input queue
RQM	Release quantum
SET.TIME	Set time
SETUP.BAUD	Set baud
SLEEP	Sleep
TIME	Get system time
WRITE	Write to output queue

Symbolic Operand Definitions

This table defines the operand types used with the Pick assembly language operators. The 'a' symbol and 'v' symbol are used as a documentation convenience. Braces are used to indicate something optional; e.g., {l} means the label, l, is optional.

An asterisk (*) in the operand field has two possible functions: (1) when used with the equate operator, the asterisk represents the current value of the program location counter; or, (2) when used in a statement requiring no operands, the asterisk forces alignment of a comment in that statement with comments of surrounding statements.

Symbol	Meaning	Definition
a	**Arithmetic Element**	This symbol is used as a convenience in documen- tation to mean any of the following four arithmetic elements: half tally, tally, double tally, or triple tally. Whenever two arithmetic elements are named as operands in an instruction (such as: a,a,l), both of the elements are implied to be, and are required to be, equal in size; i.e., both half tallies, tallies, double tallies or triple tallies.
b	**Bit**	An element having two states, addressed using a base register, a byte displacement, and a bit displacement
c	**Character**	An 8-bit element, addressed using a base register and byte displacement
d	**Double tally**	A signed, numeric, 32-bit data element, addressed using a base register and byte displacement
e	**Quad tally**	A signed, numeric, 64-bit data element, addressed using a base register and byte displacement
f	**Triple tally**	A signed, numeric, 48-bit data element, addressed using a base register and byte displacement
h	**Half tally**	A signed, numeric, 8-bit data element, addressed using a base register and byte displacement
i	**Item ID**	The name of an item, external to the current mode
l	**Label**	A location, internal to the current mode, to which the program execution can transfer

Symbol	Meaning	Definition
m	**Mode ID**	A 16-bit modal entry identification, comprised of a 4-bit entry point and a 12-bit frame number
n	**Literal**	A constant or immediate value whose size is dependent upon the instruction in which it appears
o	**Frame ID**	An integer representing a frame, ranging from 1 to MAXFID.
p	**Entry Point**	An integer, ranging from 0 to 15.
r	**Address register**	One of sixteen 64-bit elements, R0 through R15, containing a frame and displacement to point to a specific character in virtual memory
s	**Storage register**	A 48-bit element used to save the frame and displacement (virtual address) of an address register addressed using a base register and a 16-bit word displacement
t	**Tally**	A signed, numeric, 16-bit data element, addressed using a base register and byte displacement
v	**Variant**	The variant byte specifies string movement termination criteria.

Table 1 Symbolic Operand Definitions

Assembler Instructions

ADD	a n	**Add to Accumulator** - The value of element a, or the 4-byte literal n, is added to D0 with the sum replacing the original contents of D0.
ADDR	n,n	**Address** - This directive reserves storage and defines the symbol in the label field to be type s, a storage register. All symbols or variable names used as operands must have a symbol type code. The first operand specifies the displacement of the generated byte address. The second operand specifies the FID or frame number.
ADDX	a n	**Add Extended** - The value of element a, or the 6-byte literal n, is added to FP0. The sum replaces the original contents of FP0. If the operand is less than 6 bytes, a 6-byte operand is generated by extending the sign bit of the operand.
ALIGN	*	**Align** - The assembly instruction following this directive is aligned on an even address. The location counter is adjusted, if necessary, by generating one object byte of zero. This is useful before a section of DEFinitions of tallies, double tallies etc., to ensure word alignment.
AND	c,n r,n r,r	**Logical AND** - The byte of the first operand is logically ANDed with the mask byte of the second operand. The result is stored at the byte of the first operand. The second operand is unchanged.
B	l	**Branch Unconditionally** - A branch is taken to the location specified by the local label.
BBS	b,l	**Branch If Bit Set** - If the bit is set (1), then a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BBZ	b,l	**Branch If Bit Zero** - If the bit is zero (0), then a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.

BCA	r,l	**Branch If Character Alphabetic** - If the character addressed by the register is alphabetic (ASCII A-Z, a-z; hex 41-5A or 61-7A), then a branch is taken to the label. The alphabetic range is implementation specific for the character set and may not be exactly as stated above in all cases, as in, for example, European language character sets. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCE	c,c,l c,n,l c,r,l n,c,l n,r,l r,c,l r,n,l r,r,l	**Branch If Character Equal** - The character addressed by the first operand is compared with the character addressed by the second operand. If the two characters are equal, then a branch is taken to the label specified by the third operand. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCH	c,c,l c,n,l c,r,l n,c,l n,r,l r,c,l r,n,l r,r,l	**Branch If Character Higher** - The character addressed by the first operand is compared with the character addressed by the second operand. If the first operand is numerically greater than the second operand, the branch is taken to the label specified by the third operand. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCHE	c,c,l c,n,l c,r,l n,c,l n,r,l r,c,l r,n,l r,r,l	**Branch If Character Higher or Equal** - The character addressed by the first operand is compared with the character addressed by the second operand. If the first operand is numerically greater than or equal to the second operand, then a branch is taken to the label specified by the third operand. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCL	c,c,l c,n,l c,r,l n,c,l n,r,l r,c,l r,n,l r,r,l	**Branch If Character Lower** - The character addressed by the first operand is compared with the character addressed by the second operand. If the first operand is numerically less than the second operand, then a branch is taken to the label specified by the third operand. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.

BCLE	c,c,l c,n,l c,r,l n,c,l n,r,l r,c,l r,n,l r,r,l	**Branch If Character Less than or Equal** - The character addressed by the first operand is compared with the character addressed by the second operand. If the first operand is less than or equal to the second operand, then a branch is taken to the label specified by the third operand. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCN	r,l	**Branch If Character Numeric** - If the character addressed by the register is numeric (ASCII 0-9; hex 30-39), then a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCNA	r,l	**Branch If Character Not Alphabetic** - If the character addressed by the register is not alphabetic, (ASCII A-Z, a-z; hex 41-5A, 61-7A), then a branch is taken to the label. The alphabetic range is implementation specific to the character set. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCNN	r,l	**Branch If Character Not Numeric** - If the character addressed by the register is not numeric, (ASCII 0-9; hex 30-39), then a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCNX	r,l	**Branch If Character Not Hexadecimal** - If the character addressed by the register is not hexadecimal, (ASCII 0-9, A-F), then a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch.
BCU	c,c,l c,n,l c,r,l n,c,l n,r,l r,c,l r,n,l r,r,l	**Branch If Characters Unequal** - The character addressed by the first operand is compared with the character addressed by the second operand. If the two characters are unequal, then a branch is taken to the label specified by the third operand. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BCX	r,l	**Branch If Character Hexadecimal** - If the character addressed by the register is hexadecimal, (ASCII 0-9, A-F), then a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.

BDHEZ	a,l a,a,l a,n,l	**Branch Decrementing High or Equal to Zero** - The first operand is decremented by one, or by the value of the second operand if there are three operands. Then if the first operand is higher or equal to zero (>=0), a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BDHZ	a,l a,a,l a,n,l	**Branch Decrementing Higher than Zero** - The first operand is decremented by one, or by the value of the second operand if there are three operands. Then if the first operand is arithmetically higher than zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BDLEZ	a,l **a,a,l** a,n,l	**Branch Decrementing Less than or Equal to Zero** - The first operand is decremented by one, or by the value of the second operand if there are three operands. Then if the first operand is less than or equal to zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BDLZ	a,l a,a,l a,n,l	**Branch Decrementing Less than Zero** - The first operand is decremented by one, or by the value of the second operand if there are three operands. Then if the first operand is arithmetically less than zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BDNZ	a,l a,a,l a,n,l	**Branch Decrementing Not Zero** - The first operand is decremented by one, or by the value of the second operand if there are three operands. Then if the first operand is not equal to zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BDZ	a,l a,a,l a,n,l	**Branch Decrementing Zero** - The first operand is decremented by one, or by the value of the second operand if there are three operands. Then if the first operand is zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.

BE	a,a,l a,n,l n,a,l	**Branch If Equal** - If the value of the element or literal of the first operand is equal to the value of the second operand, a branch is taken to the label. The contents of both operands are treated as two's complement integers. If the operands are of the same size, and are identical, then the branch is taken. Otherwise, the sign bit (highest-order bit) of the smaller operand is extended to the left until the operands are the same size, and if the two equal size elements are identical, the branch is taken. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
	r,r,l r,s,l s,r,l s,s,l	If the register or storage register named by the first operand and the register or storage register named by the second operand are equal (that is, point to the same character), then a branch is taken to the label.
BH	a,a,l a,n,l n,a,l r,r,l r,s,l s,r,l s,s,l	**Branch If Higher** - If the value of the first operand is arithmetically higher than (>) the value of the second operand, a branch is taken to the label. The contents of both operands are treated as two's complement integers. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BHE	a,a,l a,n,l n,a,l r,r,l r,s,l s,r,l s,s,l	**Branch If Higher or Equal** - If the value of the first operand is arithmetically higher than or equal to (>=) the value of the second operand, a branch is taken to the label. The contents of both operands are treated as two's complement integers. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BHEZ	a,l n,l	**Branch If Higher or Equal to Zero** - If the value of the first operand is arithmetically higher than or equal to zero (>=0), a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BHZ	a,l n,l	**Branch If Higher than Zero** - If the value of the first operand is arithmetically higher than zero (>0), a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.

BL	a,a,l a,n,l n,a,l r,r,l r,s,l s,r,l s,s,l	**Branch If Less than** - If the value of the first operand is arithmetically less than (<) the value of the second operand, a branch is taken to the label. The contents of both operands are treated as two's complement integers. If the operands are not of the same size, the sign bit (highest-order bit) of the smaller operand is extended to the left until the operands are the same size. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BLE	a,a,l a,n,l n,a,l n,r,l r,r,l r,s,l s,r,l s,s,l	**Branch If Less than or Equal** - If the value of the first operand is arithmetically less than or equal to (<=) the value of the second operand, a branch is taken to the label. The contents of both operands are treated as two's complement integers. If the operands are not of the same size, the sign bit (highest-order bit) of the smaller operand is extended to the left until the operands are the same size. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BLEZ	a,l n,l s,l	**Branch If Less than or Equal to Zero** - If the value of the operand is arithmetically less than or equal to zero (<=0), a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BLZ	a,l n,l s,l	**Branch If Less than Zero** - If the value of the operand is arithmetically negative, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BNZ	a,l n,l s,l	**Branch If Not Zero** - If the value of the operand has any value other than zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
BSL	l m	**Branch and Stack Location** - The location of the next instruction is pushed on the return stack, then a branch is taken to the location specified by the label. This branch is used to make a subroutine call. The stack pointer element RSCWA is incremented by 4. The address of the instruction following the BSL instruction is moved to the 4-byte field in the process' PCB pointed to by the return stack pointer. Next, a branch is taken to the entry point (BSL m), or program label (BSL l). If the stack overflows, the process aborts to the debugger with a "RTN STACK FORMAT ERROR" message.
BSL*	t	**Branch and Stack Location Direct** - The next instruction's location is pushed onto the return stack. The operand is loaded into T0 and a BSLI instruction is executed.

BSLI	*	**Branch and Stack Location Indirect** - The location of the next instruction is pushed on the return stack, then a branch is taken to the mode whose address is in the lower 2-byte tally, T0, of the accumulator.
BSTE	r,r,n,l	**Branch If Strings Equal** - If two strings compare up to a delimiter, the branch is taken to the label. Both registers are incremented, then the bytes addressed by each register are compared logically. If the characters are equal and logically lower than the literal n, the increment and comparison is repeated. When characters are equal and logically greater than or equal to the literal n, the strings are considered equal and a branch is taken to the label. The strings are also considered equal and a branch taken if the characters compared are unequal, but both characters are greater than or equal to the literal n. The registers now address the string delimiter. In other cases the strings are considered unequal and the instruction terminates by falling through to the next instruction. The registers now address the dissimilar byte, not the delimiter. The function of the literal n is to specify a lower boundary for the delimiter that is considered to terminate the strings. Any character that is found to be logically greater than or equal to the literal n is considered to terminate the string. The strings do not have to be terminated by the same delimiter.
BSTU	r,r,n,l	**Branch If Strings Unequal** - If two strings do not compare up to the delimiter, then a branch is taken to the label. Operation is similar to BSTE. In other cases, the strings are considered unequal, and the instruction terminates by falling through to the next instruction.
BU	a,a,l a,n,l n,a,l	**Branch If Unequal** - If the value of the element or literal of the first operand is not arithmetically equal to (#) the value of the second operand, a branch is taken to the label. The contents of both operands are treated as two's complement integers. If the operands are not of the same size, the sign bit of the smaller operand is extended to the left until the operands are the same size. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
	r,r,l r,s,l s,r,l s,s,l	If the register or storage register of the first operand and the register or storage register of the second operand do not address the same character, then a branch is taken to the label.

BZ	a,l n,l r,l s,l	**Branch If Zero** - If the value of the operand is equal to zero, a branch is taken to the label. For this internal branch, the label must reside in the same mode, in the same frame as the branch instruction.
CHR	c	**ASCII Character** - This assembler directive, used in parameter tables and monitor assembly code, generates an in-line ASCII or binary character of one byte storage. {l} CHR C'A' Generates x'41' {l} CHR X'FF' Generates binary 1111 1111
CMNT	n	**Comment** - The statement in which this directive appears is treated as a comment. No object code is generated.
DEC	a a,a a,n	**Decrement** - The first operand is decremented by one. For the two operand form, the first operand is decremented by the value of the second operand.
DEC	r r,a r,n	**Decrement Register** - The address of the register is decremented by one. If the register is in linked format and originally pointed to the first data byte of the frame and the backward link of the current frame is zero, the register attaches to data byte zero of the current frame. Otherwise, an attempt is made to attach the register to the last data byte of the frame pointed to by the backward link of the current frame. "ILLEGAL FRAME ID" is an error which can be detected in this case.
	s s,a s,n	With a storage register, the displacement portion of the storage register is decremented by one, or by the two's complement integer contained in the second operand.
DEFx		**Define Element** - This assembler directive defines the element whose symbolic name is given by the label and whose type is given by the character (x) following "DEF". All forms, except DEFN, require two operands, which may be symbolic. The first specifies a base register; the second operand specifies displacement. Everything defining storage should be specified as one-half its actual value, if it is not a one-byte definition such as half tallies and characters (excluding literals and labels).

{1} DEFB	r,n	Defines a bit.
{1} DEFC	r,n	Defines a character.
{1} DEFD	r,n	Defines a double-tally.
{1} DEFE	r,n	Defines a quad-tally.
{1} DEFF	r,n	Defines a triple-tally.
{1} DEFH	r,n	Defines a half-tally.
{1} DEFN	n	Defines a literal.
{1} DEFM	p,o	Defines a mode and entry point.
{1} DEFS	r,n	Defines a storage register.
{1} DEFT	r,n	Defines a tally.
{1} DEFT	a,l	Defines a tally element.
{1} DEFTL	r,n	Defines the lower tally of a dtly.
{1} DEFTU	r,n	Defines the upper tally of a dtly.

DIV a n

Divide - The sign bit of the accumulator (D0) is extended into the accumulator extension (D1) to form a 64-bit dividend. The accumulator is then divided by the operand, resulting in a 32-bit quotient in D0 and a 32-bit remainder in D1. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is the sign of the dividend. The contents of the operand are not changed. A divisor of zero, which is technically illegal, returns D0 unchanged and D1 equal to zero.

DIVX a n

Divide Extended - The sign bit of the 6-byte operand is generated by extending the sign bit of the original operand. Then the contents of FP0 are divided by the operand, resulting in a 48-bit quotient in FP0 and a 48-bit remainder in FP1. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is the sign of the dividend (original FP0). A divisor of zero returns FP0 unchanged and FP1 zero.

DTLY d

Define Double Tally - This assembler directive is used in parameter tables and monitor code. 4 bytes of storage are reserved to and define a double tally containing the value specified by the operand, which may be symbolic.

ECHO.OFF *

Echo Off - Terminals attached to the Pick System usually operate in an "echo-plex" mode. In this mode, a character input from a keyboard is not displayed on the terminal video until the computer receives the character and sends it back to the terminal. Characters input from the terminal are normally echoed automatically. This instruction disables echoing by zeroing the echo bit in the active process' PIB.

ECHO.ON	*	**Echo On** - This instruction enables terminal echo by setting the echo bit in the active process' PIB.
END	*	**End** - This assembler directive marks the end of a mode. The instruction pointer will not increment past this instruction in the mode.
ENT	m	**Enter External Mode** - A branch is taken to the entry point specified by the mode ID. The high order 4 bits of the mode ID (m) are the entry point number (0-15). The remaining 12 bits of the mode ID are the FID of the frame.
ENT*	t	**Enter Direct** - Branches to the entry point specified by the operand. The operand is loaded into T0, and an ENTI instruction is executed.
ENTI	*	**Enter Indirect** - A branch is taken to the entry point defined by T0.
EP	l	**Entry Point** - At the beginning of every mode is a jump table. Each entry point is an encoded address to which control will be transferred via a branch instruction. Each mode has a maximum of 16 entry points (0-15). The entry point label is an external label which consists of an entry point number and a frame number. This is commonly known as a mode ID.
EQU	*	**Equate** - This instruction creates a symbol label with the current location as its value. The label used with this directive is equated to the value specified by the operand, which may be symbolic.
	a n	The operand is equated to the symbol in the label field.
	*n	The symbol is created with the current location counter plus or minus the literal. By having n = -1, it is possible for the SRA instruction to address a text string at one byte before the start of the text string.
FRAME	n	**Frame Location** - The 4-byte literal n defines the frame number of the ABS area into which the mode is loaded. This instruction must be on line number 1 of a mode.

FRM.LOCK	r	**Frame Lock** - Makes the frame pointed to by the register permanent in RAM. This instruction ensures that the frame is in a memory buffer, then locks that frame in memory by setting status flags to indicate it is not a candidate for being flushed to disk.
FRM.UNLOCK	r	**Frame Unlock** - Reverse the status of the frame locked in RAM. If the frame pointed to by the register was not locked, no action is taken.
FTLY	f	**Define Triple Tally** - This assembler directive is used in parameter tables and monitor code. 6 bytes of storage are reserved. The triple tally defined contains the value specified by the operand, which may be symbolic.
GET.TIME	*	**Get Time** - This instruction returns the system time and date in internal format. The date, in days since December 31, 1967, is returned in T2. The time, in milliseconds since midnight, is returned in D0.
HALT	*	**Halt** - This instruction terminates code execution with the abort "ILLEGAL OPCODE."
HTLY	n	**Define Half Tally** - This assembler directive is used in parameter tables and monitor code to reserve 1 byte of storage. The half tally defined contains the value specified by the operand, which may be symbolic.
INC	a	**Increment** - This instruction increments the operand. For the single operand form, the contents of the operand is incremented by one.
	a,a a,n	For the two operand form, the contents of the first operand is incremented by the contents of the second operand.
	r r,a r,n	For the address register form; the register is incremented by one. If the new address is not in the same buffer, either a "CROSSING FRAME LIMIT" error occurs, if the register is in unlinked format, or an attempt is made to attach the register to the first data byte of the frame pointed to by the forward link of the current frame. In this case, "FORWARD LINK ZERO" and "ILLEGAL FRAME ID" are errors which can be detected if they occur.
	s s,a s,n	For the storage register form, the displacement portion of the register is incremented by one or by the two's complement integer contained in the second operand. No address errors are detected.

INCLUDE	i	**Include item** - This assembler directive instructs the Assembler to include the item into the mode. INCLUDE is used as a means of loading common definitions into modes without redefining in each mode.
LAD	r,r r,s s,r s,s	**Load Absolute Difference** - This instruction computes the difference, in bytes, between the address pointed to by the register of the first operand and the address pointed to by the register or storage register of the second operand. The result is a positive integer in T0. The number of bytes between registers is computed as how many bytes from the address of the first operand + 1 to the byte address of the second operand. That is: R1 = 40000.1; R2 = 40000.B; thus T0 is computed B - 1 = A (decimal, 10). Both operands must address the same frame unless the frames are contiguously linked and the difference between the frame numbers is less than 32K.
LOAD	a	**Load** - For the arithmetic element, the contents of the element are loaded into the partition of the accumulator D0 that matches the element size. Then if the element is less than 32 bits the sign bit of the element is sign-extended to fill out the 32-bits of the accumulator D0. For example, if H7 contains x'80', D0 would contain x'FFFFFF80' after executing the instruction LOAD H7. 6-byte operands are loaded into FP0.
	m n	For the literal and modal address form, the 4-byte value of the literal is loaded into D0. No sign-extension occurs. For example, if n contains x'80', D0 would contain x'00000080' after executing the instruction LOAD n.
LOADX	a	**Load Extended** - For the arithmetic element, the element contents are loaded into the accumulator partition that matches the element size. If the element is less than 48 bits, (that is a half tally, tally, or double tally), the sign bit of the element is sign-extended to fill out the 48-bits of the accumulator, FP0.
	m n	For the literal and modal address form, the 6-byte value of the literal is loaded into FP0. No sign-extension occurs.
LOCK	r	**Lock** - This instruction is used for implementing semaphores. If the tally addressed by the register is zero, a lock is set by storing the lock number. This lock number is the line number + 1. If the tally addressed by the register is non-zero, that is, it is already locked by another process, the active process is deactivated and at the next snu, the lock is attempted again. This sequence is repeated until the lock is successful.

	r,l	For the two operand form, the lock is set like the single operand form except a branch is taken to the label if the lock has already been set by another process rather than waiting for it to unlock. The LOCK instruction concludes by incrementing the PCB tally element INHIBITH. This prevents the active process from breaking into the debugger until INHIBITH is decremented back to zero by the DECINHIB subroutine. This action offers some protection against another process acquiring a shared resource through a semaphore, then monopolizing that resource by breaking into the debugger. LOCK should never be used on the lock byte of a PCB as it will zero the ACF.
MBD	a	**Move Binary to Decimal** - The binary integer from element a is converted to an ASCII decimal string. Register R15 is incremented, then the ASCII string is moved to the location addressed by R15. The string is padded with blanks if BKBIT is set, or with zeroes if BKBIT is zero. The value of T4 specifies the minimum length of the string. This form of the instruction loads element a into the accumulator, then invokes the subroutine MBDSUB.
	a,r	For the two operand form, the element a is loaded into the accumulator. Then the second operand is moved to R15, a call is made to MBDSUB, then R15 is moved back into the register of the second operand.
	n,a,r	For the three operand form, the literal n specifies the minimum length of the decimal string. The literal is moved to T4, then the second operand is loaded into the accumulator. The third operand is moved to R15 and a call is made to MBDSUB, then R15 is moved back to register r. The string is padded with blanks if necessary for the length requirements. The string will exceed the length of the first operand literal if the ASCII number of the second operand is larger than the size limitation. This instruction is a macro expansion of MBDNSUB.
MBX	a,r c,r	**Move Binary to Hexadecimal** - The binary integer of element a, or character c, is converted to an ASCII hexadecimal string and stored at the register r + 1. The maximum character length of the string is specified by the number in H0. If B7 (the high-order bit of H0) is set, the string is padded with leading zeroes.
	n,a,r	The binary integer of the second operand is converted to an ASCII hexadecimal string of the length specified by the first operand literal. The string is padded with

blanks if necessary for the length requirements. If the integer is large, the string may exceed the length requirement. The MBX instruction assumes that H0 is the count of the number of characters to output.

MBXN	n,a,r n,s,r	**Move Binary to Hexadecimal** - This instruction expands to a macro which moves literal n to H0, sets B7 to force leading zeroes, then performs an MBX a,r instruction.
MCC	c,c c,r h,c h,r n,c n,h n,r r,c r,h r,r	**Move Character to Character** - The byte addressed by the first operand is moved to the byte addressed by the second operand.
MCI	c,r h,r n,r r,r	**Move Character Incrementing** - The register of the second operand is incremented by one, then the literal or character referenced by the first operand is moved to the byte addressed by the second operand.
	n,r,n	For the three operand form, the literal or tally of the third operand is moved to T0. The second operand destination register is incremented by one, then the character or literal referenced by the first operand is moved to the byte addressed by the second operand. Then T0 is decremented by one. The incrementation. character movement and decrementation of T0 loop continues until T0 becomes zero.
MDB	r,a	**Move Decimal to Binary** - The ASCII decimal character pointed to by the first operand is converted to a binary number and stored into the second operand. This is how ASCII decimal character strings are converted to a single binary integer. The first operand is incremented by one. The byte now addressed is examined. If it is numeric (ASCII 0-9, hex 30-39), the value of the second operand is multiplied by ten and the binary equivalent of the number pointed to by the first operand is added to the second operand. The process of increment, check, multiply and add is repeated until the byte addressed by the first operand is non-numeric. No check for decimal validity or overflow is made.

MFD r **Move Decimal String to Binary** - This instruction converts the ASCII decimal string at r+1 to a binary number accumulated in FP0.

FP0 is multiplied by ten for each character converted. It is not zeroed by the instruction before conversion begins. Normally FP0 should be zeroed before executing an MFD. The ASCII string may begin with a plus or minus sign and may contain one decimal point. Usually, the string is terminated by a system delimiter; however, with the exceptions already mentioned, conversion stops on any character that is not numeric. Parameters are passed to the MFD instruction in half tallies H6 and H7. The value of H6 determines the maximum number of ASCII decimal characters to convert (zero means 256). The value of the low-order four bits of H7 are a scaling factor, which is used as an exponent of ten to adjust the binary result to reflect a fractional precision. This scaling factor specifies the number of digits (0-15) to the right of an implied decimal point in the decimal equivalent of the binary result. The high-order four bits of H7 are used internally by the MFD instruction as follows:

Bit 0 - Use varies by implementation.
Bit 1 - Set if at least one numeric is converted.
Bit 2 - Set if a decimal point is detected.
Bit 3 - Set if the number is negative.

MFX r **Move Hexadecimal String to Binary** - This instruction converts the ASCII hexadecimal string at r+1 to a binary number accumulated in FP0.

FP0 is multiplied by sixteen for each character converted, and it is not zeroed by the instruction before conversion begins. FP0 should be zeroed before executing an MFX. A string is usually terminated by a system delimiter; however, conversion stops on any character that is not hexadecimal. A parameter is passed to the MFX instruction in the half tally H6 specifying the maximum number of characters to convert (zero means 256).

MIC r,c
r,r **Move Incrementing Character to Character** - The register of the first operand is incremented by one. Then the byte addressed by the first operand is moved to the character referenced by the second operand.

MII	r,r	**Move Incrementing Character to Incrementing Character** - Both register operands are incremented by one. Then the character addressed by the first operand is moved to the character addressed by the second operand.
	r,r,a r,r,n	For the three operand form, the literal of the third operand is moved into T0. The registers are incremented by one, the character addressed by the first operand is moved to the character addressed by the second operand, then T0 is decremented. The cycle of incrementation, character movement, T0 decrementation is repeated until T0 becomes zero.
MIID	r,r,v	**Move Incrementing Incrementing through Delimiter** - This instruction moves a string character by character to another register up to and including the delimiter of the source string. Both registers are incremented by one, then the character addressed by the first operand is moved to the character addressed by the second operand. The byte moved is then checked for a match with the variant byte. The process of register increment, character movement and character match check with the variant byte continues until the character moved meets the termination criteria specified by the variant byte. This instruction will alter the position of both registers by at least one location.
MIIDC	r,r,v	**Move Incrementing Incrementing through Delimiter Counting**- This instruction moves a string, character by character, to another register up to and including the delimiter of the string. T0 is decremented for each character moved. Both registers are incremented by one, the character addressed by the first operand is moved to the character addressed by the second operand, and T0 is decremented by one. The byte moved is checked for a match using the variant byte. This process of register increment, character movement, character match check is repeated until the character moved meets the termination criteria specified by the variant byte. The number of bytes moved is the difference between the original value of T0 and the final value of T0 at the termination of the instruction. If T0 was initially zeroed then the count is the negative value of T0. If T0 was initially 1 then the count is the negative value of T0 excluding the delimter. Both registers are incremented at least one.

MIIDT	r,r,v	**Move Incrementing Incrementing through Delimiter or to Count** - This instruction combines the functions of the MIID and MIIT instructions. Both registers are incremented, then the character addressed by the first operand is moved to the character addressed by the second operand, then T0 is decremented. The byte moved is checked for a match using the variant byte. This process of register increment, character movement, character match check, T0 decrement, and T0 not zero check continues until either T0 equals zero, or the byte moved meets the termination criteria specified by the variant byte, then the instruction terminates. If T0 is initially zero, no character movement occurs but both registers are incremented one.
MIIR	r,r	**Move Incrementing Incrementing to R15** - This instruction moves a string addressed by the first operand to the address of the second operand until the first register equals the address of R15. Both registers are incremented, then the character addressed by the first operand is moved to the character addressed by the second operand. Then the first operand is checked for an address match with R15. The process of increment, move and compare address is repeated until the first operand and R15 have the same address. If the first operand intially equals R15, no operation is performed and no register increments occur. If R15 is not ahead of and in the same string as the first operand, this instruction will not terminate resulting in a runaway register.
MIIT	r,r	**Move Incrementing Incrementing to Count** - This instruction moves a string of fixed length. T0 contains a byte count (up to 65,535) specifying the number of bytes to be moved. Both registers are incremented by one, then the character addressed by the first operand is moved to the character addressed by the second operand, then T0 is decremented by one. The process of register incrementation, character movement, and decrementation of T0 continues until T0 becomes zero. If T0 is initially zero, no operation is performed and no register increments occur.
MOV	a,a b,b m,a n,a	**Move** - The first operand replaces the value of the second operand.
	r,r	For the address register form, all eight bytes that define the register are copied into the second register.

	r,s	For the register to storage register form, the effective register of the address register replaces the contents of the storage register.
	s,r	For the storage register to register form, the contents of the storage register replace the address register. An attempt is made to attach the register to a frame and if the storage register is not legal, address errors will be detected at this time.
	s,s	For the storage register to storage register form, the contents of the first storage register replace the contents of the second storage register. No address errors are detectable.
MSDB	r	**Move String Decimal to Binary** - The ASCII decimal string at r+1 is converted to a binary number and stored in FP0. This instruction zeroes FP0, H6, and H7, then executes an MFD r instruction.
MSG#	n,n	**Message Number** - The value of the first operand literal is multiplied by 4, then 4 is added (n*4+4) and the result is moved to H3. The second operand literal is moved to H2. This instruction is typically used before calling the subroutine GET.MSG. The first operand specifies a message class. The second operand specifies a sequence number in that class.
MSXB	r	**Move String Hexadecimal to Binary** - The ASCII hexadecimal string at r+1 is converted to a binary number and stored in FP0. This instruction zeroes FP0, H6, and H7, then executes an MFX r instruction.
MTLY	n,m n,n	**Mode ID** - This assembler directive reserves storage and sets up the symbol in the label field to be of type m, which is a mode ID. A mode ID consists of a four-bit entry point number and a twelve-bit frame number (FID). The first operand in the instruction is used to specify the entry point number, and must be in the range ASCII 0-15; hex A-F. The second operand is used to specify the frame number and may be a literal or a previously defined mode ID.
MUL	a n	**Multiply** - The contents of D0 are multiplied by the operand. The 4-byte result is stored in D0. The sign of the product is determined by the rules of algebra. No check for hexadecimal validity or overflow is made.

MULX	a n	**Multiply Extended** - The contents of FP0 are multiplied by the operand. The 6-byte result is stored in FP0. The sign of the product is determined by the rules of algebra. No check for hexadecimal validity or overflow is made.
MXB	r,c r,a r,s	**Move Hexadecimal to Binary** - The ASCII hexadecimal string addressed by the register is converted to a binary number and stored in the second operand. Then the byte is examined, and if hexadecimal (ASCII 0-9 A-F), the second operand is multiplied by sixteen and the binary equivalent of the number addressed by the register is added to the second operand. No check for hexadecimal validity or overflow is made.
NEG	a	**Negate** - The operand is negated by replacing its value with its two's complement. Thus, the sign of the operand is changed. For example, 10 becomes -10, -300 becomes 300, etc.
NEP	*	**Null Entry Point** - This instruction specifies an entry in the jump table for an entry point that has no value. Equivalent to a HALT instruction.
NOP	*	**No Operation** - This instruction increments to the next instruction without performing any operation.
ONE	a	**One** - The value of the operand is set to a numeric one. The low order bit is set. All other bits are zeroed.
OR	c,n r,n r,r	**Logical OR** - The byte referenced by the first operand is logically ORed with the mask byte referenced by the second operand. The result replaces the first operand. The second operand is unchanged.
ORG	l n * *n	**Origin** - An assembler directive that sets the location counter to the value of the operand, which may be symbolic.
READ	r	**Read Input Queue** - The next character from the terminal input queue replaces the byte addressed by the register. If the input queue is empty, the process is suspended until a character is received from the terminal. Characters transmitted by the terminal are automatically queued in the PIB for the terminal. Control characters are not echoed.

RQM	*	**Release Time Quantum** - The process releases its time slice, getting deactivated and the next process is selected. The process must then wait for the next select-next-user (SNU). The instruction is used in a program loop that waits for an external event to occur. No assumption can be made about the timing of an RQM instruction.
RTN	*	**Return from Subroutine** - A location is popped off the stack by decrementing the return stack (RSCWA) by 4. Then a branch is made to that address. If the stack pointer underflows, the debugger is entered with a "RTN STACK EMPTY" abort.
SB	b	**Set Bit** - The referenced bit is set to one.
SET.TIME	*	**Set System Time and Date** - This monitor call sets the time and date in internal format. FP0 is set with T3 equal to the date in the number of days since Dec 31, 1967; D0 equal to the time as number of milliseconds since midnight.
SETUP.BAUD	*	**Set Baud** - This instruction sets the baud rate (that is, transmit/receive rate) of the RS-232 communications line associated with the PIB whose number is in T1. A negative number in T1 indicates the line attached to the active process. The baud rate is in T0. Standard baud rates are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200.
SETUP	r,n,d r,n,n r,t,d	**Setup Register** - The first operand is setup to the frame specified by the Frame ID in the second operand. The third operand gives the displacement within the frame.
SETUP0	r r,d r,n r,d,n	**Setup Register to Link Field** - The first operand is setup to the specified frame. For the single operand form, the FID is assumed to be in D0. For the two operand form, the Frame ID is in the second operand. Since the displacement is set to zero, the register addresses the first byte in the frame, which is the Link Field. The third operand gives the displacement within the frame.
SETUP1	r r,d r,n r,d,n	**Setup Register to Data Byte** - The first operand is setup to the specified frame. For the single operand form, the FID is assumed to be in D0. For the two operand form, the Frame ID is in the second operand. Since the displacement is set past the Link Field, the register addresses the first Data Byte in the frame. For the three operand form, the third operand is added to the FID in D0.

SHIFT	r,r	**Shift** - The byte addressed by the first operand is shifted right one bit. A zero bit is shifted in on the left. The shifted byte replaces the byte addressed by the second operand.
SICD	r,v	**Scan Incrementing Counting Delimiters** - The function of this instruction is to position the register at a specified point within a data structure containing several levels of delimiters in a minimal number of instructions. The register typically points one character before where the scan is to begin. T0 contains the delimiter count. The variant byte specifies the scan mode and termination criteria. If T0 is initially zero, or one, if bit 0 of the variant byte is set, the instruction terminates immediately without incrementing the register. The scan will unconditionally stop on a Segment Mark (X'FF'). Termination is considered normal if the specified number of delimiters are counted. With normal termination, T0 is zero and the register points at the last delimeter counted. Termination is considered abnormal if the instruction terminates before the specified number of delimiters are counted (for example, by reaching a delimiter logically greater than the one specified). With an abnormal termination, the count in T0 is the number of delimiters that must be inserted to create the data structure which was initially implied. Also, the register is decremented by one character position in preparation for a subsequent character string instruction. The variant byte is used differently for this instruction from the way it is used for other character string instructions. The format for SICD is detailed in the Character String Instructions section.
SID	r,v	**Scan Incrementing to Delimiter** - This instruction is used to find the end of a string, or to scan a string to find the first or last occurrence of a character in the string. The first operand is incremented. Then the byte addressed by the register is checked for a match with the variant byte. The scan continues until the character addressed by the first operand meets the termination criteria specified by the variant byte. This instruction will alter the position of the register by at least one location.

SIDC	r,v	**Scan Incrementing to Delimiter Counting Down** - This instruction scans a string from a register to a delimiter and keeps a count of the number of bytes scanned. The register of the first operand is incremented one. Then T0 is decremented one, then the byte addressed by the register is checked for a match with the variant byte. The process of scan, decrement T0, check match continues until the byte addressed by the register meets the termination criteria specified by the variant byte. The number of bytes scanned is the difference between the value of T0 before and after the instruction. This instruction will alter the position of the register by at least one location.
SIDT	r,v	**Scan Incrementing to Delimiter or to Count** - This instruction scans a string from a register to a delimiter or until T0 becomes zero. The register of the first operand is incremented by one. Then T0 is decremented by one. Then the byte addressed by the register is checked for a match with the variant byte. The process of scan, T0 decrement, byte match test continues until the byte addressed by the register meets the termination criteria specified by the variant byte or T0 becomes zero. No operation is performed if T0 is initially zero and no register incrementation occurs.
SIT	r	**Scan Incrementing to Count** - This instruction scans the register forward the number of bytes specified by the contents of T0. The register is incremented and T0 is decremented until T0 becomes zero. No operation is performed if T0 is initially zero. This instruction is logically equivalent to the instruction "INC r,n", however, the SIT instruction can be used to force usage of exception mode processing via XMODE if the register increments to the end of a linked frame set.
SLEEP	r	**Sleep** - The tally addressed by the register is zeroed (used to clear tally locks), then the active process is put to sleep, that is, deactivated and not reactivated, only when the system time equals the internal time in D0. Sleep should never be done on the lock byte of a PCB as it will result in zeroing the ACF.
SR	r,a r,s	**Set Register to FID** - This assembler directive reserves storage and sets up the symbol in the label field to be of type s, storage register. The first operand specifies the displacement of the generated byte address. The second operand is the FID. If the high-order bit of the value of the second operand is set, the byte address is in unlinked format; if zero, it is in linked format.

SRA	r,a r,c r,l r,s	**Set Register to Address** - The first operand is set addressing the first byte, that is, the leftmost or high-order byte, addressed by the second operand.
STORE	a s	**Store Accumulator** - The portion of the accumulator matching the size of the operand is stored in the operand. The accumulator is not changed.
SUB	a n	**Subtract from Accumulator** - The value of the operand is subtracted from D0. The difference replaces the contents of D0.
SUBX	a n	**Subtract Extended** - The value of the operand is generated by extending the sign bit of the original operand, and then subtracting the 6-byte value from FP0. The difference replaces the contents of FP0.
TEXT	n{,n}	**Reserving Text** - Reserves storage for character or hexadecimal data strings, or a combination of the two. The data must be enclosed in single quotation marks and prefaced with the letter C to select character data, or X, to select hexadecimal data. Combined forms of the two data types are separated by commas. {l} TEXT C'0123' Character string (x'30313233') {l} TEXT X'0123' Hexadecimal string (Binary 0000 0001 0010 0011) {l} TEXT C'abc',X'FF'{,...} Combined forms
TIME	*	**Get System Time** - This monitor call loads FP0 with T3 set to the date in number of days since Dec 31, 1967; D0 equal to the time as the number of milliseconds since midnight.
TLY	t	**Define Tally** - This directive is used in parameter tables and monitor code to reserve 2 bytes of storage. The tally defined contains the value specified by the operand, which may be symbolic.
WRITE	r	**Write to Output Queue** - The byte addressed by the register is placed into the terminal output queue. If the queue is full, the process is suspended until there is room in the queue.

XCC	r,r	**Exchange Characters** - The character addressed by the first operand is exchanged with the character addressed by the second operand.
XOR	c,n r,n r,r	**Logical Exclusive OR** - The byte addressed by the first operand is logically exclusive ORed with the mask byte referenced by the second operand. The result replaces the byte referenced by the first operand. The byte referenced by the second operand is unchanged.
XRR	r,r	**Exchange Register with Register** - The contents of the two registers are exchanged. In this instance, the operands are both source and destination operands. All eight bytes from each operand are copied to the other operand.
	r,s s,r s,s	The storage register instructions expand into macros which use R15 and MOV instructions.
ZB	b	**Zero Bit** - The referenced bit is zeroed.
ZERO	a s	**Zero** - The contents of the operand are replaced by zero.

System Software

PICK is a virtual memory machine with all of the virtual memory (i.e., disk) being directly addressable as if it where real memory.

The virtual memory of a PICK system resides on a disk drive divided into two types of frames: 2048-byte frames for the ABS area and 512-byte frames for the data area.

Executable Area (ABS)

The lower-numbered frames on the disk are ABS frames, which contain system software. The ABS area consists of executable object code. Software written in PICK Assembly Language is loaded onto disk in the executable area. The length of the executable area is a system generated parameter between 1023 and 4095. Frames 1 through 703 and frames 900 through 1023 of the executable area are reserved for current and future PICK software.

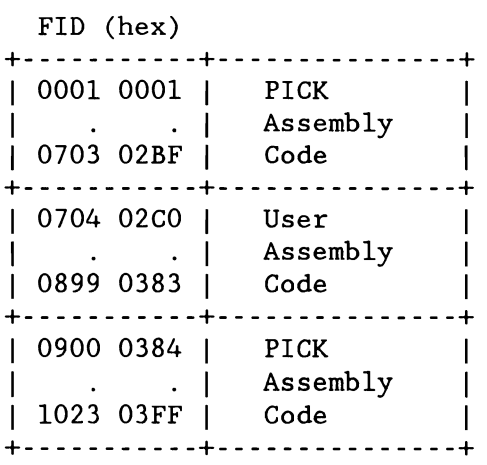


Figure 1 Executable Area (ABS)

Data Area

All frames after the ABS area are data frames. These frames contain process workspace, file and overflow areas.

Process Workspace Areas

The process workspace area contains the process' control blocks and primary and secondary workspaces.

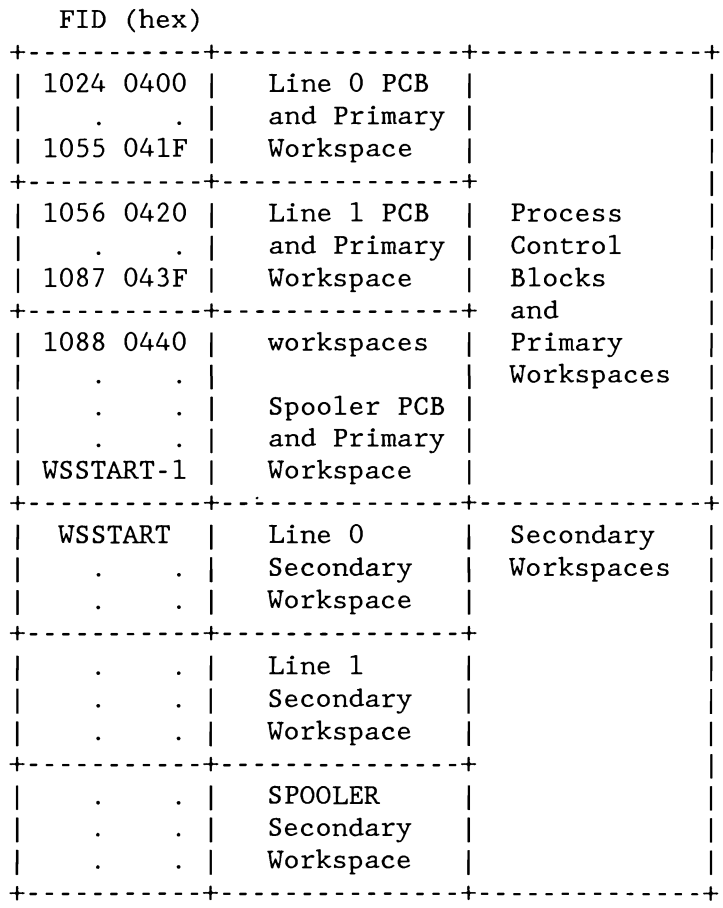


Figure 2 Process Workspace Area

Process Control Blocks

The control blocks for a process are the Primary Control Block (PCB), Secondary Control Block (SCB), Debugger Control Block (DCB) and Quadrenary Control Block (QCB).

The PCB contains storage for a process' registers, accumulators, pointers to the workspaces and additional storage elements. The SCB and QCB contain additional storage elements. The DCB contains storage elements for the Debugger.

The **MAP** verb can be used to diagram the elements of the process control blocks.

Primary Workspace Area

The primary workspace area is made up of 32 contiguous frames, pre-defined and available to each process. A quick-reference table of a process workspace is shown in Appendix B.

Each workspace is defined by a beginning pointer and an ending pointer, both of which are storage registers (S/R's). When the process is at the TCL level, all these pointers have been set to an initial condition. At other levels of processing, the beginning pointers should normally be maintained; the ending pointers may be moved by system or user routines. The address registers (A/R's) that are named after these workspaces (IS, OS, AF, etc.) need not necessarily be maintained within their associated workspaces; however, specific system routines may reset the A/R to its associated workspace. See Appendix C for the Register Conventions associated with these workspaces. Conventionally, a buffer beginning pointer addresses the byte preceding the actual location where the data starts. This is because data is usually moved into a buffer using one of the "move incrementing" type of instructions, which increment the A/R before the data movement.

Secondary Workspace Area

The secondary workspace area is a set of contiguous, linked frames initialized by the system at coldstart or system generation time. Each process has three secondary workspaces: IS, OS and HS, usually of 127 frames each. WSSTART is the starting FID of the secondary workspaces, which continues to the end of the workspace area. Processes that require additional workspace acquire and release that space from the overflow pool through standard routines such as: GETOVF, GETBLK, GETSPC and RELOVF, RELBLK, RELCHN.

The starting FID of the workspace may be computed by:

WSSTART = PCB0 + (number of lines) * 32

Each line has (3) workspaces of WSSIZE contiguous frames.

Files and Overflow Areas

After the work area are the PICK files, beginning with the SYSTEM file. The base of the SYSTEM file, SYSBASE, is the beginning of the file space. On a newly generated or restored system, all other files on the system immediately follow the SYSTEM file. At the end of the files is the start of Available Space, (Overflow), which continues until the end of the disk, MAXFID. (See the left side of the files figure.)

On a running system, the Overflow area will become fragmented as frames are taken and returned to the Overflow pool. (See the right side of the files figure.)

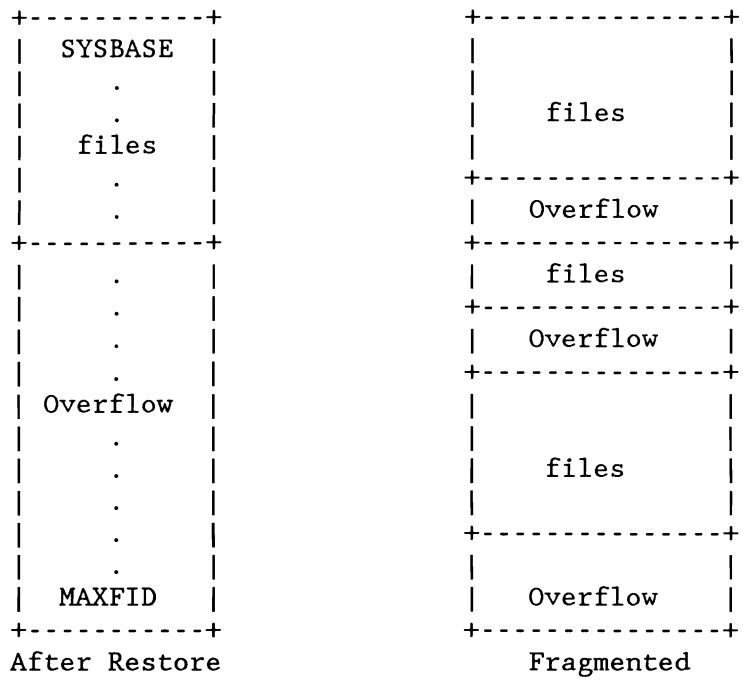


Figure 3 File and Overflow Areas

Beginning immediately after the Work Area, the remainer of the virtual memory, called the File Area, is available for the storage of data in files. The portions of the File Area not allocated to files are maintained as a pool of Available Space.

The beginning of the File Area is a system generation parameter called SYSBASE. If may be computed via the following general formula:

SYSBASE = PCB0 +
((number of processes) * 32) +
((number of processes) * (pre-assigned workspace) * 3)

Pre-assigned work-space is set to WSSIZE frames per process per work-space (WSSIZE is generally 127). Each process (including the SPOOLER) had 3 secondary workspaces of WSSIZE frames each.

As an example, a system with 17 channels (18 processes, including SPOOLER) will have the start of the File Area at frame:

1024 + (18 * 32) + (18 * 381) = 8458

The end of the File Area is the highest available disk frame, MAXFID.

File Area frames which are not allocated to the files are maintained as a pool of Available Space, often called Overflow. Available Space is used by the PICK system file management routines as additional data space and scratch workspace by other processors. The PICK system maintains a table of pointers identifying the Available Space, which may be either linked or contiguous. Contiguous Available Space consists of blocks of connecting frames, defined by starting and ending numbers, that can be taken out of the pool either singly or as a block. Linked Available Space can only be taken a frame-at-a-time. Conversely, space may be released by processors to the linked available pool a frame-at-a-time, or to the contiguous pool as a block.

At the conclusion of a FILE-RESTORE, there will be one principle block of Contiguous Available Space, extending from the end of the current data space through the last available data frame.

As the system obtains and releases Available Space and as files are created and deleted, Available Space becomes fragmented. At any particular time there may be several blocks of Contiguous Available Space, and a chain of Linked Available Space. Available frames will be placed in the Linked Available Chain only when there are 31 sets of Contiguous Available Space, representing the maximum that the system space management routines can maintain.

Logically, there is no differences between Available Space in linked chains and contiguous sets; however, certain processors, such as the CREATE-FILE processor, obtain frames from contiguous space only. Therefore, if Available Space is severely fragmented, there may actually not be enough space to create a large file.

System Subroutines

Assembly language programming is facilitated by a set of system subroutines that allow standardized interaction with the computers resources. These subroutines work with a set of addressing registers, storage registers, tallies, characters, bits, and buffer pointers, collectively called functional elements. In order to use any of these routines, therefore, it is essential that the calling routine set up the appropriate functional elements as required by the called routine's input interface.

The standard set of functional elements are pre-defined in the Virtual Permanent Symbol File (VIR.PSYM), and is therefore always available to the programmer. Included in the VIR.PSYM file are the mode ID's (program entry points) for the standard system subroutines. The symbols in the VIR.PSYM file should be treated as reserved symbols, although there is no reason that a symbol internal to an assembly language program cannot have the same name as a symbol defined in the VIR.PSYM file, so long as the symbol defined in the VIR.PSYM file is not referenced in the assembly language program.

Re-Entrancy

In practically all cases, the system software is re-entrant; that is, the same copy of the object code may be used simultaneously by more than one process. For this reason, no storage internal to an assembly language program should be utilized. Instead, the storage space directly associated with a process is used. This is part of the process's Primary, Secondary, Debugger, and Quadrenary Control Blocks.

An assembly language program may utilize storage internal to the program if it is to be used in a non re-entrant fashion; however, in most cases, the functional elements defined in the VIR.PSYM file will be sufficient.

In some cases an assembly language program may be required to use storage internal to the program. These programs should be set up to be executable by only one process at a time; that is, the code is locked while a process is using it, and any other process attempting to execute the same code waits for the first process to unlock it.

The following sequence illustrates this:

```
          ORG     X'0000'
LOCKBYT   TLY     X'0000'        INITIAL CONDITION FOR LOCK BYTE
          CMNT    *              (RESERVE TALLY FOR LOCK BYTE)
          .
          .
          .
          SETUP0 R15,LOCKBYT     POINT R15 TO LOCK BYTE AT X'0000'
          LOCK   R15             LOCK IF POSSIBLE ELSE WAIT
          BSL    DECINHIB        ENABLE BREAK KEY
          .
          .
          .
```

Frame Usage

Frames are referenced in linked or unlinked mode. In unlinked mode a frame contains 512 bytes of addressable space, physical bytes 0 through 511. In linked mode a frame contains 500 bytes of addressable space, physical bytes 12 through 511, and a 12 byte header, physical bytes 0 through 11, containing link information.

OFFSET	LINKED MODE	UNLINKED MODE
< 0	byte in previous frame uses backward links, if backward links = 0 'BACKWARD LINK ZERO' if backward links > MAXFID 'REFERENCING ILLEGAL FRAME'	'CROSSING FRAME LIMIT'
0	byte 511 in previous frame	physical byte 0 of frame
1-500	physical bytes 12-511	
1-511		physical bytes 1-511
> 511	byte in next frame uses forward links, if forward links = 0 'FORWARD LINK ZERO' if forward links > MAXFID 'REFERENCING ILLEGAL FRAME'	'CROSSING FRAME LIMIT'

Register Format

The Virtual machine has 16 registers -- R0 through R15. Each register is 8 bytes long, defined within the process's PCB/SCB/DCB/QCB. The format of these registers is as follows:

BYTE OFFSET	DESCRIPTION
0-1	real memory segment
2-3	real memory offset
4	flags
4:0	bit 0 indicates frame is LINKED/UNLINKED
4:1	bit 1 indicates frame is ATTACHED/DETACHED
5-7	FRAME ID (FID)

Storage Register Format

Each storage register is 6 bytes in length, defined within the PCB/SCB/DCB/QCB. The format of these registers is as follows:

```
 BYTE
OFFSET  DESCRIPTION
 0-1    real memory offset
 2      flags
 2:0    bit 0 indicates frame is LINKED/UNLINKED
 2:1    bit 1 indicates frame is ATTACHED/DETACHED
 3-5    FRAME ID (FID)
```

Addressing Modes

The Virtual machine supports 4 addressing modes:

```
MODE NAME  MODE DESCRIPTION
IMMEDIATE  the data is contained in the operand
           MCC C'G',R15
DIRECT     the contents of one of the 16 registers is
           referenced
           MOV R7,R8
INDIRECT   the contents of one of the 16 registers (R0 - R15)
           is used to reference the data
           MCC R7,R8
           MIC R7,R8
           MII R7,R8
RELATIVE   the data is referenced using one of the 16
           registers (R0 through R15) and an offset
           MOV R0;10,VAR
```

PSYM Format

The PSYM files (PSYM, VIR.PSYM, and NAT.PSYM) contain global symbols:

```
SYMBOL  DESCRIPTION       LENGTH  PHYSICAL OFFSET
  B     bit               1 bit   =OFFSET
  C     character         1 byte  =OFFSET
  D     double tally      4 bytes =OFFSET*2
  F     triple tally      6 bytes =OFFSET*2
  H     half tally        1 byte  =OFFSET
  R     register          8 bytes =OFFSET*2
  S     storage register  6 bytes =OFFSET*2
  T     tally             2 bytes =OFFSET*2
```

The PSYM symbols have the following format:

```
line 1  B/C/D/H/L/F/S/T
line 2  offset
line 3  base register

line 1 R
line 2 register number (0 through 15)

line 1 M
line 2 enter point
line 3 frame id

line 1 N
line 2 value
```

Subroutine Categories and Descriptions

Many of the standard Pick System software routines are available to assembly language programmers. These routines, which may be considered functional extensions to Pick Assembly Language, are particularly useful for interacting with the file system and for performing I/O with the terminal, tape, and printer.

Unless otherwise specified, each routine is called as a subroutine using a BSL instruction and returns to the calling program by way of an RTN instruction.

The available routines are described in the following pages using the structure shown below. All sections of the structure are not present in every description, but if present, they appear in the order given.

SUBROUTINE.NAME

This section gives a brief description of the functional operation of the system routine.

Input Interface

This section describes input parameters which the calling program should pass to the system routine.

Output Interface

This section describes output parameters passed back to the calling program from the system routine.

Element Usage

This section lists elements which may be altered during execution of the system routine. A user should also assume that D0, D1, D2, T4, T5, R14, R15, SYSR0, SYSR1, and SYSR2 may be altered by any system routine.

Subroutine Usage

This section lists other subroutines which may be called by the system routine and specifies the maximum additional subroutine nesting level.

Exits

This section specifies subroutines which may be called upon completion of the current system routine.

Errors

This section describes any error conditions which may be detected by the system routine.

Example

This section gives an example of how to use this subroutine.

Conversion Subroutines

These routines do data conversion. MBDSUB is a stand-alone subroutine which converts a binary number to a decimal ASCII string. It is used by the different variations of the MBD instruction.

The other routines described in this section are elements of the PICK System Conversion Processor, which performs the data conversions defined for ACCESS and PICK/BASIC, and also provides a facility for implementing user-written conversion routines. The Conversion Processor, invoked by calling CONV, determines which user routine is requested, transfers to it and returns through CONVEXIT.

CONV	conversion processor
CONVEXIT	conversion processor exit
MBDSUB	convert binary to decimal
MBDSUBX	convert binary to decimal
CVDreg	convert ASCII decimal to binary
CVXreg	convert ASCII hex to binary
ASCII	convert ASCII to EBCDIC
EBCDIC	convert EBCDIC to ASCII

File I/O Subroutines

The file I/O routines are complementary and interrelated. Some of them are used internally by others. Notice that the interfaces for GETITM, RETIX, and UPDITM are very similar.

A file is opened by calling the open routine, which defines the file by storing its base fid, modulo, and the appropriate flags in basemod. A file must be opened before calling any of the other file I/O routines.

Once a file is opened, a specific item is retrieved from the file by calling RETIX. All items of the file are retrieved by repetitively calling GETITM, which internally uses GNSEQI, GNTBLI, and RETIX to retrieve the items in the order in which they are stored. Items are added, deleted, or modified by calling UPDITM.

GETITM	get the next item
GMMBMS	get the master dictionary
GNSEQI	get the next sequential item
GNTBLI	get next table entry
DICTOPEN	open file
RETIX	retrieve item
UPDITM	update item
RDREC	read links of a FID
HASH	locate group into which item-ID would hash
LINK	initialize a set of contiguous frames
GLOCK	lock a group
GUNLOCK	unlock a group
GETACBMS	get base, modulo and separation of ACC file
CONFIG	get base, modulo and separation of SYSTEM file
SORT	ACCESS interface

Overflow Subroutines

The Overflow routines manage the pool of common available frames known as the system Overflow space.

ATTOVF	attach a frame from Overflow
NEXTIR	attach a frame from Overflow off IR register
RELOVF	release a frame to Overflow
GETOVF	get a frame from Overflow
DLINIT	get a set of contiguous frames from Overflow

System-Level Retrieval Subroutines

These subroutines retrieve system configuration parameters and allow access to the system time and date.

The TIME routine gets the current system time in hours, minutes, and seconds (that is, hh:mm:ss). DATE returns the system date in day, month, and year (that is, dd mmm yyyy). GMAXFID returns the largest legal frame number of the executing system configuration. The GPCB0 routine returns the frame number of the level 0 PCB for process 0.

DATE	get date
GMAXFID	get maximum frame number
GPCB0	get frame number of PCB0
TIME	get time
TIMDATE	get time & date
PRIVTST1	check for a minimum of privilege level "SYS1"
PRIVTST2	check for a minimum of privilege level "SYS2"
LINESUB	get process line number
SLEEP	put process to sleep for specified time
DECINHIB	enable break key

Tape Subroutines

These routines manipulate and perform the I/O for the virtual tape drive.

TPSTAT	get tape status
TPINIT	initialize the tape and its status word
TATT	attach the tape unit to current process
TDET	detach the tape unit from current process
WEOF	write end of file mark
TPREAD	read data from tape
TPWRITE	write data to tape
WTLABEL	write a standard label to tape
RDLABEL	read a standard label from tape

Terminal and Printer Subroutines

These subroutines perform terminal I/O and write data to printers, tape, and hold files.

READLIN issues a prompt character, then reads the user's response into the ib buffer. Several editing functions are available as characters are being input. WRTLIN outputs a line of data from the OB buffer to the active process' terminal. It provides automatic footing, heading, and page numbering capabilities. However, if WRTLIN is called with LPBIT set, the line of data is not sent to the terminal, but is routed as specified by the SP-ASSIGN options that are in effect. These options, set by the SP-ASSIGN verb, may specify that the data be spooled to a printer, written to a hold file, written to tape, or simply discarded. WRTLIN calls SP.PPUT to accomplish the specified output.

The GET.MSG and PRINT routines retrieve and display system messages. The PRTERR routine retrieves a message from the system MESSAGES file and displays it, along with parameters which may be provided. SYSTEM.CURSOR performs cursor positioning and forms control for either the terminal or printer.

GET.MSG	get message
PCRLF	print cr lf on terminal
PRINT	print message on terminal
PRTERR	print error message
READLIN	read line from terminal
SYSTEM.CURSOR	terminal/printer control
WRTLIN	write line to terminal
GETBUF	read a line; process control characters
PONOFF	toggle echo on or off
BLOCK.LETTERS	block print a string
GETOPT	process and option string
SETTERM	set term type
RESETTERM	set term type
SETLPTR	set line printer as output device
SETUPTERM	setup terminal/printer characteristics
INITTERM	setup terminal/printer characteristics

Workspace Routines

These routines manipulate a process' workspace.

PINIT	initialize PCB, SCB and DCB
GETUPD	initialize the UPD register triad
TSINIT	initialize the TS register triad
ISINIT	initialize all workspace
WSINIT	initialize BMS, AF, CS, IB, OB, TS triads
HSISOS	initialize HS, IS, OS register triads
ISOS	initialize IS and OS register triads
XISOS	swap IS and OS register triads

Wrapup Routines

These routines do wrapup processing, which consists of processing the history string, closing open print files, releasing temporary Overflow space, and initializing certain process-related elements in preparation for beginning another task.

WRAPUP is the primary wrapup routine. MD99, MD992, MD993, MD994, and MD995 are entries to WRAPUP which offer convenient interfaces for generating error messages.

Substrings within the history string may specify retrieving and displaying error messages from the MESSAGES file, updating file items, and deleting file items.

MD99	message numbers in REJCTR, REJ0, REJ1; uses RMODE
MD992	message number in C1, parameter in D9
MD993	message number in C1, parameter in C2
MD994	message number in C1, parameter at R4
MD995	message number in C1, parameter at BMSBEG
MD999	process history string; uses RMODE
LOGOFF	wrapup a process and log it off

ASCII

Convert a character from ASCII to EBCDIC.

Input Interface

IB	R	Points to the character to be converted.

Output Interface

IB	R	Points to the same location but now the character is EBCDIC equivalent.

Element Usage

R15	R
T0	T

Subroutine Usage

none

ATTOVF - ATTSPC

Attach Overflow - Obtain a frame from the overflow table and link it to the frame specified in double tally RECORD. The forward link field of the frame specified in RECORD is set to point to the overflow frame obtained. The backward link field of the overflow frame is set to the value of RECORD. The other link fields of this overflow frame are zeroed.

ATTSPC - functions the same as ATTOVF except that if no overflow is available, the user is prompted to try again or quit.

Input Interface

RECORD	D	Contains the FID of the frame to which an overflow frame is to be linked.

Output Interface

OVRFLW	D	Contains the FID of the overflow frame if obtained, or zero if no more frames are available.

Element Usage

INHIBITH	B	Incremented while overflow table is accessed.
D0	D	Used by GETOVF
R14	R	Utility
R15	R	Utility

Subroutine Usage

GETOVF
Two additional levels of subroutine linkage required

BLOCK.LETTERS

This routine prints block letters on the terminal or line printer. It is used, for instance, by the TCL verbs **BLOCK-TERM** and **BLOCK-PRINT**.

Input Interface

IS	R	Points one byte before the first character to be output; the end of data is marked by the character pair SM Z (_Z); no space after the SM. If any element in the data string contains a SM, it must be terminated by a SB.
ZBIT	B	If set, output is directed to the terminal, otherwise output is passed to the spooler for line printer listing or other use.
OBSIZE	T	Contains the maximum number of characters on each output line.
OB	R	= OBBEG.
SBO	B	If set, no test for terminal or printer output is made, terminal or printer characteristics are not initialized, the output device is not advanced to top-of-form, and the heading is not set null. All these actions take place if SBO is not set.
AFBEG	S	
BMSBEG	S	Point to scratch area.
HSEND	S	At end of history string.
LISTFLAG	B	Output list/no list.
SMCONV	B	
NOBLNK	B	
LFDLY	T	As required by WRTLIN.
PAGSIZE	T	from term settings.
PAGSKIP	T	
PAGFRMT	B	Pagination control.

Output Interface

OB	R	= OBBEG.
PAGINATE	B	set.
PAGHEAD	S	Points to a null page heading (SM) at HSEND if SB0=0.

Element Usage

BITS	C
SC0	C
SC1	C
SC2	C
REJCTR	T
C1	T
CTR16	T
CTR17	T
CTR18	T
CTR19	T
D0	D
D1	D

BASE	D	
MODULO	T	Utility.
SEPAR	T	
IR	R	
UPD	R	
BMS	R	
AF	R	
OB	R	
CS	R	
TS	R	
R15	R	
SR4	S	
.		
.		
SR22	S	
CTR	T	Used by CVTNIR.
R14	R	Used by RETIX.
T7	T	Used by WRTLIN.
SYSR	S	

Subroutine Usage

RETIX; GBMS if the system file "BLOCK-CONVERT" is found; CVTNIR; WRTLIN; NEWPAGE if required; PRNTHDR if SB0=0; PCLOSEALL and SETLPTR if SB0=0 and ZBIT=0; SETTERM if SB0=1 or ZBIT=1.

Six additional levels of subroutine linkage required if "BLOCK-CONVERT" is a "Q"-code item in the master dictionary, otherwise five levels required.

Errors

BLOCK-SUB exits to WRAPUP (MD995 or MD99) under the following conditions:

Error Number	Error Type
520	Null input data.
521	Too many characters (more than nine) in a word to block.
522	BLOCK-CONVERT file missing or improperly defined in the master dictionary.
523	Block output would exceed page width.
524	An input character is not in the BLOCK-CONVERT file.
525	An input character is improperly formatted in the BLOCK-CONVERT file.

CONFIG - GPCB0

CONFIG and GPCB0 both retrieve the FID for PCB0.

Input Interface

none

Output Interface

T0	T	Holds the FID of PCB0.

Element Usage

R14	R
T0	T

CONV - CONVEXIT

Conversion Exit - These entry points are used to call the entire conversion processor as a subroutine, which will perform any and all valid conversions specified in the input string. Other entry points may be used to perform certain specific conversions. Multiple conversion codes are separated by VM's in the conversion string. Conversion is called by the ACCESS pre-processor to perform conversions on "input" data (in selection criteria), and by the LIST/SORT processor to perform "output" conversion.

CONV is the usual mode ID used to invoke conversion processing. The loop control structure uses CONVEXIT as the entry point to which any part of the conversion processor returns in order to check if more conversion is required (further VM's and conversion codes in the conversion string).

Input Interface

TSBEG	S	Points one before the value to be converted; the value is converted "in place", and the buffer is used for scratch space; therefore it must be large enough to contain the converted value; the value to be converted is terminated by any of the standard system delimiters (SM, AM, VM, or SVM).
IS	R	Points to the first character of the conversion code specification string for CONV; for CONVEXIT, points at least one byte before the next conversion code (after a VM) or AM at the end of the string, or to the AM; the code string must end with an AM; initial semicolons (;) are ignored.
MBIT	B	Set if input conversion is to be performed; zero for output conversion.
DBIT	B	As required by TRANSLATE.
DAF1	B	
XBIT	B	As required by FUNC.

Output Interface

TSBEG	S	Points one before the converted value.
TS	R	Points to the last character of the converted value; a SM is also placed one past this location; TS=TSEND=TSBEG if a null value is returned.
TSEND	S	
IS	R	Points to the AM terminating the conversion codes.

Element Usage

Element		Conversions Where Used
DBIT	B	F,T
XBIT	B	F
GMBIT	B	F
WMBIT	B	F
SB10	B	All
SB12	B	All
DAF1	B	T
DAF9	B	T
SC2	C	C,D,F,T
T3	T	F,MD
T4	T	D,F,MD,MT
T5	T	D,F,MD,MT
T6	T	D,F,M
T7	T	F,MD
CTR1	T	C,F,G,T
CTR12	T	F
CTR13	T	F
CTR20	T	All
CTR21	T	D,MD,T
CTR22	T	D
CTR23	T	D,MD
CTR28	T	T
D1	D	C,F,MT,T
D2	D	D,F,MD,MT
D3	D	MT
D7	D	F
D8	D	F
D9	D	F
FP0	F	F,MD
FP1	F	F,MD
FP2	F	F,MD
FP3	F	F
FP4	F	F
FP5	F	F
FPX (SYSR0)	F	F,MD,T
FPY	F	F,MD
BASE	D	T
MODULO	T	T
SEPAR	T	T
RECORD	D	T
SIZE	T	T
NNCF	H	T
FRMN	D	T
FRMP	D	T
NPCF	H	T

Element		Conversions Where Used
XMODE	T	C,F,MT,T
IR	R	T
BMS	R	T
R14	R	D,MD,MT,MX,T
R15	R	All
SYSR1	S	T
SYSR2	S	T
S4	S	T
S5	S	F
S6	S	C,T
S7	S	All
SR0	S	C,F
SR1	S	F
SR4	S	C,T

Subroutine Usage

CVTHIS for "U" conversions; QCORR for "G" conversions; TRANSLATE for "T" conversions; CONCATENATE for "C" conversions; additional subroutines as used by routines listed under "Exits" below, and by user-written routines.

The number of additional levels of subroutine linkage required depends on the conversions performed - see the documentation for the various conversion routines for more specific information.

NOTE: For "F" conversions, CFUNC may call CONV recursively.

User Exits Conversion Processing

The conversion processor will pass control to a user-written routine if a "uyxxx" code is found in the conversion string, where "xxx" is the hexadecimal mode ID of the user routine and where "y" is the entry point. This routine can then perform special conversion before returning. The input interface for the user routine will be identical to that described in the preceding section. After performing the conversion, the user routine should set up the output interface elements to be compatible with CONVEXIT, and then exit via an external branch to that point to continue the conversion process if multiple conversions are specified. further conversions from being performed. Elements used by the regular conversion routines may safely be used by user routines; however, if additional elements are needed, a complete knowledge of the processor that called CONV (LIST, SELECTION, etc.) will be necessary.

Exits

To IDATE for "D" conversions on input (MBIT=1); to ODATE for "D" conversions on output; to ICONVMD or OCONVMD for "MD" conversions on input or output; to CFUNC for "F" conversions; to TIMECONV for "MT" conversions; to HEXCONV for "MX" conversions; all these routines, however, return to CONVEXIT.

For output conversion, a null value returned causes an immediate end of conversion processing.

Errors

CONV exits to WRAPUP after setting RMODE to zero under the following conditions:

705	Illegal conversion code.
706	Illegal "T" conversion: format incorrect, filename cannot be found, etc.
707	DL/ID cannot be found for a "T" conversion file.

WRAPUP is also entered without setting RMODE to zero under the following error conditions:

708	Value cannot be converted by a "T" conversion.
339	Invalid format for input data conversion.

CVDreg - CVXreg

Convert Value from Decimal or Hexadecimal to Binary - Each subroutine converts a string that starts one past the register and is terminated by an invalid character. Decimal conversions are terminated by a non-decimal and hex conversions are terminated by a non-hexadecimal character. The converted value is stored in the accumulator FP0. The register will point to the terminating character upon exit. MSDB & MSXB are used to do the conversion.

Input Interface

Subroutine	Conversion	Register
CVDIB	decimal	IB
CVDIR	decimal	IR
CVDIS	decimal	IS
CVDOS	decimal	OS
CVDR15	decimal	R15
CVXIB	hexadecimal	IB
CVXIR	hexadecimal	IR
CVXIS	hexadecimal	IS
CVXOS	hexadecimal	OS
CVXR15	hexadecimal	R15

Output Interface

FP0	F	Contains the converted binary value.
CTR1	T	Contains the low-order two bytes of FP0. Except for CVDR15 and CVXR15.
NUMBIT	B	Set to one if terminated by a system delimeter or a decimal point, cleared to zero upon termination by all others.

Element Usage

none.

Subroutine Usage

MSDB; MSXB

DATE

DATE places the date into the area pointed to by R15. The format is DD MMM YYY.

Input Interface

R15	R	Points one before the location where the date will be stored.

Output Interface

R15	R	Points to the last character in the date.

Element Usage

R14	R
R15	R
D0	D

DECINHIB

DECINHIB is used to decrement the half tally INHIBITH in an attempt to re-enable the break key. Normally, this half tally is first incremented and then this subroutine is called at a later time to decrement it. DECINHIB decrements INHIBITH by one if it is not zero. If INHIBITH is already zero and a break key has been requested the debugger is entered. This mechanism will ensure that different routines requiring break inihibition may call each other without problems of break key reactivation.

Input Interface

none

Output Interface

INHIBITH	H	decremented as described above.

Element Usage

INHIBITH	H

Exits

To the debugger if INHIBITH is zero

DICTOPEN - FILEOPEN - GETFILE - OPENDD

These routines are used to set up the BASE, MODULO, SEPAR parameters for a file based on a given file name. The file name must be of the form: {DICT/DATA} Dictname{,Dataname}.

DICTOPEN opens only the dictionary portion of a file.

FILEOPEN & GETFILE open only the specified portion of a file.

OPENDD opens both the dictionary and data portions of the file. If only the dictionary portion of the file is specified then only the dictionary portion is opened.

Input Interface

IS	R	Points one character before the file name (any number of blanks) and of the form shown above. It must be terminated by a blank, a system delimiter or a character specified in SC0.
RTNFLG	B	Set to one if the routine should return to the calling program if the file cannot be opened. If zero and the file cannot be opened WRAPUP is entered.

Output Interface

BASE	D	Contains the base fid where the file begins, if found.
MODULO	T	Contains the modulo of the file, if found.
SEPAR	T	Contains the separation of the file, if found.
DBASE	D	(OPENDD only) Contains the base fid of the dictionary of the file, if found.
DMOD	T	(OPENDD only) Contains the modulo of the dictionary of the file, if found.
DSEP	T	(OPENDD only) Contains the separation of the dictionary of the file, if found.
FBASE	D	(OPENDD only) = BASE, if found.
FMOD	T	(OPENDD only) = MODULO, if found.
FSEP	T	(OPENDD only) = SEPAR, if found.
IS	R	Points one character past the file name
BMSBEG	S	Will have a copy of the file name without "DICT" or "DATA".
BMS	R	Points to am AM added after the file name.
RMBIT	B	Set to one if the file parameters are successfully retrieved.
DAF8	B	Set to one if only the dictionary portion was opened.

Element Usage

SC1 H
SC2 H
all elements used by RETIX.

Subroutine Usage

Seven additional levels of subroutine linkage may be required.

Exits

To WRAPUP if RTNFLG = 0.

DLINIT

DLINIT is used to obtain a block of contiguous overflow space for a file. After checking the input parameters and obtaining the necessary number of frames, if available, it enters DLINIT1 to initialize the frames. If not enough space is available for the file, DLINIT calls NOSPACE to determine if processing should be aborted.

Input Interface

MODULO	T	Contains the modulo and separation parameters for the file,
SEPAR	T	if MODULO is initially less than or equal to zero, it is set to eleven; if SEPAR is initially less than or equal to zero, it is set to one, and if initially greater than 127 it is set to 127.

Output Interface

BASE	D	Contains the beginning FID of a contiguous block of size MODULO*SEPAR if the space is available, otherwise unchanged.
OVRFLW	D	=BASE if the requested space is available, otherwise =O.
RMBIT	B	Set if the requested space is obtained, otherwise unchanged.

Element Usage

R14	R	
R15	R	Used by GETBLK.
INHIBITH	H	
D0	D	

Subroutine Usage

GETBLK; NOSPACE if the requested space is unavailable.

Three additional levels of subroutine linkage required.

Exits

To DLINIT1 if the requested space is obtained; to NSPCQ (WRAPUP) from NOSPACE if the space is unavailable and processing is aborted by the user.

DLINIT1

DLINIT1 initializes the link fields of a file as specified by its base, modulo, and separation parameters, and sets each group empty by adding an AM at the beginning (in the first data byte).

Input Interface

BASE	D	Contains the base, modulo, and separation
MODULO	T	of the file; one frame is linked
SEPAR	T	even if MODULO is less than or equal to zero.

Output Interface

R14	R	Points to the first data byte in the first frame of the last group in the file (set by LINK).
R15	R	Points to the last byte of the last frame of the last group in the file (set by LINK).
RECORD	D	One greater than the FID of the last frame of the last group in the file.
NNCF	H	= SEPAR-1.

Element Usage

CTR1	T	Utility.
FRMN	D	
FRMP	D	Used by LINK.
NPCF	H	

Subroutine Usage

LINK
One additional level of subroutine linkage required.

DPTRCHK

DPTRCHK is used to check if an item is a D pointer. The criteria is: Attribute 1 must start with a D, optionally followed by C,X,Y or Z. Attribute 2 through 4 must be numeric. There must be at least 10 total attributes.

Input Interface

R15	R	Must point on the 1st character of 1st attribute

Output Interface

R15	R	Remains unchanged
DBIT	B	Set if valid D pointer
XBIT	B	Set if DX
YBIT	B	Set if DY
CBIT	B	Set if DC

Element Usage

R15	R
SR15	S
D2	D
T3	T
FP0	f

Subroutine Usage

One additional level of subroutine linkage required.

EBCDIC

EBCDIC is used to convert a character from EBCDIC to ASCII.

Input Interface

IB	R	Points to the character to be converted.

Output Interface

IB	R	Points to the same location but now the character is ASCII equivalent.

Element Usage

R15	R
T0	T

Subroutine Usage

none.

GACBMS

GACBMS retrieves the base, modulo and separation of the system ACC file.

Input Interface

none

Output Interface

BASE	D	Contains the base fid of the ACC file.
MODULO	T	Contains the modulo of the ACC file.
SEPAR	T	Contains the separation of the ACC file.

Element Usage

same as FILEOPEN

Subroutine Usage

Seven additional levels of subroutine linkage may be required.

GETBUF

This routine accepts input data from the terminal and performs an editing on the characters obtained. GETBUF also prints an initial prompt character at the terminal before reading input. Control is returned when a non-editing control character is input, or when the number of characters specified in T0 or T1 are input. These keystrokes allow the entry of Pick delimiters:

Control-H	Logically backspaces the buffer pointer; echoes character in BSPCH.
Control-X	Logically deletes the entire input buffer; echoes CR/LF and prints the prompt character (>).
Control-R	Retypes the input line.
Rubout	Ignored; the character is echoed, but is not stored in the buffer.
Control-shift-K	Converts to internal delimiter SB; echoes [.
Control-shift-L	Converts to internal delimiter SVM; echoes /.
Control-shift-M	Converts to internal delimiter VM; echoes].
Control-shift-N	Converts to internal delimiter AM; echoes ^.
Control-shift-O	Converts to internal delimiter SM; Echoes _.

NOTE: The high order bit of all characters that are input is zeroed.

Input Interface

BSPCH	C	Contains the character to be echoed to the terminal when the back space key is pressed.
PRMPC	C	Character output as a "prompt" when input is first requested by GETBUF, and after certain editing operations.
T0	T	Contains the maximum number of characters accepted.
T1	T	Contains the maximum number of characters to be accepted.
R14	R	Points one byte before the beginning of the input buffer area.
R15	R	Points one byte before the beginning of the input buffer area.

Output Interface

R15	R	Points to the control character causing return to the calling routine.

Element Usage

D0

GETITM

This routine sequentially retrieves all items in a file. It is called repetitively to obtain items one at a time until all items have been retrieved. The order in which the items are returned is the same as the storage sequence.

If the items retrieved are to be updated by the calling routine (using routine UPDITM), this should be flagged to GETITM by setting bit DAF1. For updating, GETITM performs a two-stage retrieval process by first storing all item-ids (per group) in a table, and then using this table to actually retrieve the items on each call. This is necessary because, if the calling routine updates an item, the data within this group shifts around; GETITM cannot simply maintain a pointer to the next item in the group, as it does if the "update" option is not flagged.

An initial entry condition must also flag GETITM by zeroing bit DAF7 before the first call. GETITM then sets up and maintains certain pointers which should not be altered by calling routines until all the items in the file have been retrieved (or DAF7 is zeroed again).

NOTE: The Output Interface elements are functionally equivalent with those of RETIX.

Input Interface

DAF7	B	Initial entry flag: must be zeroed on the first call to GETITM.
DAF1	B	If set, the "update" option is in effect.
DBASE	D	must be the base fid of the file
DMOD	T	must be the modulo of the file
DSEP	T	must be the separation of the file
BMSBEG	R	Points one byte prior to an area where the item-id of the item retrieved on each call may be copied.
OVRFLCTR	D	Meaningful only if DAF1 is set, if non-zero, the value is used as the starting FID of the overflow space table where the list item-id is stored; if zero, GETSPC is called to obtain space for the table.

Output Interface

RMBIT	B	
SIZE	T	
R14	R	(See RETIX documentation).
IR	R	
SR4	S	
XMODE	T	
SR0	S	= R14 if DAF1 is set, otherwise as set by GNSEQI.
BMS	R	As set by RETIX if DAF1 is set, otherwise as set by GNSEQI.
DMSEND	S	= BMS if DAF1 is set, otherwise unchanged.
DAF9	B	= 0.

Element Usage

BASE	D	
MODULO	T	
SEPAR	T	
RECORD	D	Used by GETITM and other subroutines for accessing file data.
NNCF	H	
FRMN	D	
FRMP	D	
NPCF	H	
OVRFLW	D	Used by GETSPC if DAF1 is set and OVRFLCTR is initially zero.

These elements should not be altered by any other routine while GETITM is used:

DAF1	B	(See Input Interface).
DAF7	B	
DBASE	D	The beginning FID of the current group being processed.
DMOD	T	The number of groups left to be processed.
DSEP	T	Unchanged.
SBASE	D	The saved values of DBASE
SMOD	T	DMOD
SSEP	T	and DSEP when the routine was first called.
NXTITM	S	Points one before the next item-id in the prestored table if DAF1 is set, otherwise points to the last AM of the item previously returned.
OVRFLCTR	D	Contains the starting FID of the overflow space table if DAF1 is set, otherwise unchanged.

Subroutine Usage

RDREC, GNSEQI; GNTBLI (local), RETIX, and GETSPC (if OVRFLCTR = 0) if DAF1 is set. BMSOVF used with XMODE.
Four additional levels of subroutine linkage required

Errors

See RETIX documentation "Exits"; GETITM, however, continues retrieving items until no more are present even after the occurrence of errors.

GETOPT

This routine converts an option string to internal usage. Options are character strings or a single or pair of numeric strings. A pair of numeric strings are separated by a hyphen. Alphabetic options set their corresponding flags ("A" sets AFLG, etc.). All flags AFLG thru ZFLG are cleared upon entry. So are NUMFLG1 and NUMFLG2.

Input Interface

IR	R	Points one before the option string.

Output Interface

NUMFLG1	B	Set if D4 has a value
D4	D	holds the 1st numeric value
NUMFLG2	B	Set if D5 has a value
D5	D	holds the 2nd numeric value
AFLG	B	Set if A option found
.		
.		
ZFLG	B	Set if Z option found
RMBIT	B	Set if no errors are found in the option format, otherwise zeroed and all flags described above cleared.
IR	R	Points to the last character processed

Element Usage

D0 and D1

Subroutine Usage

Two additional levels of subroutine linkage required.

Example

```
:LIST MD (B5

BFLG = 1; NUMFLG1 = 1; D4 = 5; NUMFLG2 = 0
AFLG = 0; CFLG thru ZFLG = 0; D5 = 0
```

GETOVF - GETBLK - GETSPC

These routines obtain overflow frames from the overflow space pool maintained by the system. GETOVF and GETSPC are used to obtain a single frame; GETBLK is used to obtain a block of contiguous space (used mainly by the CREATE-FILE processor). The link fields of the frames obtained by a call to GETBLK are not reset or initialized in any way - this is a function of the calling routine. GETOVF and GETSPC zero all the link fields of the frame they return. These routines cannot be interrupted until processing is complete.

Input Interface

D0	D	Contains the number of frames needed (block size), for GETBLK only.

Output Interface

OVRFLW	D	If the needed space is obtained, this element contains the FID of the frame returned (for GETOVF and GETSPC) or the FID of the first frame in the block returned (for GETBLK); if the space is unavailable, OVRFLW = 0.

Element Usage

INHIBITH	B	
D0	D	Utility.
R14	R	
R15	R	

Subroutine Usage

SYSGET (but not used by GETOVF, if a frame is obtained from a multiple-frame block in the system overflow table): three internal subroutines; GETOVF called by GETSPC; NOSPACE called by GETSPC if no frames are available

One additional level of subroutine linkage required by GETOVF and GETBLK; three levels required by GETSPC.

Exits

For GETSPC; to NSPCQ if no more frames are available and processing is aborted by the user; this is a function of NOSPACE.

GETUPD

GETUPD initializes the UPD register triad to point to the UPD workspace (frame PCB+28).

Input Interface

None

Output Interface

UPD	R	Points to the first data byte of the 28th frame after the process's PCB.
UPDBEG	S	Points to the first byte of the above frame
UPDEND	S	Points to the last byte of the above frame.

GLOCK - GUNLOCK - GUNLOCK.LINE - GUNLOCK.ALL

GLOCK and GUNLOCK lock and unlock a group in a file. They are used to ensure that a file will not be updated by more than one process at a time. GLOCK will lock a group in a file so that any other attempt to lock the group will cause that process to hang until the lock is unlocked.

GUNLOCK frees the lock on the group.

GUNLOCK.LINE unlocks all the groups locked by this line.

GUNLOCK.ALL initializes the group lock tables.

Locking

First, the lock table is locked, then the group to be locked is entered into the table and the table is unlocked. If the group has already been locked by another process or the table is full of entries, then the terminal will beep until either the lock is free or an entry in the table becomes available.

Unlocking

First the lock table is locked, then the table is searched for the group to be unlocked. If found and it is associated with this process, the entry is made available. Finally, the lock table is unlocked.

Input Interface

RECORD	D	Contains the beginning FID of the group to be locked.

Output Interface

none

Element Usage

CH9	C
CTR1	T
R14	R
R15	R
SYSR0	S
SYSR1	S
SC2	H
D0	D

Subroutine Usage

One additional level of subroutine linkage required to either LOCK or UNLOCK

GMAXFID

GMAXFID returns the MAXFID of the system to the accumulator.

Input Interface

none

Output Interface

D0	D	Contains the MAXFID

Element Usage

R14	R
D0	D

GMMBMS

GMMBMS is used to retrieve the SYSTEM file's base, modulo and separation.

Input Interface

none

Output Interface

BASE	D	Contains the base FID of the SYSTEM file.
MODULO	T	Contains the modulo of the SYSTEM file.
SEPAR	T	Contains the separation of the SYSTEM file.

Element Usage

R14	R

GNSEQI

This routine gets the next sequential item from a file. If the pointer into the file (register NXTITM) is at the end of a group, it returns with bit RMBIT zero; otherwise it copies the item-id into the area specified by register BMS, updates NXTITM, sets RMBIT, sets registers pointing to the beginning and end of the item, and returns the item size in tally SIZE. If a non-hexadecimal digit is found in the item count field, or the computed item size is negative or zero, GNSEQI immediately returns to the routine which called it.

Input Interface

NXTITM	S	Points one before the beginning of the next item to be retrieved (or the AM at the end of the group).
BMS	R	Points one before the area to which the item-id is to be copied.

Output Interface

RMBIT	B	Set if an item was successfully retrieved, otherwise zeroed.
NXTITM	S	Points one before the following item or end-of-group AM if RMBIT is set, otherwise unchanged.
BMS	R	Points to an AM after the copied item-id if the item was retrieved, otherwise unchanged.
SR0	S	The initial value of NXTITM if not at the end of the group, otherwise unchanged.
IR	R	Points to the AM after the item-id if RMBIT is set; points to the AM before the item-id if SIZE is zero negative; points to the AM indicating end of group data if there were no more items in the group when the routine was called; points to the character in error if a non-hexadecimal character is found in the item count field.
SIZE	T	Contains the value of the item count field if RMBIT is set.
XMODE	D	= 0.

GNTBLI

This routine retrieves the next entry from a table consisting of strings (typically item-ids) separated by AM's, and terminated by a SM. On each call, the routine checks if its pointer (register NXTITM) is at the end of the table. If it is, the routine exits with bit RMBIT zero; otherwise the next table element is copied into the buffer specified by register BMS, NXTITM is set pointing to the following element, and RMBIT is set.

Input Interface

NXTITM	S	Points one before the next table entry (or SM).
BMS	R	Points one before the area to which the table entry is to be copied.

Output Interface

NXTITM	S	Points to the AM following the entry which was copied, if one was copied, otherwise one before the SM at the end of the table.
IR	R	= NXTITM if an element was copied, otherwise NXTITM + 1.
BMS	R	Points to an attribute mark one past the end of the entry copy, if present, otherwise unchanged.
RMBIT	B	Zeroed if NXTITM points to the end of the table when the routine is called, otherwise set.

HASH

This routine computes the starting FID of the group into which an item would hash.

Input Interface

BMSBEG	S	Points one character before the item-id
BASE	D	Contains the base of the file to hash into
MODULO	T	Contains the modulo of the file to hash into
SEPAR	T	Contains the separation of the file to hash into

Output Interface

RECORD	D	Contains the FID to which the item-id would hash to

Element Usage

R14	R
R15	R
D0	D

HSISOS

This routine sets up the register triads for the HS, IS, and OS workspaces as described below. It does not link frames in the workspaces.

Input Interface

R2	R	Points to the Secondary Control Block (PCB+1).
HS	R	Points to the beginning of the HS workspace (PCB+10).
HSBEG	S	= HS.
HSEND	S	
IS	R	Points to the beginning of the IS workspace (PCB+16).
ISBEG	S	= IS.
ISEND	S	Points to the last data byte in the primary OS workspace (3000 bytes past ISBEG).
OS	R	Points to the beginning of the OS workspace (PCB+22).
OSBEG	S	= OS.
OSEND	S	Points to the last data byte in the primary OS workspace (3000 bytes past OSBEG). The first byte in each workspace is set to X'00'.

Element Usage

D0

INITTERM - RESETTERM

These routines are used to intitialize terminal and line printer characteristics. RESETTERM is called from WRAPUP before reentering TCL; INITTERM is called from LOGON.

Input Interface

OBSIZE	T	Contains the value of the output (OB) buffer (RESETTERM only).
OBBEG	S	Points to the start of the OB buffer.

Output Interface

TOBSIZE	T	
TPAGSIZE	T	
POBSIZE	T	Initialized to default values, as by.
PPAGSIZE	T	SETUPTERM (INITTERM only).
PAGSKIP	T	
LFDLY	T	
BSPCH	C	
CCDEL	B	
SMCONV	B	
STKFLG	B	
PAGINATE	B	
NOBLNK	B	
LPBIT	B	=0.
TPAGNUM	T	
TLINCTR	T	
PPAGNUM	T	
PLINCTR	T	
PAGNUM	T	
LINCTR	T	
PAGHEAD	S	Contains zero in the frame field.
OB	R	= OBBEG.
OBSIZE	T	TOBSIZE.
R14	R	= OBBEG+OBSIZE.
OBEND	S	

ISINIT

ISINIT simply invokes WSINIT and HSISOS to initialize all the process workspace pointers.

Input Interface

See WSINIT and HSISOS documentation

Output Interface

BMS,AF,CS,IB,OB,TS,HS,IS,OS triads are initialized

See WSINIT and HSISOS documentation

Element Usage

D0

Subroutine Usage

WSINIT, HSISOS

Three additional levels of subroutine linkage required

LINK

LINK initializes the links of a set of contiguous frames. The frames will be linked contiguously forward and backward.

Input Interface

RECORD	D	Contains the starting FID of a set of contiguous frames
NNCF	H	Contains one less than the number of frames in the set.

Output Interface

R14	R	Points to zeroth byte of 1st frame
R15	R	Points to last byte of last frame

Element Usage

FRMN	D
FRMP	D
RECORD	D
NNCF	H
NPCF	H
R14	R
R15	R
D0	D

Subroutine Usage

One additional level of subroutine linkage required.

LINESUB

This routine returns the line number of the calling process in the accumulator

Input Interface

None

Output Interface

D0	D	Contains the line number associated with the process.

Element Usage

R14	R
D1	D

Subroutine Usage

GPCBO.

One additional level of subroutine linkage required.

LOGOFF

LOGOFF is a routine used to exit the TCL process and return to the logon process. It will abort any executes currently in progress, initialize all workspace, detach the tape, compute charges and send the process to logon.

LOGOFF changes the process status to 3 and then enters WRAPUP for the final processing. Process status is reflected by USER:

USER	Process Status
-1	spooler, not valid user, not logged on
1	ABS restore, coldstart tape, file restore
2	in process of logging off
3	OFF was entered, log off user
4	debugger
5	logged on, normal running
6	user is valid and in process of logging on

Input Interface

none.

Output Interface

RSCWA	T	Set = x'0184' (Initializes return stack)
INHIBITH	H	Set to one (can't break till logged on)
RTNFLG	B	Set to one
XMODE	T	Set to zero
USER	T	Set to 2 (logging off)

Subroutine Usage

Four additional levels of subroutine linkage required.

Exits

To MD99 to perform the final wrapup before being sent to logon.

MBDSUB - MBDNSUB - MBDSUBX

These routines convert a binary number to the equivalent string of decimal or hexadecimal characters. The number is specified in the 4-byte Accumulator for MBDSUB and MBDNSUB; the 6-byte accumulator for MBDSUBX.

MBDSUB and MBDSUBX return only as many characters as are needed to represent the number, while MBDNSUB always return a specified minimum number of characters (padded with leading zeroes or blanks when necessary). For a negative number, a minus sign precedes the numeric string

These subroutines are implicitly called by the instructions MBD and MBDN.

Input Interface

D0	D	The number to be converted; for MBDSUB, MBDNSUB.
FP0	F	The number to be converted; for MBDSUBX.
T4	T	The minimum string length. Leading zeroes or blanks are padded to ensure that the string is at least this length. The string may exceed this length if the value is high enough. For MBDNSUB.
BKBIT	B	Set if leading blanks are required as fill. Zero if zeroes are required as fill. For MBDNSUB
R15	R	Points one byte prior to the area where the converted string is to be stored. The area must be at least eighteen bytes in length for MBDSUBX. MBDSUB and MBDNSUB require at most eleven bytes.

Output Interface

BKBIT	B	= 0.
R15	R	Points to the last converted character.

MD200 - MD201

TCL-II Processor - These entry points (not subroutines) into the TCL-II Processor are used whenever a verb requires access to a file, either to all items or to explicitly specified items within a file.

MD200 is entered from the TCL-I processor after decoding the verb (primary mode ID=2).

MD201 is used by TCL-II itself to regain control from WRAPUP under certain conditions. TCL-II exits to the processor whose mode ID is specified in MODEID2. Typically, processors such as the EDITOR, ASSEMBLER, LOADER, etc., use TCL-II to feed them the set of items which was specified in the input data.

On entry, the TCL-II Processor checks the verb definition for a set of option characters in attribute 5. Verb options are the single characters, used in any sequence and combination, listed below (all other characters are ignored):

Option	Meaning
C	Copy - items retrieved are copied to the IS workspace.
E	Expand - items retrieved are expanded and copied to the IS work space; ignored if the "C" option is not present.
F	File access only - file parameters are set up but any item-list is ignored by TCL-II; if this option is present, any others are ignored.
N	New item acceptable - if the item specified is not on file, the secondary processor still gets control (the EDITOR, for example, can process a new item).
P	Print - on a full file retrieval (all items), the item-id of each item is printed as it is retrieved.
U	Updating sequence flagged - if items are to update as retrieved, this option is mandatory.
Z	Final entry required - the secondary processor will be entered once more after all items have been retrieved (the COPY processor, for instance, uses this option to print a message).

The input data string to TCL-II consists of the file name, followed by a list of items or an asterisk, which specifies all items in the file. The file name may be preceded by the modifier, DICT, which specifies access to the dictionary of the file. The item-list may be followed by a list of options, enclosed in parentheses, for the secondary processor. See GETOPT documentation for further information about options.

Input Interface

IR	R	Points to the AM before attribute 5 of the verb.
SR4	S	Points to the AM at the end of the verb.
MODEID2	T	Contains the mode ID of the processor to which TCL-II transfers control (assuming no error conditions are encountered).
BMSBEG	S	Points one prior to an area where the file name is to be copied, if the "F" option is present, otherwise one prior to an area where item-ids are to be copied.
ISBEG	S	Points one prior to an area where items are to be copied, if the "C" option is present.

Elements as required by GETFILE

Output Interface

DAF1	B	Set if the "U" option is specified.
DAF2	B	Set if the "C" option is specified.
DAF3	B	Set if the "P" option is specified.
DAF4	B	Set if the "N" option is specified.
DAF5	B	Set if the "Z" option is specified.
DAF6	B	Set if the "F" option is specified, or if a full file retrieval is specified (no "F" option).
DAF10	B	Set if more than one item is specified in the input data, but not a full file retrieval ("*").
DAF11	B	Set if the "E" option is specified.

NOTE: The above bits are not initialized to zero.

DAF8	B	Set if a file dictionary is being accessed, otherwise reset (from GETFILE).
DAF9	B	= 0.
IS	R	Points one past the end of the file name in the input string if the "F" option is present; points to the last AM in the copied item if the "C" option is present, otherwise to the end of the input string.
ISBEG	S	Unchanged.
BMSBEG	S	
RMBIT	B	Set if the file is successfully retrieved if the "F" option is present.
SBASE	D	Contain the base, modulo, and separation
SMOD	T	of the file being accessed.
SSEP	T	
BASE	D	= SBASE, SMOD, SSEP on the first exit
MODULO	T	only (from MD200).
SEPAR	T	
DBASE	D	Contain the base, modulo, and separation
DMOD	T	of the dictionary of the file being
DSEP	T	accessed if the "F" option is present.
SC0	C	Contains a SB if the last item-id in the input string is enclosed in quotes, otherwise is blank.

The following specifications are meaningful only when the "F" option is not present:

SR0	S	Points one prior to the count field of the retrieved item.
SIZE	T	Contains the value of the count field of the retrieved item.
SR4	S	Points to the last AM of the retrieved item.
ISEND	S	= IS if the "C" option is present.
IR	R	Points to the last AM of the retrieved item to be copied, if the "C" option is present, otherwise points to the AM following the item-id.
RMODE	T	MD201 if items are left to be processed, otherwise = 0.
XMODE	T	= 0.

Elements as set up by GETOPT if the input data contains an option string

Element Usage

C1	T	Used for error messages.

Elements used by the various subroutines below

Subroutine Usage

GETFILE; if no "F" option: GETOPT if the input data contains an option string, GETITM for full file retrieval, RETIX and one internal subroutine if not full file retrieval, GETSPC if more than one item (but not "*") specified, EXPAND if the "E" option is present, WRTLIN if the "P" option is present. MD201 only : WSINIT; GNTBLI if more than one item (but not "*") specified. MD995 and BMSOVF used with XMODE.

Seven additional levels of subroutine linkage required by MD200; five additional levels required by MD201 for full file retrieval, otherwise three levels required.

Errors

These conditions cause an exit to the WRAPUP processor:

Error	Condition
13	DL/ID item not found, or in bad format.
199	IS work space not big enough when the "C" option is specified.
200	No file name specified.
201	File name illegal or incoreectly defined in the M/DICT.
202	Item not on file; all messages of this type are stored until all items have been processed; items which are on file are still processed.
203	No item list specified.
209	The format of the option list is bad.

MD99 - MD992 - MD993 - MD994 - MD995 - MD999

WRAPUP Processor - These are the entry points into the system routine which wraps up the processing initiated by a TCL statement, performs disk updates and prints messages as required, and reinitializes functional elements for processing another TCL statement. WRAPUP may also be treated as a subroutine by setting tally RMODE to the mode ID of the routine to which WRAPUP should return control after it is done.

NOTE: The WRAPUP Processor always sets the return stack to a null or empty condition before exiting.

The various entry points are provided to simplify the interface requirements when WRAPUP is used to store or print messages from the ERRMSG file; the features of each can be seen in the following table:

MD99	Message numbers (without any parameters) may be stored in REJCTR, REJ0; and REJ1 (no action is taken if zero. If RMODE is zero, messages are printed regardless of the value of VOBIT. The messages are set up in the history string and control passes to MD999.
MD992	C1 contains a message number; D9 contains a numeric parameter. The value in C1 is converted to an ASCII string and used as the item-id to be retrieved from the ERRMSG file. This entry is commonly used for numeric parameters that might exceed 32767. The message is set up in the history string and control passes to MD99.
MD993	C1 contains a message number; C2 contains a numeric parameter; the value in C1, converted to an ASCII string, is used as the item-id of an item to be retrieved from the message file (normally ERRMSG). The message is set up in the history string, and control passes to MD99.
MD994	C1 contains a message number; IS points one before the beginning of a string parameter, which is terminated by an AM or SM; the message is set up in the history string and control passes to MD99.
MD995	Like MD994, except the string parameter is stored at BMSBEG+1 through an AM or SM.
MD999	The history string is processed, and process work spaces are reinitialized; control passes to TCL if RMODE is zero, otherwise to the routine specified by RMODE.

Input Interface

HSBEG	S	Points one byte before the beginning and
HSEND	S	the end, respectively, of the history string; if HSBEG=HSEND, the string is null.

Three types of history string elements are recognized by WRAPUP; all others are ignored. The type of processing done for each element depends on the second, and possibly third character of the element string. (The quote marks in the following examples are not part of the string.)

1. Output message

```
SM "O" AM message-id AM (parameter AM...) SM
```

Where "message-id" is the item-id (normally a decimal numeric) of an item in the message file

The parameter string is passed to PRTERR for message formatting (see PRTERR documentation)

2. Disk Update/Delete

```
SM "DU" AM base VM modulo VM separ AM item-id AM item-
body AM SM

SM "DD" AM base VM modulo VM separ AM item-id AM SM
```

Where "DU" causes the item in the file specified by "base", "modulo", and "separation" to be replace, and "DD" deletes it

3. (End of history string)

```
SM Z
```

Conventionally, a process wishing to add data to the history string begins at HSEND+1. After the additional elements are added, the string is terminated by a SM, a space and "Z". HSEND is set pointing to this SM.

WMODE	T	If non-zero, the value is used as the mode ID for an indirect subroutine call (BSLI *) executed immediately after the history string has been processed, and before work space and printer characteristics are reset; this allows special processing to be done on any entry into WRAPUP.
RMODE	T	If non-zero, WRAPUP exits to the specified mode ID instead of to TCL.
VOBIT	B	If set and RMODE is non-zero, messages are stored in the history string for output on a later entry into WRAPUP with RMODE zero.

REJCTR	T	May contain message numbers which do not require parameters. REJCTR is always tested first;
REJ0	T	then REJ0 is tested and used;
REJ1	T	and then REJ1 is tested and used; no action is taken on a zero value; a value of 9999 is used internally by WRAPUP to identify which messages have been processed, and should not be used as an input value for REJ0 or REJ1.
C1	T	(See MD993, MD994, and MD995 above).
C2	T	
LPBIT	B	If set, all open spool files are closed.
OVRFLCTR	D	If non-zero, used as the starting FID of a linked set of overflow frames released to the system overflow space pool. Used by SORT, for instance, to store the beginning FID of a sorted table, in which case overflow space used by SORT is always released, even if processing is aborted by an "END" command from DEBUG.
USER	T	Controls the final exit from WRAPUP when RMODE=0.

Output Interface

HSEND	S	= HSBEG except when messages are stored instead of printed.
VOBIT	B	
LPBIT	B	
WMODE	T	= 0.
REJCTR	T	
REJ0	T	
REJ1	T	
RMODE	T	
INHIBIT	B	

Return stack — Null: RSEND=X'01B0', RSCWA=X'0184', and the rest of the return stack is filled with X'FF'

Elements as initialized by WSINIT (and ISINIT if RMODE=0).

The following elements are set up only if RMODE=0:

XMODE	T	= 0.
OVRFLCTR	T	
IBSIZE	T	= 140.

Element Usage

UPD	R	
BASE	D	
MODULO	T	Used in disk updates.
SEPAR	T	
CH8	C	
R15	R	Used by NSPCQ.

Elements used by the subroutines below:

Subroutine Usage

WSINIT; MBDSUB for message numbers; PRTERR to print messages; CVTNIS and UPDITM to do disk updates; CRLFPRINT if a format error is found in a "DD" or "DU" history string element; PCLOSEALL if LPBIT=1; if RMODE=0: ISINIT, RESETTERM, RELSP (if USER=2), RELCHN (if OVRFLCTR is non-zero); UNLOCK, GLOCK, GUNLOCK.LINE, and TILD by NSPCQ.

Maximum of seven additional levels of subroutine linkage required if RELCHN must print an error message; maximum of six levels required for PRTERR. Four levels required for UPDITM. Three levels required for ISINIT. Two levels always needed for WSINIT.

Exits

To the entry point specified in RMODE if non-zero. To LOGOFF if USER=3 (set, for instance, by the DEBUG "OFF" command). To MD0 if USER=2 (set by the LOGOFF processor); otherwise to MD1.

Errors

If a format error is found in a "DD" or "DU" history string element, the message:

```
DISK-UPD STRING ERR
```

is displayed, and processing continues with the next element.

NEWPAGE

This routine is used to skip to a new page of the terminal or line printer and print a heading. No action is performed, however, if bit PAGINATE or tally PAGSIZE is zero.

Input Interface

As for WRTLIN, except OB is first set equal to OBBEG by this routine

Output Interface

Same as for WRTLIN.

Element Usage

Same as for WRTLIN.

Subroutine Usage

WRTLIN and routines called by it, if PAGINATE is set and PAGSIZE is greater than zero. Additional subroutine linkage required only if WRTLIN is called. See WRTLIN documentation for the nubmer of additional levels of linkage required, and add 1.

NEXTIR - NEXTOVF - BMSOVF

NEXTIR obtains the forward linked frame of the frame to which register IR currently points. If the forward link is zero, the routine attempts to obtain an available frame for the system overflow space pool and link it up appropriately (see ATTOVF documentation). In addition, if a frame is obtained, the IR register triad is set up before return, using routine RDREC.

NEXTOVF may be used in a special way to handle end-of-linked-frame conditions automatically when using register IR with single- or multiple-byte move or scan instructions (MIID, MII, or MCI). Tally XMODE should be set to the mode ID of NEXTOVF before the instruction is executed. If the instruction causes IR to reach an end-of-linked-frame condition (Forward Link Zero), the system will generate a subroutine call to NEXTOVF, which will attempt to obtain and link up an available frame, and then resume execution of the interrupted instruction (assuming a frame was obtained). If there are no more frames in the overflow space pool, NOSPACE is called. The "increment register by tally" instruction cannot be handled in this manner.

NEXTOVF is also used by UPDITM with register TS. If NEXTOVF is entered with TS at an end-of-linked-frames condition, a branch is taken to a point inside UPDITM. Under any other condition (other than IR or TS end-of-linked-frame), NEXTOVF immediately enters the DEBUGGER.

BMSOVF is used to handle end-of-linked-frame conditions automatically when using register BMS (R8).

Input Interface

IR	R	Points into the frame whose forward-linked frame is to be obtained (displacement unimportant).
ACF	H	For NEXTOVF only, must contain X'06' for IR end-of-linked-frame handling (set automatically by MIID, MII, and MCI instructions).
BMS	R	For BMSOVF only, points into the frame whose forward link is zero (displacement unimportant)

Output Interface

IR	R	Point to the first data byte of the forward linked frame.
IRBEG	S	= IR.
IREND	S	Points to the last byte of the forward linked frame.
RECORD	D	Contains the FID of the frame to which IR points.
R15	R	
NNCF	H	
FRMN	D	As set by RDLINK for the FID in RECORD.
FRMP	D	
NPCF	H	
OVRFLW	D	= RECORD if ATTOVF called, otherwise unchanged.

Element Usage

R14 R Used by RDLINK.
Elements used by ATTOVF if a frame is obtained from the overflow space pool.

Subroutine Usage

RDLINK; ATTOVF if a frame must be obtained from the overflow space pool; NOSPACE if ATTOVF cannot find any more frames.
Three additional levels of subroutine linkage required.

Exits

Normally returns via RDREC; possibly to NSPCQ if NOSPACE used; to 5,DB1 if ACF not X'06' or X'0D' (NEXTOVF only).

PCRLF - FFDLY

PCRLF prints a carriage return and line feed on the terminal and enters FFDLY, which prints a specified number of delay characters (X'00').

Input Interface

LFDLY	H	Contains the delay count (for PCRLF only).
T0	T	Contains the delay count (for FFDLY only).

Output Interface

None

Element Usage

R14	R

PINIT

PINIT is used for process initialization. Pointers are set up to all workspaces; links are set up in frames of linked workspaces (HS, IS, OS, and PROC). All elements in the primary, secondary, and tertiary (DEBUG) control blocks are zeroed, except as noted below.

Input Interface

R0	R	Points to the PCB of the process to be initialized.

Output Interface

R2	R	Points to the process's SCB (PCB+1).
HS	R	The beginning of the HS workspace (PCB+10).
HSBEG	S	= HS.
HSEND	S	
IS	R	Points to the beginning of the IS workspace (PCB+16).
ISBEG	S	= IS.
ISEND	S	
OS	R	Point to the beginning of the OS workspace (PCB+22).
OSBEG	S	= OS.
OSEND	S	
IBSIZE	T	= 140
OBSIZE	T	= 100
TTLY	T	= 0 (For DEBUG use).
INHIBIT	B	= 1.
PBUFBEG	S	set to beginning of PROC workspace (PCB+6)
PBUFEND	S	set to end of PROC workspace (PCB+9)

Other elements as initialized by WSINIT

Address registers, and the PCB elements PRMPC, SC0, SC1, and SC2 (all characters) are not zeroed. In addition, the tertiary control block is initialized for the debugger by setting the corresponding INDEBUG bit to 1, and setting the corresponding R1 and return stack elements to execute debugger code.

Subroutine Usage

WSINIT (local), LINK.
Three additional levels of subroutine linkage required.

PONOFF

Echo Toggle - PONOFF is used to toggle the bit LISTFLAG, which controls output to the terminal. If set to one, output is enabled.

Input Interface

IFLG	B	Set to one to force disable of output.
LFLG	B	Set to one to force enable of output.

Output Interface

LISTFLG	B	Will be toggled if IFLG & LFLG are zero

Element Usage

none.

PRINT - CRLFPRINT

PRINT and CRLFPRINT send messages to the terminal. A message is a block of text embedded in the calling program. CRLFPRINT precedes the text with a CR and LF (carriage return and line feed).

NOTE: These routines are not consistent with line printer and pagination conventions. Therefore, they should be used for short prompts and error messages. Also, they cannot be captured through the Basic EXECUTE statement.

The message must be a string that immediately follows the subroutine call. This string must also be terminated by one of the system delimeters. The different delimeters cause different terminating action to be taken.

Input Interface

Delimeter	Action Taken
SM	End of message. CR & LF printed. RTN executed.
AM	End of message. CR & LF printed. RTN executed.
VM	CR & LF printed and message processing continues.
SVM	End of message. RTN executed without printing CR & LF.

Message must follow BSL instruction.

Output Interface

none

Element Usage

R14	R
R15	R
D0	D

Example

```
BSL    PRINT
.
.
.
TEXT   C'HELLO',X'FF'    SM terminated message
```

PRIVTST1 - PRIVTST2

These routines check to see if the calling process has appropriate system privilege levels: PRIVTST1 checks for a minimum privilege level of "SYS1"; PRIVTST2 checks for a minimum privilege level of "SYS2".

If the process does not have the appropriate system privilege level, the following actions are taken:

bits, PQFLG and LISTFLAG, are set to zero,
tally, RMODE, is set to zero,
the History String is set to null (HSEND=HSBEG),
tally REJCTR is set to 82 (an error message number),
an exit is taken to MD99.

Otherwise, the routines run normally.

Entry	Bit tested (error if not set).
PRIVTST1	SYSPRIV1
PRIVTST2	SYSPRIV2

PROC User Exits

A user-written program can gain control during execution of a PROC by using the Uxxxx or Pxxxx command in the PROC where "xxxx" is the hexadecimal mode ID of the user routine. The routine can perform special processing, and then return control to the PROC processor. Necessarily, certain elements used by the PROC processor are maintained by the user program; these elements are marked with an asterisk in the table below.

Input Interface

*BASE	D	
*MODULO	T	
*SEPAR	T	
*PQBEG	S	Points one prior to the first PROC statement.
*PQEND	S	Points to the terminal AM of the PROC.
PQCUR	S	Points to the AM following the Uxxxx or
IR	R	Pxxxx statement.
*PBUFBEG	S	Points to the buffer containing the primary and secondary (if any) input buffers; buffer format is SB ... Primary input ... SM SB ... Secondary input ... SM.
*ISBEG	S	Points to the buffer containing the primary output line.
*STKBEG	S	Points to the buffer containing "stacked input" (secondary output).
IB	R	Is the current input buffer pointer (may point within either the primary or secondary input buffers).
*SR35	S	Points to the beginning of the current input buffer.
*SBIT	B	Set if a ST-ON command is in effect.
*ZBIT	B	Reset to identify the PROC processor in certain system subroutines.
*SC2	C	Blank.

		SBIT on	SBIT off
IS	R	Points to the last byte moved into the secondary output buffer	Points to the last byte moved into the primary output buffer
UPD	R	Points to the last byte moved into the secondary output buffer	Points to the last byte moved into the primary output buffer

Output Interface

IR	R	Points to the AM preceding the next PROC statement to be executed. May be altered to change PROC execution.
IS	R	May be altered as needed to alter data
UPD	R	within the input and output buffers, but
IB	R	the formats described above must be maintained

Exits

The normal method of returning control to the PROC processor is to execute an external branch instruction (ENT) to PQXIT. To return control and also reset the buffers to an empty condition, entry PQLINK may be used. If it is necessary to abort PROC control and exit to WRAPUP, bit PQFLG should be reset before branching to any of the WRAPUP entry points.

When a PROC eventually transfers control to TCL (via the "P" operator), certain elements are expected to be in an initial condition. Therefore, unless the elements are deliberately set up as a means of passing parameters to other processors, these elements should be reset before returning to the PROC. Specifically, the bits ABIT through ZBIT are expected to be zero in the TCL-II and ACCESS processors. It is best to avoid usage of these bits in PROC user exits. Also, the scan character registers SC0, SC1, and SC2 must contain an SB, a blank, and a blank, respectively.

PRTERR

PRTERR is used to retrieve and print a message from the system file ERRMSG. If EBASE is zero, PRTERR attempts to set EBASE, EMOD, and ESEP to the parameters for the system file ERRMSG, and exits abnormally if unable to do so. A parameter string may be passed to the routine, in which case the parameters are formatted and inserted according to the codes in the message item. Items in the ERRMSG file consist of an arbitrary number of lines where a line is delimited by an AM, with each line containing a code letter in column one, possibly followed by a string or numeric parameter (numeric parameters enclosed in parentheses). The possible codes and their meanings are listed below. Braces indicate optional parameters; the parenthesis are required; no blank is necessary between the code letter and the parameter string.

Code	Definition
A {(*n*)}	Insert parameter - The next parameter from the parameter string, if any, is placed into the output buffer; if *n* is specified, the parameter is left-justified in a blank field of length *n*.
D	Insert date - The system date in DD MMM YYYY format is added to the output buffer.
E [string]	Insert item.id - The message item-id, surrounded by brackets, is placed into the output.
H {string}	Insert string - The character string is moved into the output buffer.
L {(*n*)}	Linefeed - The output buffer is printed, and *n* line feeds are output; (one if *n* is not specified).
R {(*n*)}	Insert parameter - The next parameter from the parameter string, if any, is moved to the current position in the output buffer. If an *n* is specified, the parameter is right-justified in a blank field of length *n*.
S {(*n*)}	Set pointer - The pointer to the current position in the output buffer is repositioned to the specified column; column one if *n* is not present.
T	Insert time - The system time in HH:MM:SS is added to the output buffer.
X (*n*)	Skip parameter - The pointer to the current position in the output buffer is incremented by *n* spaces; if the end of a line is reached (see below), the buffer is printed and a new line is started.

Input Interface

TS	R	Points one prior to the message item-id, which must be terminated by an AM; parameters optionally follow, being delimited by AM's; the parameter string must end with a SM.
EBASE	D	ERRMSG file.
EMOD	T	
ESEP	T	
MBASE	D	Master Dictionary. Exit abnormally if zero.
MMOD	T	
MSEP	T	
OBSIZE	T	Contains the maximum number of characters to be output on a line (normally set at logon time).

Other elements as required by WRTLIN.

Output Interface

TS	R	Points to the AM after the message item-id if no parameters are processed.
EBASE	D	
EMOD	T	
ESEP	T	
LINCTR	T	Updated if bit PAGINATE is set.
PAGNUM	T	

Element Usage

SB60	B	
SB61	B	
CTR0	T	
T6	T	
BASE	D	
MODULO	T	
SEPAR	T	
AF	R	
IR	R	
BMS	R	
BMSBEG	S	
OB	R	
R14	R	
SR4	S	
CTR1	T	Used with "R" code messages .
SYSR1	S	Used with "S" code messages.
INHIBIT	B	Set during retrieval of file ERRMSG, if EBASE is originally zero, and reset afterwards to the value on entry .

All elements used by WRTLIN (unless PRTERR exits abnormally), and elements used by GBMS if PRTERR attempts retrieval of the system file ERRMSG.

Subroutine Usage

RETIX, WRTLIN, DATE (for "D" code messages), TIME (for "T" code messages), GBMS (for retrieving ERRMSG); Six additional levels of subroutine linkage required if GBMS attempts retrieval of an ERRMSG file which is a "Q" code item, otherwise four levels required.

Exits

To 2,ABSL if EBASE and MBASE are both zero.

RDLABEL - RDLABEL1

RDLABEL is used to read a standard 80 byte label on the tape.

RDLABEL1 is used to be sure only reel one is inserted.

Label Format

```
_L.XXXX.HH:MM:SS..DD/MMM/YYYY.{47 chars for name}^RR_
```

where

_	is a segment mark
.	is a blank
^	is an attribute mark
XXXX	is the 4-character block size in ASCII hex
HH:MM:SS	is the current system time
DD/MMM/YYYY	is the current system date
RR	is the current 2-character reel number

Input Interface

none

Output Interface

UNLABEL	B	Set to one if if data on tape is not a label. Otherwise it is cleared.
TPRECL	T	If UNLABEL is zero then it holds the value of the block size specified in the label read.

Element Usage

R14	R
R15	R
T3	T
Accumulator	

Subroutine Usage

TPSTAT is called by this subroutine. Seven additional levels of subroutine linkage may be required.

RDLINK - RDREC

RDLINK is used to read the links of the FID number stored in RECORD. RDREC also sets up IR to RECORD.

Input Interface

RECORD	T	Contains the FID to read its links

Output Interface

RECORD	D	unchanged
IR	R	Points to byte zero of frame in RECORD (RDREC only)
R15	R	Points to byte one of frame in RECORD
NNCF	H	set to link info
NPCF	H	set to link info
FRMN	H	set to link info
FRMP	H	set to link info

Element Usage

R15	R
D0	D

Subroutine Usage

none

READLIN - READLINX - READIB

READLIN, READLINX and READIB are the standard terminal input routines. IBBEG points to the area where the input characters will be stored. Characters will be input until a CR or LF is encountered or the number of characters equals the count in IBSIZE. The CR or LF is overwritten with a SM and IBEND will point to this segment mark upon return.

READLIN and READIB check first for stacked input and process tab characters as padded blanks but READLINX does not.

READLIN and READLINX echo a trailing CR and LF to the terminal when the CR or LF is entered to terminate the input sequence but READIB does not.

Editing	Action
CTRL-H	Backspace function
CTRL-W	Backspace word function
CTRL-X	Cancel line function

Input Interface

IBBEG	S	Start of buffer.
IBSIZE	T	Maximum number of characters to accept.
STKFLG	B	Set if stack exists, stacked input available. Zero if to get input from terminal.
CCDEL	B	Set to delete control characters entered.
ITABFLG	B	Set to process tab characters using input tab table in QCB.
PRMPC	C	The prompt character.

Output Interface

IB	R	= IBBEG.
IBEND	S	SM terminating data.
STKFLG	B	Zeroed if the end of stacked data has been reached
STKINP	S	Points the next available line of stacked data

Subroutine Usage

Two additional levels of subroutine linkage required

RELBLK - RELCHN - RELOVF

These routines are used to release frames to overflow. RELOVF is used to release a single frame, RELBLK is used to release a block of contiguous frames, and RELCHN is used to release a chain of linked frames (which may or may not be contiguous). A call to RELCHN specifies the FID of a linked set of frames; the routine will release all frames in the chain until a zero forward link is encountered.

Input Interface

OVRFLW	D	Contains the FID of the frame to be released (for RELOVF), or the first FID of the block or chain to be relased (for REBLK and RELCHN, respectively).
D0	D	Contains the number of frames (block size) to be released; for RELBLK only.

Output Interface

None

Element Usage

OVRFLW	D	
R14	R	Utility.
R15	R	
D0	D	Accumulator.
D1	D	Used by SYSREL.
D2	D	

Subroutine Usage

SYSREL; two internal subroutines.
Two additional levels of subroutine linkage required.

RETI - RETIX - RETIXU - RETIXX

These are the entry points to the standard system routine for retrieving an item from a file. The item-ID is explicitly specified to the routine, as are the base, modulo, and separation file parameters.

If the entry RETIXX is used, the number of the first frame in the group in which the item may be stored must be specified additionally.

The other entries perform a hashing algorithm to determine the group. The group is searched sequentially for a matching item-ID. If the routine finds a match, it returns pointers to the beginning and end of the item, and the item size (from the item count field).

If entry RETIXU is used, the group is locked during processing, preventing other programs from accessing (and possibly changing) the data. The item-ID is specified in a buffer defined by register BMSBEG.

If entry RETI is used, register BMS must point to the last byte of the item-ID, and an AM will be appended by the routine.

For all other entry points, the item-ID must already be terminated by an AM.

Input Interface

BMSBEG	S	Points one byte before the item-ID.
BMS	R	Points to the last character of the item-ID, for RETI, RETIXX, and RETIX only.
BASE	D	
MODULO	T	
SEPAR	T	
RECORD	D	Contains the beginning FID of the group to be searched; for RETIXX only.

Output Interface

BMS	R	Point to the last character of the item-ID.
BMSEND	S	= BMS.
RECORD	D	Contains the beginning FID of the group to which the item-ID hashes (set if HASH is called).
NNCF	H	
FRMN	D	Contain the link fields of the frame
FRMP	D	specified in RECORD; set by RDREC.
NPCF	H	
XMODE	T	= 0.

		Item Found	Item Not Found
RMBIT	B	= 1	= 0
SIZE	T	= value of item count field	= 0
R14	R	Points one byte prior to the item count field	Points to the last AM of the last item in the group
IR	R	Points to the first AM of the item	Points to the AM indicating end of group data (=R14+1)
SR4	S	Points to the last AM of the item	=R14

Element Usage

D0	D	Accumulator.
D1	D	Accumulator.
R15	R	

Subroutine Usage

RDREC (local), HASH (except for RETIXX; local), GLOCK (RETIXU only), IROVF (for IR overflow space handling and error conditions); Three additional levels of subroutine linkage required (for IROVF and GLOCK, RDREC and HASH require one level)

Exits

If the data in the group is bad - premature end of linked frames, or non-hexadecimal character encountered in the count field - the message.

```
GROUP FORMAT ERROR  xxxxxx
```

is returned (where *xxxxxx* is the FID indicating where the error was found), and the routine returns with an "item not found" condition. Data is not destroyed, and the group format error will remain.

RMODE

RMODE is a tally used to store the mode ID of a routine which used by WRAPUP to exit to instead of going to TCL. If RMODE is non-zero, the return stack is cleared (RSCWA is set = X'0184') and register triads AF, BMS, CS, TS, IB and OB are initialized. WRAPUP will not print any messages until the final entry to WRAPUP (RMODE is zero). In this way, WRAPUP can be used as a subroutine. The Selection Processor uses this technique. WRAPUP is called after each item is processed and RMODE returns control to the Selection Processor.

SETLPTR - SETTERM

These routines are used to set output characteristics such as line width, page depth, etc., to the previously-specified values for either the terminal or the line printer. In addition, the current line number and page number are saved so that when switching from terminal to line printer and back, pagination will continue automatically from the previous values.

Input Interface

LPBIT	B	Reset by SETTERM; set by SETLPTR.
LINCTR	T	Current line number.
PAGNUM	T	Current page number.
OBSIZE	T	Size of the OB buffer.
TPAGSIZE	T	Number of printable lines or per page for the terminal or line.
PPAGSIZE	T	printer.
TOBSIZE	T	Size of the output (OB) or buffer for the terminal or line printer.
POBSIZE	T	
TLINCTR	T	Contains the current line number for the or terminal or line printer.
PLINCTR	T	
TPAGNUM	T	Contains the current page number for the or terminal or line printer.
PPAGNUM	T	

NOTE: TPAGSIZE, TOBSIZE, TLINCTR, and TPAGNUM are required only by SETTERM; PPAGSIZE, POBSIZE, PLINCTR, and PPAGNUM are required only by SETLPTR.

Output Interface

PAGSIZE	T	
OBSIZE	T	Set to the appropriate characteristics
LINCTR	T	for terminal or line printer output.
PAGNUM	T	
TLINCTR	T	= LINCTR; TLINCTR set by SETLPTR, PLINCTR or set by SETTERM.
PLINCTR	T	
OBSIZE	T	= 79 if originally zero.
R14	R	= OBBEG + OBSIZE.
OBEND	S	The area from the address pointed to by OBBEG to that pointed to by OBEND is filled with blanks

SETUPTERM

Set the default value for terminal and line printer characteristics (as used by INITTERM).

Input Interface

BSPCH	C	The character to be echoed for a backspace.
LFDLY	T	The number of fill characters to be output after a CR/LF in the lower byte; if the upper byte is greater than one, a form feed is output before each page of paginated output, and that number of fill characters is output.
TOBSIZE	T	The terminal line width.
TPAGSIZE	T	The terminal page depth.
POBSIZE	T	The printer line width.
PPAGSIZE	T	The printer page depth.
PAGSKIP	T	The number of lines to be skipped at the bottom of each page.

Output Interface

Default values initialized as described

SLEEP

This routine causes the calling process to go into an inactive state for a specified amount of time. Either the amount of time to sleep or the time at which to wake up may be specified.

Input Interface

D0	D	Contains the number of seconds to sleep, up to 86400 (one day), or the time to wake up (number of seconds past midnight) if RMBIT is reset.
RMBIT	B	Set if D0 contains the number of seconds to sleep, and reset if it contains the time to wake up.

Output Interface

None

Element Usage

T2	T	Used on a monitor call
D2	D	to get system time.

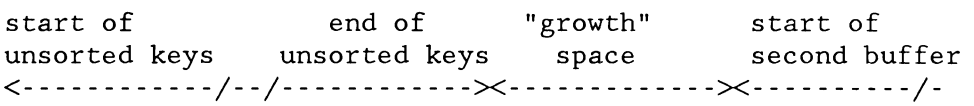
Subroutine Usage

SLEEP uses SLEEPSUB; One addition level of subroutine linkage required by SLEEPSUB.

SORT

This routine sorts an arbitrarily long string of keys in ascending sequence only; the calling program must complement the keys if a descending sort is required. The keys are separated by SM's when presented to SORT, they are returned separated by SB's. Any character, including system delimiters other than the SM and SB may be present within the keys.

The sort algorithm used is an n-way polyphase sort-merge. The original unsorted key string may grow by a factor of 10%. A separate buffer is required for the sorted key string, which is about the same length as the unsorted key string. The growth space is contiguous to the end of the original key string; the second buffer may be specified anywhere. SORT automatically obtains and links overflow space whenever needed. Due to this, one can follow standard system convention and build the entire unsorted string in an overflow table with OVRFLW containing the beginning FID the setup is:



The second buffer pointer then is merely set at the end of the growth space, and SORT is allowed to obtain additional space as required. Alternately, the entire set of buffers may be in the IS or OS workspace if they are large enough.

Input Interface

SR1	S	Points to the SM preceding the first key.
SR2	S	Points to the SM terminating the last key.
SR3	S	Points to the beginning of the second buffer.

Output Interface

SR1	S	Points before the SB preceding the first sorted key (the exact offset varies from case to case). The end of the sorted keys (separated by SB's) is marked by a SM.

Element Usage

HBIT	B
LBIT	B
SB1	B
SC2	C
XMODE	T
D0	D
IS	R
OS	R
BMS	R
TS	R
CS	R
R14	R
R15	R
S1	S
S2	S
S3	S
S5	S
S7	S
S8	S
S9	S

Subroutine Usage

COMP; GWS used with XMODE.
Four additional levels of subroutine linkage required.

SYSTEM.CURSOR

SYSTEM.CURSOR is used to generate the escape sequences needed for screen format commands commonly used from BASIC with PRINT @()

By using positive numbers in T0, x,y coordinate positioning is achieved.

By using negative numbers according to the chart below screen functions are generated.

Code	Function Action
-1	clear screen and place cursor at home position.
-2	move cursor to home position.
-3	clear to end of screen from current position.
-4	clear to end of line from current position.
-5	enable blink function.
-6	disable blink function.
-7	enable protect function.
-8	disable protect function.
-9	backspace function.
-10	move cursor up one line.
-11	enable protect mode.
-12	disable protect mode.
-13	reverse video on.
-14	reverse video off.
-15	underline on.
-16	underline off.
-17	slave on.
-18	slave off.
-19	cursor right.
-20	cursor left.
-21	graphics (alt char set) on.
-22	graphics (alt char set) off.
-23	keyboard lock.
-24	keyboard unlock.
-25	control char enable.
-26	control char disable.
-27	write status line.
-28	erase status line.
-29	initialize terminal modes.
-30	download function keys.
-31	non-embedded standout on.
-32	non-embedded standout off.
-99	embedded visual attributes.
-100	half intensity.
-101	full intensity.

Input Interface

R15	R	Points one before location to store the escape sequence
T0	T	Holds the code for the function: T0 < 0 gets system function. T0 >= 0 and CTR10 > -2 gets x,y positioning. T0 >= 0 and CTR10 < -1 gets user defined function
CTR10	T	Y coordinate (row) for x,y positioning. May also contain a parameter to pass to a user defined function
CTR11	T	X coordinate (column) for x,y positioning.

Output Interface

R15	R	Points on the last character in the escape sequence

Element Usage

R14	R
R15	R
CTR10	T
CTR11	T
CTR40	T
T0	T
T2	T
T4	T

Example

Putting an x at 15,15.

```
LOAD   15           T0 = positive--x,y coordinates
STORE  CTR10        copy T0 to y coordinate storage
STORE  CTR11        copy T0 to x coordinate storage
MOV    OB,R15       set up pointer in output buffer
BSL    SYSTEM.CURSOR
MOV    R15,OB
MCI    C'x',OB      put x after escape sequence
BSL    WRTLIN       write it to the terminal
```

TATT

TATT is used to attach the tape device to the current user. If unsuccessful then bit ATTACH is cleared.

Input Interface

TPRECL	T	Contains the block size desired. If not in a valid range then the upper tally of D4 is used instead.

Output Interface

ATTACH	B	Set if the tape was successfully attached otherwise is is cleared.

Element Usage

R14	R
R15	R
D4	D
D0	D

TDET

TDET is used to detach the tape from the current process. The QCB is unlocked for other uses.

Input Interface

none

Output Interface

ATTACH	B	Cleared

TIME - DATE - TIMDATE

These routines return the system time and/or the system date, and store it in the buffer area specified by register R15. The time is returned as on a 24-hour clock.

Entry	Buffer size	Format required (bytes)
TIME	9	HH:MM:SS
DATE	12	DD MM YYYY
TIMDATE	22	HH:MM:SS DD MMM YYYY

Input Interface

R15	R	Points one prior to the buffer area.

Output Interface

R15	R	Points to the last byte of the data stored; the byte immediately following contains a blank.
R14FID	D	= 0 (DATE and TIMDATE only).

Element Usage

D0	D	Accumulator.
D1	D	Used by TIME and TIMDATE only.
D2	D	
D3	D	

Subroutine Usage

TIME used by TIMDATE; MBDSUB used by TIME.

Two additional levels of subroutine linkage required by TIMDATE, one level required by TIME, none by DATE

TPINIT

TPINIT is used to initialize the tape control block (QCB), it's buffers and the tape status word TAPSTW.

The bits defined in the Tape Status word are:

PROTECT
EOTBIT
EOFBIT
TPRDY
PARITY

Input Interface

ATTACH	B	Must be set to one by TATT subroutine

Output Interface

PROTECT	B	Set if write protected
EOFBIT	B	Set if end of file
EOTBIT	B	Set if end of tape, zero if beginning of tape
TPRDY	B	Set to one if no errors occur
PARITY	B	Set on a parity error
R13	R	Used to point to the tape control block (QCB)

Element Usage

R14	R
R15	R
D0	D

Subroutine Usage

Two additional levels of subroutine linkage required.

Example

```
BSL   TATT
BBZ   ATTACH,XIT    attached to another user?
BSL   TPINIT
  .
  .
  .
```

TPREAD - TPWRITE

TPREAD reads a specified number of bytes from the tape into a buffer pointed to by R15 at entry to the routine.

TPWRITE writes a specified number of bytes from the buffer pointed to by R15.

Both TPREAD and TPWRITE use a virtual tape drive with common routines. The initial execution of either entry point causes initialization of two buffers of a size sufficient to contain TPRECL, assigned during execution of the T-ATT verb or obtained by the RDLABEL routine from the tape record size in the standard R83 tape label. These buffers are released during WRAPUP processing after RMODE and WMODE complete. The process returns to TCL or the CHAIN or PROC analogs to TCL.

At all times after initialization, R7 is live and points to the current address or write location in the tape buffers. R7 must be saved and reserved if used between reads or writes. The contents of the accumulator is the number of characters to transfer to or from the tape buffer. The alignment of R7 in the buffer and the relative size of TPRECL and D0 do not need to be considered.

If D0 is zero on a read, then TPREAD will return to the calling routine with R7 pointing one before the next string to be read, XMODE will be set to the tape handler routine TPRXMODE or TPWXMODE, and the old XMODE, if any, will be in YMODE. This allows transparant tape reading using MIID or MIIT R7, XX. A forward link zero fault on R7 will cause the next tape record to be read into the last buffer, R7 to be reset to the beginning of the current buffer; and execution then continues in the MII instruction. The user is responsible for handling an end-of-file condition when reading the tape. When this occurs, the EOFBIT will be set.

If D0 is zero on a write, then TPWRITE will fill the rest of the tape buffer with the character pointed to by R15, which will cause the buffer to be written to tape. This is recommended in order to send the last partial tape record to the tape, after which WEOF should be executed.

Input Interface

ATTACH	B	Must be set. Use T-ATT verb.
TPRECL	T	As above.
R15	R	Points to one byte before the source or destination buffer start location.
R7	R	Must be the same at the beginning of the next tape operation as it was at the end of the last tape operation. Initialized by TPREAD TPWRITE on first-time call.
D0	D	Counts the number of bytes to be transferred to or from the tape buffers.

Output Interface

R15	R	Points at the end of the source or destination buffer if D0 was non-zero; unchanged if D0 was zero.
D0	D	= 0.
EOFBIT	B	Indicates end-of-file on read if set.
EOTBIT	B	Indicates end-of-tape if set; the tape handler will rewind the tape and tell the operator to mount the next tape. This may be executed in the middle of an MII instruction, as above, which will then continue to execute when the new reel is mounted and the label handled.

Element Usage

The tape handler will stack and restore most of the elements which it uses. However, the following elements are modified:

T5	T	
T6	T	
T7	T	
YMODE	T	For any current XMODE.
D2	D	
R2;H0	H	Store a flag.
R4	R	Used as a pointer to the text block in the write-label routine.
R7	R	The tape buffer pointer.
R14	R	
R15	R	

Subroutine Usage

TPREAD and TPWRITE use an extensive set of internal subroutines in such a way that element usage is transparent outside of the above set. Both may go to seven levels of subroutine usage if either encounters a parity error while handling a label on the second and following tape reels.

Control is transferred to the PRINT, CRLFPRINT and PCRLF routines for attention by the operator in a manner transparant to the calling routine. They include no write ring, parity error after ten retries, tape not ready, and block transfer incomplete messages and recovery alternatives.

TPSTAT

TPSTAT is used to check the status of the tape drive until ready. If the drive is not ready (TPRDY = 0), the status bits are retrieved in a loop until the tape becomes ready (TPRDY = 1) or a time out occurs. If a time out occurs the following message will be displayed:

```
TAPE NOT READY
CONTINUE OR QUIT (C/Q)
```

This subroutine is useful because it will not return until the tape is ready to process its next operation. Setting TPRDY to zero and calling this subroutine, returns control only when the previous operation has completed.

Input Interface

none

Output Interface

TPRDY	B	Always one upon exit
EOFBIT	B	Updated if TPRDY is zero upon entry
EOTBIT	B	Updated if TPRDY is zero upon entry
PARITY	B	Updated if TPRDY is zero upon entry
PROTECT	B	Updated if TPRDY is zero upon entry

Element Usage

R14	R
H0	D

Subroutine Usage

One additional levels of subroutine linkage required.

Exits

Aborts during looping phase on a time out.

TSINIT

This routine initializes the register triad associated with the TS workspace.

Input Interface

None

Output Interface

TS	R	Point to the beginning of the TS workspace (PCB+5).
TSBEG	S	= TS.
TSEND	S	Point to the last byte of the TS workspace (511 bytes past TSBEG); this is an unlinked workspace. The first byte of the workspace is set to X'00'.

Subroutine Usage

One internal subroutine.
One additional level of subroutine linkage required.

UPDITM

UPDITM perform updates to a file defined by its base FID, modulo, and separation. If the item is to be deleted, these routines compress the remainder of the data in the group in which the item resides. If the item is to be added, it is added at the end of the current data in the group. If the item is to be replaced, it is replaced in place, sliding the remaining items in the group to the left or right as necessary.

If the update causes the data in the group to reach the end of the linked frames, NEXTOVF is entered to obtain another frame from the overflow space pool and link it to the previous linked set; as many as required are added. If the deletion or replacement of an item causes an empty frame at the end of the linked frame set, and that frame is not in the primary area of the group, it is released to the overflow space pool.

UPDITM uses RETIXU to retrieve the item to be updated, locking the group. Once the item is retrieved, processing cannot be interrupted until completed.

Input Interface

BMSBEG	S	Points one prior to the item-id of the item to be updated; the item-id must be terminated by an AM.
TS	R	Points one prior to the item body to be added or replaced (no item-id or count field); not needed for deletions; the item body must be terminated by a SM.
CH8	C	Contains the character 'D' for item deletion; 'U' for item addition or replacement.
BASE	D	Contain the base, modulo, and separation
MODULO	T	of the file being updated.
SEPAR	T	

Output Interface

Remainder of the last frame in the group filled with blanks

Element Usage

D3	D
D4	D
NNCF	H
FRMN	D
FRMP	D
NPCF	H

Elements used by the various subroutines below.

Subroutine Usage

RDREC; HASH, GLOCK, and RETIXU RELCHN if overflow frames returned; WTLINK if data ends in the last frame of "prime" space, or in overflow space; COPYALL if the item is on file; BKUPD; GUNLOCK; NEXTOVF, BMSOVF, and IROVF used with XMODE.
Four additional levels of subroutine linkage required by UPDITM.

Errors

If the group data is bad (premature end of linked, or non-hexadecimal character found in an item count field), IROVF is entered to print a warning message, and the group data is terminated at the end of the last good item before processing continues.

WEOF

WEOF is used to write end of file mark on a tape that has been previously attached.

Input Interface

ATTACH	B	Must be set by TATT

Output Interface

EOFBIT	Set to one

Element Usage

R14	R
R15	R
D0	D

Example

```
        BSL   TATT
        BBZ   ATTACH,XIT  attached to another user?
        BSL   TPINIT
        BSL   WEOF
```

WMODE

WMODE is a tally which is used to store a mode ID of a routine which will be called as a subroutine from WRAPUP. Right before WRAPUP initializes a process' workspace it checks WMODE. If this tally is non-zero then a BSL* WMODE instruction is executed. This mechanism is useful to do extra or special final processing not normally done by the standard WRAPUP routines.

WRTLIN - WRITOB

These routines output data to the terminal or line printer. WRTLIN deletes trailing blanks from the data and enters WT2. WT2 adds a trailing carriage return and line feed, increments LINCTR, and enters WRITOB, which outputs the data.

The data to be output begins at the address referenced by OBBEG and continues through the address pointed to by OB. Output is routed to the terminal if bit LPBIT is off, otherwise it is stored in the printer spooling area. Pagination and pageheading routines are invoked automatically, if bit PAGINATE is set. If it is set, when the number of lines output in the current page (in LINCTR) exceeds the page size (in PAGSIZE), the following actions take place: 1) the number of lines in PAGSKIP are skipped; 2) the page number in PAGNUM is incremented; and, 3) a new heading is printed. A value of zero in PAGSIZE suppresses pagination regardless of the PAGINATE setting.

Input Interface

OBBEG	S	Points one byte prior to the output data buffer.
OB	R	Points to the last character; the buffer must extend at least one character beyond this location.
LPBIT	B	If set, output is routed to the spooler. The routine SETLPTR should be used to set this bit so printer characters are set up correctly.
LISTFLAG	B	If set, all output to the terminal is suppressed.
NOBLNK	B	If set, blanking of the output buffer is suppressed.
LFDLY	T	Lower byte contains the number of fill characters to be output after a CR/LF.
PAGINATE	B	If set, pagination and page-headings are invoked.
PFILE	T	Contains the print file number for PPUT; meaningful only if LPBIT is set.

The following specifications are meaningful only if PAGINATE is set:

PAGHEAD	S	Points one byte before the beginning of the page-heading message; if the frame field of this register is zero, no heading is printed.
PAGHEAD	S	Points to the location of the page-heading message.
PAGSIZE	T	Contains the number of printable lines per page.
PAGSKIP	T	Contains the number of lines to be skipped at the bottom of each page.
PAGNUM	T	Contains the current page number.
PAGFRMT	B	If set, the process pauses at the end of each page of output until some terminal input (even just a carriage return) is entered.
LFDLY	T	If the upper byte is greater than one, and output is to the terminal, a form-feed (X'0C') is output at the top of each page, and the number n the upper byte is used as the number of fill characters output after the form feed.

Output Interface

OB	R	= OBBEG.

The following specifications are meaningful only if PAGINATE is set:

LINCTR	T	Reset appropriately.
PAGNUM	T	
T7	T	Contains the original value of PAGNUM.

Element Usage

R14	R	
R15	R	Scratch
SYSR1	S	
R8	R	
RECORD	T	Used by PPUT (when LPBIT is set).
OVRFLW	T	
SYSR2	S	Used if PAGINATE is set and the header message contains a VM.
T4	T	
T5	T	Used if PAGINATE is set and the header message contains an SVM.
D2	D	
D3	D	

All elements used by ATTOVF (called by PPUT if more disk space needed).

Subroutine Usage

FFDLY, PPUT (if LPBIT set0, WT2 (if PAGINATE set and the header message contains a VM), TIMDATE (if PAGINATE set and the header contains a SVM), DATE (if PAGINATE set and the header message contains two SVM's in succession).

Four additional levels of subroutine linkage required if LPBIT is set; three levels required for TIMDATE; one level always required for LFDLY.

WSINIT

This routine initializes the process workspaces shown below. In each case, the "beginning" storage register (and associated address register, if present) is set pointing to the first byte of the workspace. The "ending" storage register is set pointing to the last data byte. All workspaces, except TS and the PROC workspace (called PBUF), are contained in one frame; PBUFBEG and PBUFEND define a 4-frame linked workspace.

Workspace Triad	Size (Bytes)
BMS, BMSBEG, BMSEND	50
CS, CSBEG, CSEND	50
AF, AFBEG, AFEND	100
TS, TSBEG, TSEND	Contents of IBSIZE; max. 140
IB, IBBEG, IBEND	Contents of OBSIZE; max. 140
OB, OBBEG, OBEND	511
PBUF, PBUFBEG, PBUFEND	2000 (4 linked frames)

Input Interface

IBSIZE	T	Size of IB buffer.
OBSIZE	T	Size of OB buffer.

Output Interface

Registers are set up as described above. The first byte of each workspace, except the OB, is set to x'00'. The OB workspace is filled with blanks (x'20'). IBSIZE and OBSIZE are set to 140 if initially greater.

Element Usage

R14	R
R15	R

Subroutine Usage

TSINIT (local), and one internal subroutine.
Two additional levels of subroutine linkage required.

WTLABEL - WTLABEL1

WTLABEL is used to write a standard 80 byte label on the tape. WTLABEL1 is used to be sure reel one is current reel number.

Label Format

```
_L.XXXX.HH:MM:SS..DD/MMM/YYYY.{47 chars for name}^RR_
```

where

^	is an attribute mark
.	is a blank
_	is a segment mark
XXXX	is the 4-character block size in ASCII hex
HH:MM:SS	is the current system time
DD/MMM/YYYY	is the current system date
RR	is the current 2-character reel number

Input Interface

IS	R	Points one before the label name, terminated by a SM

Output Interface

UNLABEL	B	Set to one if IS points to a null string otherwise cleared. If IS points one before a SM then it is considered a null string.

Element Usage

R14	R
R15	R
T3	T
Accumulator	

Subroutine Usage

TPSTAT is called by this subroutine. Seven additional levels of subroutine linkage may be required.

XISOS

XISOS exchanges the contents of the register triad IS with register triad OS.

Input Interface

none

Output Interface

IS	R	= OS
ISBEG	S	= OSBEG
ISEND	S	= OSEND
OS	R	= IS
OSBEG	S	= ISBEG
OSEND	S	= ISEND

Element Usage

Accumulator

XMODE

XMODE is a tally where a mode ID is stored. This mode ID is used to pass control when a Forward Link Zero condition occurs. If an instruction reaches an end-of-linked-frame condition, the debugger is entered. The offending register number is stored in ACF. D0 is copied into D1. If XMODE is zero, the instruction aborts with a "Forward Link Zero" message. If XMODE is non-zero, control passes through ENT* XMODE to the specified mode ID. This is known as an 'XMODE trap'. An XMODE trap will occur when using character string instructions.

The most common use for XMODE trapping is to attach a frame to the end of the set of linked frames and continue executing the instruction that caused the trap. This involves setting up the register that caused the trap (stored in ACF) to the last byte executed and RTNing to the instruction that caused the trap. Usually the last byte executed successfully is the last byte in the frame.

There are standard system subroutines that use this mechanism. If using a string instruction with IR or BMS then have XMODE set to IROVF or BMSOVF and overflow attachment will be transparent.

When writing XMODE trap routines, check ACF for the proper register. The forward link zero could occur on any register, especially during debugging a piece of code. On MIIT type instructions that use the accumulator, since D0 is used by the debugger to transfer control, you must copy D1 back to D0.

Example

Handling a possible Forward Link Zero.

```
FRM.SIZE  DEFN  511
          MOV   X.TRAP,XMODE     Able to trap FLZ aborts
          MIID  R15,HS,X'E0'     Copy data
          ZERO  XMODE            Disable trap mechanism
            .
            .
            .
X.TRAP    EQU   *
*
* XMODE TRAP HANDLER, ATTACHES OVERFLOW.
*
          BE    ACF,15,GOOD.REG  If not R15, enter Debugger
          ZERO  XMODE            Prevent endless loop
          ENT   5,DB1            Enter debugger with FLZ
GOOD.REG  MOV   R15FID,RECORD    Set up for ATTOVF
          BSL   ATTOVF           Link up another frame
          SETUP R15,FRM.SIZE,R15FID
                                 last byte of frame
          RTN   *                Done
```

System Debugger

The System Debugger is mainly a tool for monitoring and controlling the execution of assembly language programs. To accomplish this, facilities are provided:

to step through instructions,

to set breakpoints,

to display and modify data anywhere in virtual memory,

to verify the proper operation of programs,

to identify programming errors.

Other uses of the Debugger are:

to terminate program execution;

to enable or disable terminal display;

to send messages between processes;

to report and log system abort conditions.

Since full utilization of the Debugger allows changing data anywhere in the system, security becomes an important consideration. Therefore, most debugger commands require SYS2 privileges. A user with SYS0 or SYS1 privileges is essentially restricted to resuming the execution of a program, terminating the execution of a program, or logging off.

Entering The Debugger

The Debugger is entered for three different reasons:

1) to respond to a break condition,

2) to respond to a system abort condition,

3) in a transitory manner, to take advantage of its distinct environment.

Break Conditions

The most common break condition is caused by pressing the terminal <BREAK> key (<CTRL-BREAK> on some terminals). Generally, you may enter the Debugger at will simply by pressing the <BREAK> key. Program execution can be terminated by pressing the <BREAK> key, then typing the word **END**. Other break conditions, defined from within the Debugger, are breakpoints, data change breaks, instruction step breaks, and program change breaks.

There are times when the Debugger will not respond to the <BREAK> key. Certain system functions, for example, inhibit break until they finish what they have begun. For these cases, the break is postponed, and the effect is that there may be a noticeable delay between the time the break key is depressed, and the time the Debugger is entered. Another example is that an application might be implemented in such a way that the break key is always inhibited. In this case, pressing <BREAK> has no effect.

When the Debugger is entered due to a break condition, it identifies the break by displaying whichever of the following is applicable:

Break Condition	Indication
Break Key	i fid:ddd
Breakpoint	b fid:ddd
Data Change	i fid:ddd y fid.ddd data...
Instruction Step	e fid:ddd
Program Change	m fid:ddd | r fid:ddd

"fid.ddd data..." represents the frame number and displacement of the data element being monitored, followed by its data.

After displaying the reason for a break, the Debugger displays the contents of any data elements which are being traced, then prompts with an exclamation point (!) for what to do next. The following format is typical:

```
i fid:ddd
!
```

System Abort Conditions

Hardware failures or assembly-level programming errors can cause situations where the execution of an instruction or the performance of a system function becomes either illegal or illogical. When this happens, the task in progress is aborted. The Debugger is entered to report the reason for the abort by displaying one of the following messages:

ILLEGAL OPCODE ABORT @ fid:ddd

RTN STACK FORMAT ERR ABORT @ fid:ddd

REFERENCING FRAME ZERO ; reg= n abort @ fid:ddd

CROSSING FRAME LIMIT ; reg= n abort @ fid:ddd

FORWARD LINK ZERO ; reg= n abort @ fid:ddd

BACKWARD LINK ZERO ; reg= n abort @ fid:ddd

PRIVILEGED OPCODE ABORT @ fid:ddd

REFERENCING ILLEGAL FRAME ; reg= n abort @ fid:ddd

GROUP FORMAT ERROR XXXXXX

Transitory Debugger Entries

Sometimes the Debugger is entered and exited without the user being aware. One example of this is testing to see if a break condition has been met. Any time the program step mode is in effect, or any breakpoints or data change breaks are defined, the Debugger is entered prior to the execution of each virtual instruction. Then, unless a break condition has been satisfied, it is exited, only to be entered again for the next instruction. This activity causes system performance to degrade when breakpoints, especially data change breaks, are set.

A second example is receiving a message from another process. The Debugger performs this function because it can interrupt a task in progress, display the message, then resume the task at the point it was interrupted.

Referencing Data

Data may be referenced from the Debugger directly or indirectly; numerically, by frame and displacement; or symbolically, by name. The symbol tables are defined from TCL before symbolic references by using the **SET-SYM** verb.

Data Specification

Data locations and display formats are defined for the Debugger through data specifications, or <data spec>'s. The format of a <data spec> is as follows:

{prefix}{address}{suffix}

Data, from anywhere in virtual memory, may be displayed by entering a <data spec> in response to the Debugger's exclamation mark prompt. Data specifications may also be placed in the trace or data change tables to cause the selected data fields to be displayed on break conditions or monitored for change.

Prefix

{**prefix**}{address}{suffix}

The prefix is a single character which specifies the data display mode:

C	character display
I	integer display
X	hexadecimal display

If a prefix is not included in a <data spec>, then the previous, or default, data display mode is used. Hexadecimal is the default.

Address

{prefix}{**address**}{suffix}

The address defines the location of the data element in virtual memory. It may be presented numerically or symbolically, and may specify a direct or indirect addressing mode.

A direct numeric address consists of a frame number and displacement, or a displacement alone (that is, fid.ddd or ddd). The PCB FID is assumed if a frame number is not specified.

1024.10	the 16th byte (X'10') in frame 1024 (period means hex)
1024,12	the 12th byte (X'0C') in frame 1024 (comma means decimal)

A direct symbolic address consists of a symbol name preceded by "/." A window size, which may be overridden, is automatically determined by the symbol type.

/OBBEG	OBBEG storage register in the PCB
/CTR21	CTR21 in the SCB
/FP0	arithmetic accumulator

An indirect address may consist of a register name, a symbol name preceded by an asterisk (*), or a numeric address preceded by an asterisk. A symbolic element specified for indirect addressing must be a register or storage register. A location specified numerically is treated as a storage register. An error message is displayed if it does not contain a valid frame number and displacement.

R14	address of R14
*R14	address pointed to by R14
*OBBEG	address pointed to by OBBEG
*.1D4	also address pointed to by OBBEG

If an address is not included in a <data spec>, then the next data window beyond the previous one is used.

Suffix

{prefix}{address}{**suffix**}

The suffix begins with a semicolon. It may include combinations of the following: a character specifying the element type (T), an offset from the specified address (O), and a number defining the width of the data field (W).

The format of a suffix is:

;{T}{O}{W}

T	type - a single character specifying the element type
O	offset - a positive or negative number specifying an offset from the address
W	width - a positive number defining the width of the data field (that is, data window)

A window definition (that is, ;W) specifies the width of the data element being monitored or displayed.

;16	16 byte data window

An explicit offset may be used in conjunction with a window definition (that is, ;O,W). The data reference is to the address, plus the offset, for the window specified.

;10,8	8 byte data window at address+10
;-5.10	16 byte data window at address-5

If a type is specified in a suffix, the field width and display format are determined automatically. However, either may be overridden by using a prefix or window definition. The characteristics implied by the various types are as follows:

Type	Display Format	Window Size
B	bit	1 bit
C	character	1 byte
D	integer	4 bytes (double tally)
F	integer	6 bytes (triple tally)
H	integer	1 byte (half tally)
R	hexadecimal	8 bytes (register)
S	hexadecimal	6 bytes (storage register)
T	integer	2 bytes (tally)

An offset, used in conjunction with a type definition, expresses multiples of the field width implied by the type.

;S	6 byte hexadecimal data window (storage register)
;D5	4 byte integer data window at address+(5*4)
;B4,4	4 bit data window beginning at address+4 bits

When a suffix defines the type as "bit," a displacement in a numeric address (that is, fid.ddd) is also assumed to be in bits.

If a suffix is not included in a <data spec>, then the previous, or default, window definition is used. ;8 is the default.

Data Reference Examples

These examples show several ways data may be displayed from the Debugger:

<data spec>	Displayed Data
X1024.0;8	Hexadecimal display of 8 bytes beginning at displacement 0 of frame 1024
X.100;8	Hexadecimal display of 8 bytes beginning at location X'100' in the PCB (that is, R0).
/R0	Same as above.
CR15;16	Character display of 16 characters beginning at the location R15 points to.
/OBBEG	Contents of the OBBEG storage register.
C*OBBEG;32	Character display of 32 characters beginning at the location OBBEG points to.
/ABIT;4	Bit display of 4 bits beginning with ABIT.

Debugger Commands

The Debugger prompts for a command with the exclamation mark (!). Any of these commands may then be entered, terminated by a carriage return. If an invalid command or illegal address is entered, the Debugger displays an error message and prompts for a new command.

A	**Address** - The A command, without an argument, displays the address of the next virtual assembly instruction to be executed.
A/elmtname	If an element is specified, the address of the element is displayed.
B B/fid B/fid.ddd	**Set breakpoint** - An entry is placed in the breakpoint table to cause a break to occur whenever the address, specified by mode and displacement, is encountered during instruction execution. If no address is specified, the Debugger breaks at all instructions in the mode. A maximum of four breakpoints may be in effect at a time. A "+" is displayed if the breakpoint is added; "TBL FULL" is displayed if the table already holds four entries. The B command, without an argument, displays "-" and removes all entries from the breakpoint table.
C{addr}{suffix}	**Character display** - The C command puts the Debugger in character display mode. In this mode, each character from storage is converted to a printable ASCII character for display; non-printable characters are represented as periods (for example, x'0061626300' is displayed as ".abc."). Normally, C is used as a prefix in a data specification. When used alone, it displays the next data window beyond the one previously displayed.
D	**Display tables** - The current state of the breakpoint, trace, and data change tables are displayed in the following order: BRK TBL TRC TBL CHG TBL
E{n}	**Toggle instruction step mode** - The E command alone turns off the single step mode. With instruction stepping enabled, a break to the Debugger occurs following the execution of each virtual instruction.
END <cr>	**End** - The Debugger tables and other control elements are initialized, then the user's process is sent to the TCL Command Processor. This command is used to conclude a debugging session.

G

Go - The G command, without an argument, causes instruction execution to proceed with the next virtual instruction. Typing linefeed is equivalent to this command form, and only requires one keystroke. If the G command is used with an illegal address, or with no address following a system abort condition, "ADDR" is displayed, then the Debugger prompts for a new command.

G/fid
G/fid.ddd
G/fid{.ddd}

If a FID is specified, instruction execution commences at the mode and displacement given, or at the beginning of the mode if a displacement is not provided.

I{addr}{suffix}

Integer display - This command puts the Debugger in integer display mode. In this mode the data from storage is treated as a numeric element (that is, a half tally, tally, double tally, or triple tally) and is displayed as a decimal integer (for example, if T0 = x'FFFF', then I/T0 displays "-1"). A window specification greater than six bytes (that is, > triple tally) may cause unexpected results, as only the low-order tally is converted and displayed. Normally, I is used as a prefix in a data specification. When used alone, it displays the next data window beyond the one which was previously displayed.

K
K/fid
K/fid.ddd

Remove (kill) breakpoint - The entry, specified by FID and displacement, is removed from the breakpoint table. A displacement of zero is assumed if one is not provided. A "-" is displayed if the breakpoint is removed; "NOT IN TBL" is displayed if the defined breakpoint is not in the table. The K command, without an argument, displays "-" and removes all entries from the breakpoint table.

L
L<data spec>

Display/modify frame links - The L command displays the link fields of the frame identified by <data spec> in the following format:

fid NNCF : FRMN FRMP : NPCF =

FID = current frame number
NNCF = number of next contiguous frames
FRMN = next linked frame
FRMP = prior linked frame
NPCF = number of prior contiguous frames

M

Toggle program step mode (modal trace) - The M command alternately enables or disables program stepping; "on" or "off" is displayed as appropriate. With program stepping enabled, a break to the Debugger occurs following the execution of each virtual instruction that causes a transfer from one assembly program (mode) to another. The program name and displacement of the next virtual instruction are displayed:

M fid:ddd | r fid:ddd

An "M" indicates entering a new mode through a B or BSL instruction. An "R" indicates returning to a previous mode, to the location following a BSL.

ME
ME linenumber

Assign PCB by line - The ME command causes subsequent debugger PCB references to be made to the PCB associated with the specified line number (that is, someone else's PCB).

The ME command, without an argument, causes the Debugger to revert to using the PCB associated with the active process (that is, your own PCB).

N
N skipcount

Set break skip count - The N command establishes a number of break conditions that the Debugger will skip before actually breaking, and prompting for a command. The trace data for all break conditions are displayed whether a break occurs or not. A system abort condition is handled immediately, regardless of the skip count.

The N command, without an argument, sets the break skip count to zero to cause the Debugger to break for all break conditions.

OFF

Logoff - The active process is logged off and returned to the logon process. Executing this command is equivalent to executing "OFF" at TCL.

P

Toggle terminal display mode - The P command alternately enables or disables terminal output display, by setting or zeroing LISTFLAG. "ON" or "OFF" is displayed, as appropriate.

R
R regnumber

Display data through register - The R command, when used with a number between 0 and 15, displays data indirectly through the specified register (i.e., the data pointed to by the register). The data display type and window may be specified concurrently (for example, XR15.10;8); otherwise, the most recent, or default, specification is used.

The R command, without an argument, displays the next data window beyond the one which was previously displayed.

T
T<data spec>

Set trace definition- An entry placed in the trace table causes the specified data window to be displayed whenever a break occurs. A maximum of eight trace definitions may be in effect at a time. A "+" is displayed if the trace definition is added; "TBL FULL" is displayed if the table already holds eight entries.

The T command, without an argument, displays "-" and removes all entries from the trace table.

TIME

Display time and date - The current system time and date are displayed as follows:

hh:mm:ss dd mmm yyyy

U
U<data spec>

Remove trace definition (untrace) - The entry, identified by the data specification, is removed from the trace table. A "-" is displayed if the trace definition is removed; "NOT IN TBL" is displayed if the defined entry is not in the table.

The U command, without an argument, displays "-" and removes all entries from the trace table.

X{addr}{suffix}

Hexadecimal display - The X command puts the Debugger in hexadecimal display mode. In this mode, each character from storage is converted to two printable ASCII hexadecimal characters for display. Normally, X is used as a prefix in a data specification. When used alone, it displays the next data window beyond the one which was previously displayed.

Y
Y<data spec>

Set data change definition - An entry placed in the data change table causes a break to occur if the specified data changes. A maximum of three data change definitions may be in effect at a time. A "+" is displayed if the data change definition is added; "TBL FULL" is displayed if the table already holds three entries. Using the data change feature makes the system runs significantly slower.

The Y command, without an argument, displays "-" and removes all entries from the data change table.

Z
Z<data spec>

Remove data change definition - The entry, identified by the data specification, is removed from the data change table. A "-" is displayed if the data change definition is removed; "NOT IN TBL" is displayed if the entry is not in the table.

The Z command without an argument displays "-" and removes all entries from the data change table.

Arithmetic Utility Commands

The arithmetic utility commands may be executed from either the Debugger or TCL. For accurate results, the numeric arguments should be whole numbers, with no punctuation other than a leading plus or minus sign.

ADDD n1 n2	**Add decimal to decimal** - The two decimal numbers are added and the sum is displayed.
ADDX n1 n2	**Add hexadecimal to hexadecimal** - The two hexadecimal numbers are added and the sum is displayed.
DIVD n1 n2	**Divide decimal by decimal** - The decimal number n1 is divided by the decimal number n2 and the quotient and remainder are displayed.
DIVX n1 n2	**Divide hexadecimal by hexadecimal** - The hexadecimal number n1 is divided by the hexadecimal number n2 and the quotient and remainder are displayed.
DTX n	**Convert decimal to hexadecimal** - The decimal number n is converted to its hexadecimal equivalent and displayed.
MULD n1 n2	**Multiply decimal times decimal** - The two decimal numbers are multiplied and the product is displayed.
MULX n1 n2	**Multiply hexadecimal times hexadecimal** - The two hexadecimal numbers are multiplied and the product is displayed.
SUBD n1 n2	**Subtract decimal from decimal** - The decimal number n2 is subtracted from the decimal number n1 and the difference is displayed.
SUBX n1 n2	**Subtract hexadecimal from hexadecimal** - The hexadecimal number n2 is subtracted from the hexadecimal number n1 and the difference is displayed.
XTD n	**Convert hexadecimal to decimal** - The hexadecimal number n is converted to its decimal equivalent and displayed.

Interacting with the Debugger

The Debugger prompts for a command with the exclamation mark (!). A command or <data spec>, terminated by a carriage return, may be entered in response. Depressing the linefeed key at the "!" prompt is equivalent to typing "G<cr>." That is, it causes the next virtual instruction to be executed. This feature is often used when stepping through instructions because it only requires one keystroke.

Entering an invalid command, naming an undefined symbol, or using an illegal address causes "CMND?," "ILLGL SYM," or "ADDR" to be displayed. The Debugger prompts for a new command after displaying an error message.

If a <data spec> is entered, the defined data window is displayed, followed by an equal sign (=):

```
!/ABIT<cr> 0=
```

The "=" is a solicitation to change the data being displayed:

```
!/ABIT<cr> 0=1<cr>
!/ABIT<cr> 1=
```

New data may be entered to modify that shown. Alternatively, a carriage return may be typed to return to the command processor; a <CTRL> P may be typed to display the previous data window; or a <CTRL> N, or linefeed, may be typed to display the next data window. A <CTRL> N and linefeed differ in that a <CTRL> N causes the window to be displayed on a new line preceded by its address, while a linefeed causes it to be displayed on the same line with no address. Any of the characters just mentioned may also be used to terminate data entry.

Debugger Data Entry Conventions

In some cases, the new data must be compatible with the data being modified. The following is a summary of conventions and restrictions which apply when entering data:

<bits>	**Bit string insertion** - If the display type is "bit," then the input string must only contain 1's and 0's, must be less than 40 characters long, and must be no greater than the width of the display window. The bits in the display window are replaced by the bits in the input string, beginning from the left.
'<characters>	**Character string insertion** - A character string is preceded by a single quote ('). It must contain printable characters, and must be less than 39 characters long. The characters in the display window are replaced by those in the input string, beginning from the left.

.<hexadecimal> **Hexadecimal string insertion** - A hexadecimal string is preceded by a period (.). It must only contain hexadecimal characters, must contain an even number of nibbles, and must not be more than 19 characters long. The characters in the display window are replaced by the characters in the input string, beginning from the left.

{+}{-}<integer> **Integer insertion** - An integer may contain a leading plus or minus sign; otherwise, it must only contain decimal characters. The display window is treated as a numeric element (that is, half tally, tally, double tally, or triple tally). It must be 1, 2, 4, or 6 bytes in length, and must not cross a frame boundary. The new integer replaces all previous data in the window.

Editing functions which may be used while entering data are:

<CTRL> H (backspace),
<CTRL> R (redisplay),
<CTRL> W (cancel last word), and
<CTRL> X (cancel line).

Functions described for the system subroutine, "READLIN", are not supported.

Appendix A

ABS Frames

Frames	Used By
001 - 207	Pick Systems
208 - 208	Mainlink
209 - 221	Pick Systems
222 - 223	Jet
224 - 311	Pick Systems
312 - 313	Compu-sheet
314 - 339	Pick Systems
340 - 380	Jet
381 - 398	Pick Systems
399 - 399	Accu-Cursor
400 - 407	Pick Systems
408 - 417	Accu-Plot
418 - 419	Pick Systems
420 - 421	Compu-Sheet
422 - 422	Pick Systems
423 - 428	Compu-Sheet
429 - 429	Pick Systems
430 - 459	Accu-Plot
460 - 467	Compu-Sheet
468 - 469	Pick Systems
470 - 485	Mainlink
486 - 511	Compu-Sheet
512 - 559	Jet
560 - 572	Mainlink
573 - 574	**Available**
575 - 591	Mainlink
592 - 599	**Available**
600 - 703	Keyword
704 - 899	**Available**
900 - 1023	Pick Systems

Appendix B

Process Workspace

PCB Offset	Name	Size	Description
0	PCB	512	Accumulators Address Registers Return Stack
1	SCB	512	Counters Storage Registers
2	DCB	512	System Debugger Control Block
3	QCB	512	Used by System Software Routines
4	BMS	50	Disc File I/O
	AF	50	Scratch
	CS	100	Scratch
	IB	0-140	Terminal Input Routine
	OB	0-140	Terminal Output Routine
5	TS	512	Conversions
6 - 9	PROC	2048	PROC Working Space
10 - 15	HS	3072	Used for WRAPUP Messages
16 - 21	IS	3072	Used by System Subroutines
22 - 27	OS	3072	Used by System Subroutines
28 - 29	UPD	1024	Used By Data/BASIC Debugger
29 - 31		1536	System Software Routines PickWare or available

Workspace	Linked?	Remarks
BMS	No	Normally contains an item-ID when communicating with the disc file I/O routines; typically, the item-ID is copied to the BMS area, starting at BMSBEG+1; BMSBEG may be moved to point within any scratch area. BMSEND normally points to the last byte of the item-ID; BMS (A/R) is freely usable except when explicitly or implicitly calling a disc file I/O routine.
AF	No	These workspaces are used by any system subroutine,
CS	No	though the A/R's are used as a scratch registers.
IB	No	Used by terminal input routines to read data; IBBEG may be moved to point within any scratch area before use: IBEND conventionally points to the logical end of data; IB A/R is freely usable except when explicitly or implicitly calling a terminal input routine.
OB	No	Used by terminal output routines to write data. OBEG and OBEND should not be altered; they always point to the beginning and end of the OB area; OB (A/R) conventionally points one byte before the next available location in the OB buffer.
TS	No	This workspace is used by the system subroutines associated with the Conversion processor, though the TS A/R is used as a scratch register.
PROC	Yes	Used exclusively by the PROC processor for working storage; user-exits from PROC's may change pointers in this area.
HS	Yes	Used as a means of passing messages to the WRAPUP processor at the conclusion of a TCL statement; may be used as a scratch area if there is no conflict with the WRAPUP History String formats; HSBEG should not be altered; HSEND conventionally points one byte before the next available location in the buffer (initial condition is HSBEG=HSEND).
IS	Yes	These workspaces are used interchangeably by some
OS	Yes	system routines since they are the same size (equal to HS). Specific usage is noted under the various system routines. ISBEG and OSBEG should not be altered, but may be interchanged if necessary; initially, ISEND and OSEND point 3000 bytes past ISBEG and OSBEG respectively. IS and OS A/R's are freely usable except when calling system subroutines that use them.

Appendix C

Register Conventions

Registers 0 and 1 have specifically defined meanings; they should never be changed in any way by programs. The other fourteen may be considered general purpose registers with the limitation that system software conventions determine the usage of most address registers.

Register	Synonym	Usage
R0	**PCB**	Addresses byte zero of the process' Primary Control Block
R1	**Program Counter**	When the process is inactive, addresses the location of the next instruction to execute. When active, addresses byte zero of the frame in which the process is currently executing.
R2	**SCB**	Points to the Secondary Control Block at logon and after entering the debugger or WRAPUP Processor.
R3	**HS**	History String, Used for WRAPUP messages. Exit to TCL via WRAPUP resets R3 to PCB+10.
R4	**IS**	Used by System Subroutines
R5	**OS**	Used by System Subroutines
R6	**UPD**	Used by the Pick/BASIC Debugger
R7	**IR**	Used by RETIX and for Item Retrieval
R8	**BMS**	Used for Disc File I/O
R9	**AF**	Used by System Subroutines and as scratch
R10	**IB**	Freely usable except when explicitly or implicitly calling a Terminal Input Routine.
R11	**OB**	Freely usable except when explicitly or implicitly calling a Terminal Output Routine.
R12	**CS**	Used by System Subroutines, and as scratch
R13	**TS**	Used by System Subroutines associated with the Conversion Processor, and as scratch
R14	Scratch	Used by System Subroutines; Available to the User
R15	Scratch	Used by System Subroutines and some instructions; Available to the User

Appendix D

Linked Frame Format

The first frame of a linked set of frames will have zero *npcf* and *backward link* fields. The last frame of such a set will have zero *nncf* and *forward link* fields. The *nncf* and *npcf* fields are normally zero, except in the linked workspace allocated to each process and in files that have a separation greater than one.

Byte	Display	Description
0	*	Unused byte
1	*nncf*	Number of next contiguous frames (count of frames that are linked forward off this one, whose FIDs are sequential to this)
2-5	*forward link*	FID of the frame that is next in logical sequence to this one
6-9	*backward link*	FID of the frame that is logically previous to this one
A	*npcf*	Number of previous contiguous frames (count of frames that are linked backwards to this one, whose FIDs are sequential to this)
B	*	Unused byte
C	*Start of data*	Following the link fields is the 500-byte data block.

PICK®
SYSTEMS

Customer Comment Form

R83 Assembly Manual
Jan. 1990 Edition

Pick Systems would like to hear from your regarding the **R83 Assembly Manual**. Your comments, recommended enhancements and suggested corrections on both the product and this manual will assist us in making improvements.

Please indicate the type of user/reader you most nearly represent:

☐ Systems Programmer ☐ Applications Programmer
☐ Consultant ☐ Prospective User

Comments: _____

Enhancements:
Is any material missing? If so, please describe and indicate where it should be placed:_____

Please make suggestions for improvement:_____

Corrections:

Page Number	Description

Thank you.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7303 IRVINE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

PICK®
SYSTEMS

1691 Browning
Irvine, CA 92714

PICK®

SYSTEMS

1691 Browning, Irvine, CA 92714
(714) 261-7425 FAX (714) 250-8187