REFERENCE MATERIAL
FOR
CCS 476 AND CCS 573

L. K. Flanigan
W. E. Riddle

Department of Computer and Communication Sciences
University of Michigan

Fall/Winter 1972-73
The reference material in this volume is an amalgam of two sets of notes: a set prepared by Professor Riddle for CCS 476; and a set prepared by Professor Flanigan for CCS 573. Since the two courses each provide lectures on System/360, we decided to combine the two sets of notes into one volume for both courses to use. The bulk of this reference material will be used in both courses; some individual sections may be used directly by only one of the courses. This material is designed to provide backup, reference material for the course lectures; it is not self-contained, and it is not designed to provide self-instruction or to aid independent study. Frequent reference will be made to this material in the lectures, and students are therefore encouraged to carry these notes to the lectures in each course. We also hope that this material will serve as a reference to basic System/360 concepts throughout the course, thereby reducing the need to peruse the IBM manuals and the MTS manuals. We would appreciate any comments, corrections, and/or suggestions generated by this reference material.

L. K. Flanigan

W. E. Riddle
TABLE OF CONTENTS

I. INTRODUCTION TO SYSTEM/360 ........................................ II.1
II. FIXED AND FLOATING POINT INSTRUCTION SETS ...................... III.1
III. BRANCHING INSTRUCTION SET ......................................... IV.1
IV. INTRODUCTION TO ASSEMBLER LANGUAGE ............................ V.1
V. DECIMAL INSTRUCTION SET ............................................. VI.1
VI. LOGICAL INSTRUCTION SET ............................................. VII.1
VII. EXAMPLE PROGRAMS .................................................. VIII.1
VIII. INTRODUCTION TO MACRO ASSEMBLER ............................ IX.1
IX. STATUS SWITCHING INSTRUCTION SET ................................ X.1
X. I/O INSTRUCTION SET ................................................ XI.1
XI. THE U. OF M. SYSTEM/360 MODEL 67 ................................ XII.1
XII. EXCERPTS FROM MTS VOL. 3 .......................................
BASIC SYSTEM/360 STRUCTURE:

CONCEPTUAL CPU-STORAGE ORGANIZATION:
MAIN STORE -- DATA TYPES

FIXED UNITS:  
- BYTE = 8 BITS
- HALFWORD = 2 BYTES
- FULL WORD = WORD = 2 HALFWORDS = 4 BYTES
- DOUBLE WORD = 2 WORDS = 8 BYTES

DATA TYPES:  
- FIXED POINT -- HALFWORD, FULL WORD
- FLOATING POINT -- WORD, DOUBLE WORD, EXTENDED (16 BYTES)
- DECIMAL -- 1 TO 31 DIGITS + SIGN (UP TO 16 BYTES)
- CHARACTER -- 1 TO 256 CHARACTERS (BYTES)

FIXED POINT FORMAT:

- HALFWORD
  \[ S \begin{array}{c} 1 \vphantom{1} \\ \hline -15 \end{array} \]
  \[ S = \begin{cases} 0 & \text{positive} \\ 1 & \text{negative} \end{cases} \]

- FULL WORD
  \[ S \begin{array}{c} 1 \vphantom{1} \\ \hline -31 \end{array} \]

MAXIMUM POSITIVE: \(7F\) \(FFFF\) = 2,147,483,647 (32767)
MAXIMUM NEGATIVE: \(80000000\) = -2,147,483,648 (-32768)

FLOATING POINT FORMAT:

- SHORT
  \[ S \begin{array}{c} 1 \vphantom{1} \\ \hline 78 \end{array} \]
  \[ \sim 7 \text{ DEC PLACES} \]

- LONG
  \[ S \begin{array}{c} 1 \vphantom{1} \\ \hline 78 \end{array} \]
  \[ \sim 17 \text{ DEC PLACES} \]

CHARACTERISTIC     FRACTION
EXTENDED: DATA TYPE IS TWO LONG FLOATING POINT NUMBERS

VALUE: \(S \cdot \text{FRACTION}_{16} \cdot (\text{CHARACTERISTIC} - 10)_{10}\)

RANGE: \((5.1)_{10} - 79 \ldots (7.2)_{10} 75\)

CHARACTER FORMAT:
ONE CHARACTER = ONE BYTE; THUS, UP TO 256 DISTINCT CHARACTERS
SAMPLE EBCDIC ENCODINGS:

\begin{tabular}{|c|c|c|c|}
\hline
0-9 & F0-F9 & + & 4E \hline
A-I & C1-C9 & \vphantom{+} & 60 \hline
J-R & D1-D9 & \vphantom{+} & 4B \hline
S-Z & E2-E9 & \vphantom{+} & 5E \hline
\hline
\end{tabular}
DECIMAL FORMAT:

ZONED FORMAT:

\[
\begin{array}{cccccccc}
\end{array}
\]

- Z = ZONE = HEX F
- D = DIGIT = O -- 9
- S = SIGN = + HEX C (1100)
- HEX D (1101)

PACKED DECIMAL:

\[
\begin{array}{cccccccc}
  & D & D & D & D & - & - & - & D & D & D & S \\
\end{array}
\]

- D = DIGIT = O -- 9
- S = SIGN = + HEX C (1100)
- HEX D (1101)

UP TO 16 BYTE LENGTH = UP TO 31 DIGITS + SIGN

EXAMPLE:

<table>
<thead>
<tr>
<th></th>
<th>+68h</th>
<th>-68h</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIXED POINT FORMAT</td>
<td>000002AC</td>
<td>FFFFDF54</td>
</tr>
<tr>
<td>FLOATING POINT FORMAT</td>
<td>432AC000</td>
<td>C32AC000</td>
</tr>
<tr>
<td>PACKED DECIMAL FORMAT</td>
<td>0000684C</td>
<td>0000684D</td>
</tr>
<tr>
<td>ZONED DECIMAL FORMAT</td>
<td>F068F8CL</td>
<td>F068F8DL</td>
</tr>
<tr>
<td>CHARACTER STRING</td>
<td>4E6F8PL</td>
<td>6E6F8PL</td>
</tr>
</tbody>
</table>

CONVERSIONS:

SCARDS, READ

<table>
<thead>
<tr>
<th>CHARACTER STRING</th>
</tr>
</thead>
</table>

SPRINT, WRITE

<table>
<thead>
<tr>
<th>BY PROGRAM (MOVE SIGN)</th>
</tr>
</thead>
</table>

PACK

<table>
<thead>
<tr>
<th>ZONED DECIMAL</th>
</tr>
</thead>
</table>

UNPK

<table>
<thead>
<tr>
<th>PACKED DECIMAL</th>
</tr>
</thead>
</table>

CVB

<table>
<thead>
<tr>
<th>FIXED POINT</th>
</tr>
</thead>
</table>

CVD

<table>
<thead>
<tr>
<th>FLOATING POINT</th>
</tr>
</thead>
</table>

BY PROGRAM
CONCEPTUAL CONTROL SEQUENCE:

ILR = INSTRUCTION LOCATION REGISTER -- CONTAINS THE ADDRESS OF THE FIRST BYTE OF THE NEXT INSTRUCTION TO BE EXECUTED (HALF WORD ALIGNED)

1. NEXT INSTRUCTION FETCHED FROM STORAGE AS SPECIFIED BY ILR
2. ILR UPDATED BY LENGTH OF INSTRUCTION FETCHED
3. INSTRUCTION DECODED IN CONTROL UNIT
4. INSTRUCTION OPERANDS FETCHED FROM STORAGE, IF NECESSARY
5. INSTRUCTION EXECUTED USING SPECIFIED OPERANDS
6. RESTART CYCLE -- GO TO STEP 1.

INSTRUCTIONS FETCHED SEQUENTIALLY FROM STORAGE EXCEPT FOR BRANCH, INTERRUPT, OR STATUS SWITCH.

INSTRUCTION FORMATS:

<table>
<thead>
<tr>
<th>TYPE</th>
<th>BITS</th>
<th>0,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>RX</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

INSTRUCTIONS MUST BE HALF WORD ALIGNED.

ADDRESS COMPUTATION:

RR: NONE
RX: $D + C(B) + C(X)$
RS, SI, SS: $D + C(B)$

$B_n, X$ SPECIFY GBP Rs IF NON-ZERO; D IS FROM INSTRUCTION ITSELF; ADDRESS COMPUTED PRIOR TO INSTRUCTION EXECUTION

OPERAND LENGTH SPECIFICATION:

RR, RX, RS, SI: IMPLIED BY OP CODE -- ONE OP FOR EACH TYPE AND LENGTH
SS: SPECIFIED IN LENGTH FIELD(S) OF INSTRUCTION ITSELF
PROGRAM STATUS WORD (PSW):

<table>
<thead>
<tr>
<th>SYSTEM MASK</th>
<th>KEY</th>
<th>AMWP</th>
<th>INTERRUPTION CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

SYSTEM MASK: BIT 0-6 MASK CHANNELS 0-6, RESPECTIVELY

BIT 7 MASKS EXTERNAL INTERRUPT (TIMER)

O = MASKED; 1 = UNMASKED

KEY:

STORAGE PROTECTION KEY

4 BIT STORE PROTECT -- 1 BIT FETCH PROTECT

MAY PROTECT BLOCKS OF 2048 BYTES

PSW KEY MUST MATCH STORAGE BLOCK KEY IF USED

AMWP:

A (12) 0 = EBCDIC; 1 = ASCII

M (13) 0 = MACHINE CHECK MASKED; 1 = UNMASKED

W (14) 0 = RUNNING STATE; 1 = WAIT STATE

P (15) 0 = SUPERVISOR STATE; 1 = PROBLEM STATE

INTERRUPTION CODE: THIS IS INFORMATION STORED WITH THE PSW WHENEVER AN INTERRUPT IS ACCEPTED BY THE CPU. THE INFORMATION DESCRIBES THE CAUSE OF THE INTERRUPT AND DEPENDS UPON THE TYPE OF INTERRUPT.

<table>
<thead>
<tr>
<th>ILC</th>
<th>CC</th>
<th>PROGRAM MASK</th>
<th>INSTRUCTION LOCATION REGISTER (ILR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>34</td>
<td>36</td>
<td>40</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ILC: INSTRUCTION LENGTH CODE = LENGTH OF CURRENTLY EXECUTING INSTRUCTION IN HALF WORDS (2 = FULL WORD INSTRUCTION)

CC: CONDITION CODE -- SET BY ALL TEST AND COMPARE INSTRUCTIONS, MANY ARITHMETIC INSTRUCTIONS, AND SEVERAL OTHERS, TO INDICATE RESULT OF THE INSTRUCTION EXECUTION; CC MAY ITSELF BE TESTED FOR BRANCHING PURPOSES

PROGRAM MASK: BIT 36 = FIXED POINT OVERFLOW MASK

BIT 37 = DECIMAL OVERFLOW MASK

BIT 38 = EXPONENT UNDERFLOW MASK

BIT 39 = SIGNIFICANCE MASK

ILR: INSTRUCTION LOCATION REGISTER -- EXCEPT FOR STATUS SWITCH OR BRANCH OR INTERRUPT, NEXT INSTRUCTION TO BE EXECUTED IS BY DEFINITION THE INSTRUCTION LOCATED AT THE ADDRESS IN THE ILR

NOTE: IN CASE OF MULTIPLE CPUS IN THE SYSTEM, EACH CPU HAS ITS OWN PSW.
**INTERJPT CLASSES:**

**INTERRUPT PRIORITY**
- MACHINE CHECK
- PROGRAM OR SVC
- EXTERNAL
- I/O

**PROCESSING PRIORITY**

**CPU INTERRUPT PROCESSING:**

CPU processes interrupt when it arises if the CPU is in a state of accepting interrupts and the specific interrupt is not masked. CPU processing is as follows:

1. Current instruction in CPU is allowed to finish if possible
2. Current PSW is formed with appropriate interruption code
3. Current PSW is stored in old PSW location for interrupt class
4. PSW found at new PSW location for interrupt class becomes current PSW

Note: If masked, program interrupts are lost, while I/O, external, and/or machine check interrupts remain pending.

**OLD AND NEW PSW LOCATIONS (ADDRESSES IN HEX):**

<table>
<thead>
<tr>
<th>OLD PSW</th>
<th>INTERRUPT CLASS</th>
<th>NEW PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>EXTERNAL</td>
<td>58</td>
</tr>
<tr>
<td>20</td>
<td>SVC</td>
<td>60</td>
</tr>
<tr>
<td>28</td>
<td>PROGRAM</td>
<td>68</td>
</tr>
<tr>
<td>30</td>
<td>MACHINE CHECK</td>
<td>70</td>
</tr>
<tr>
<td>38</td>
<td>I/O</td>
<td>78</td>
</tr>
</tbody>
</table>

Each location is double word; for multiple CPUs, each has a unique set of old/new PSW locations.

**INTERRUPTION CODE:**

**MACHINE CHECK:** Code ignored

**SVC:**
- Bits 2h-31 set to second byte of SVC
- Bits 21-23 set to channel address
- Bits 2h-31 set to device address
- CSW (location 40) also stored

**I/O:**
- Bit 24 = timer
- Bit 25 = attention key
- Bits 26-31 = external signals 2-7

**EXTERNAL:**
- Bit 24 = timer
- Bit 25 = attention key
- Bits 26-31 = external signals 2-7

**PROGRAM:**
- 1 = operation
- 2 = privileged operation (M)
- 3 = execute (Ex)
- 4 = protection (P)
- 5 = addressing (A)
- 6 = specification (S)
- 7 = data (D)
- 8 = fixed point overflow (IF)
- 9 = fixed point divide (IK)
- A = decimal overflow (DF)
- B = decimal divide (DK)
- C = exponent overflow (E)
- D = exponent underflow (U)
- E = significance (IS)
- F = floating point divide (FK)

(* = may be masked in program mask)
SEMANTICS OF THE SYSTEM/360 MACHINE INSTRUCTIONS

Each of the following sections of these reference notes covers a different subset of the System/360 machine instructions. At the beginning of each of the sections (except the section on the input/output instructions), all of the instructions covered in that section are listed. For each instruction is given its assembler language format, an indication of whether or not its execution affects the setting of the condition code (if the instruction is prefaced with a '*' then it does affect the setting of the condition code), and an explanation of the semantics of the instruction. The semantics of the instructions are explained by showing their effect on the values in registers and in main storage in terms of the operations carried out on these values. In most cases the operations are simple and a correspondingly simple description of the operation is all that is needed. These descriptions are given in the notation described on the next few pages. Some of the instructions have a rather complex effect and for these, short programs in an easily understood algorithmic programming language, employing statements composed from the notation described below as well as more normal programming language statements, are presented.

Notation for describing semantics

Predefined symbols

- \( R_i, B_i, X_i, D_i, L_i, I_i, M \) and \( L \) are used and they have the same meaning as in the POOL manual. Whether \( R_i \) denotes a general purpose or floating point register is determined by context.
- \( Gti \) refers to the specific general purpose register with name \( i \).
- \( CC \) refers to the two-bit condition code register which is part of the PSW.
- \( IAR \) refers to the 24-bit instruction address register which is also part of the PSW.
- \( PSW \) refers to the 64-bit program status (double)word.
- All numbers should be understood to be decimal numbers unless specified otherwise by giving the radix of the number system as a subscript at the end of the number.
- Temporary locations are defined as needed for saving intermediate results. They are descriptively named (e.g., temp, counter, etc.)
Descriptions of basic operations

Letting:
- \texttt{reg} stand for a specification of one of the general purpose or floating point registers or one of the special registers (i.e., \texttt{VR}, \texttt{PC}, \texttt{SP}, etc.),
- \texttt{addr} stand for an address specification, i.e., the name of one of the general purpose registers or a 24-bit fixed binary number (M.R. for consistency of notation and agreement with the PDP manual, addresses are considered to be 32-bit binary numbers of which only the low-order 24-bits are actually used),
- \texttt{value} stand for an \( n \)-bit binary number, where the size, \( n \), and the interpretation of fields within the \( n \)-bits is specified implicitly by the nature of the instruction being described (e.g., in address arithmetic the numbers are 24-bit ones, in fixed binary arithmetic they are 32-bit ones with the demarcation of a sign field and an integer field, etc.),

then the basic operations are:
- \texttt{(addr)} the result of this operation is a value, namely that stored at the location specified by the address
- \texttt{\{addr\}_{i-j}} this operation limits attention to the selected bit positions (with numbering from the left, starting at zero) within the field at the specified address; in some of the decimal instructions, the \( i-j \) subscripts denote the bytes rather than the bits which are of interest -- this will be apparent from context
- \texttt{\[reg\]} is the value zero if \( \texttt{reg} \) is zero; otherwise it is \texttt{\( \texttt{reg} \)}
- \texttt{\{value\}_{i-j}} this operation has the effect of truncating bits by selecting out bit positions within the binary representation of the value; as with \texttt{\{addr\}_{i-j}}, the \( i-j \) subscripts sometimes denote byte positions
- \texttt{\textit{reg}}, \texttt{\textit{reg}+1} serves to denote the even-odd register pair which starts with \texttt{reg}
value\uparrow means that the value is a 16-bit binary number and should be expanded into a 32-bit binary number by extending the sign-bit into the high-order 16 bit positions

value' denotes the two's-complement of the value

|value| denotes the absolute value of the value

<value> denotes normalization of a floating-point number

value \rightarrow addr reflects the operation of storing the value at the specified address; the width of the field used at the specified location is implied by the size of the value and/or the type of the instruction

value \rightarrow \{addr\}_{i-j} denotes somewhat the same operation except that the value is stored in the subfield between bit positions (or byte positions, in some instances) i and j of the addressed location

convert_{i-j} also denotes somewhat the same operation except that the value is converted from a number in the number system with radix i into one in the number system with the radix j

\begin{center}
\begin{array}{c}
\text{value} \\
\text{\rightarrow} \\
\{addr\}_{i-j}
\end{array}
\end{center}

\begin{center}
\begin{array}{c}
\text{b} \\
\downarrow
\{addr\}_{i-j}
\end{array}
\end{center}

denotes a right shift operation: the bits in the selected subfield of the field at the specified location are shifted right the number of bit positions given by value; the bit b (either zero or one) is shifted into the vacated positions at the left-hand end; bits shifted out of the right-hand end are dropped

\begin{center}
\begin{array}{c}
\text{value} \\
\text{\leftarrow} \\
\{addr\}_{i-j}
\end{array}
\end{center}

\begin{center}
\begin{array}{c}
b \\
\downarrow
\{addr\}_{i-j}
\end{array}
\end{center}

denotes a left shift operation

integer arithmetic or floating-point arithmetic operations depending upon the context where used (note that the previously defined use for parentheses precludes their use for ordering the operations in an expression -- the operators have the normally defined precedence and parentheses are never used to mean an overriding of this precedence)
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+_16$</td>
<td>modulo-16 addition</td>
</tr>
<tr>
<td>$&amp;$, $&amp;$</td>
<td>logical arithmetic on 32-bit unsigned values</td>
</tr>
<tr>
<td>$\land$, $\lor$, $\oplus$</td>
<td>the logical operators and, (inclusive) or, and exclusive or</td>
</tr>
</tbody>
</table>
**FIXED POINT ARITHMETIC INSTRUCTIONS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LR</strong> $R_1, R_2$</td>
<td>$(R_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>L</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$([b_2] + [x_2] + d_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>LH</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$([b_2] + [x_2] + d_2) \uparrow \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>LTR</strong> $R_1, R_2$</td>
<td>$a. (R_2) \rightarrow R_1$; $b. \text{if } (R_1) = \text{condition} \rightarrow \text{condition else } \text{if } (R_2) &lt; \text{condition} \rightarrow \text{condition else } 1 \rightarrow \text{condition}$</td>
</tr>
<tr>
<td><strong>LCR</strong> $R_1, R_2$</td>
<td>$(R_2)' \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>LPR</strong> $R_1, R_2$</td>
<td>$</td>
</tr>
<tr>
<td><strong>LNR</strong> $R_1, R_2$</td>
<td>$</td>
</tr>
<tr>
<td><strong>LM</strong> $R_1, R_3, D_2(b_2)$</td>
<td>$a. ([b_2] + d_2) \rightarrow \text{temp};$ $b. \text{determine } n \text{ such that } R_1 + 16(n) = R_3$ $c. ([\text{temp}]^4) \rightarrow R_1; ([\text{temp}]^8) \rightarrow R_1 + 16 \cdot 1;$ $([\text{temp}]^{16}) \rightarrow R_1 + 16 \cdot 2;$ $\ldots; ([\text{temp}]^{4*4(n)}) \rightarrow R_1 + 16(n)$</td>
</tr>
<tr>
<td><strong>LA</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$a. ([b_2] + [x_2] + d_2) \rightarrow {R_1}<em>{b_2 - 31}$ $b. 0 \rightarrow {R_1}</em>{0-7}$</td>
</tr>
<tr>
<td><strong>AR</strong> $R_1, R_2$</td>
<td>$(R_1) \cdot (R_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>A</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$(R_1) + ([b_2] + [x_2] + d_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>AH</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$(R_1) + ([b_2] + [x_2] + d_2) \uparrow \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>ALR</strong> $R_1, R_2$</td>
<td>$(R_1) \Theta (R_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>AL</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$(R_1) \Theta ([b_2] + [x_2] + d_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>SR</strong> $R_1, R_2$</td>
<td>$(R_1) - (R_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>S</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$(R_1) - ([b_2] + [x_2] + d_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>SH</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$(R_1) - ([b_2] + [x_2] + d_2) \uparrow \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>SLR</strong> $R_1, R_2$</td>
<td>$(R_1) \Theta (R_2) \rightarrow R_1$</td>
</tr>
<tr>
<td><strong>SL</strong> $R_1, D_2(x_2, b_2)$</td>
<td>$(R_1) \Theta ([b_2] + [x_2] + d_2) \rightarrow R_1$</td>
</tr>
</tbody>
</table>
MR  \( R_1, R_2 \)

\( a. \) check that \( R_1 \) is even [assembler does
\( b. \) \((R_1+1) \cdot (R_2) \rightarrow \{R_1, R_1+1\}_{0-63} \)

M  \( R_1, D_2(x_2, B_2) \)

\( a. \) check that \( R_1 \) is even [assembler does:
\( b. \) \((R_1+1) \cdot ([B_2]+[x_2]+D_2) \rightarrow \{R_1, R_1+1\}_{0-63} \)

MH  \( R_1, D_2(x_2, B_2) \)

\( \{(R_1) \cdot ([B_2]+[x_2]+D_2)\mod_{32-63} \rightarrow R_1 \)

DR  \( R_1, R_2 \)

\( a. \) check that \( R_1 \) is even [assembler does:
\( b. \) \(\{(R_1,R_1+1)\}_{0-63} / (R_2) \rightarrow \text{quotient +} \)
\( c. \) \(\{\text{rem}\} \rightarrow R_1 ; \)
\( \{\text{quotient}\} \rightarrow R_1+1 \)

D  \( R_1, D_2(x_2, B_2) \)

\( a. \) check that \( R_1 \) is even [assembler does:
\( b. \) \(\{(R_1,R_1+1)\}_{0-63} / ([B_2]+[x_2]+D_2) \rightarrow \text{quotient + } \text{rem} ; \)
\( c. \) \(\{\text{rem}\} \rightarrow R_1 ; \)
\( \{\text{quotient}\} \rightarrow R_1+1 \)

* SLA  \( R_1, D_2(\text{B}_2) \)

\( a. \) \([\text{B}_2] + D_2 \rightarrow \text{temp} ; \)
\( b. \) \(\{\text{temp}\}_{26-31} \leftarrow 0 \)
\( \{R_1\}_{1-31} \)

* SRA  \( R_1, D_2(\text{B}_2) \)

\( a. \) \([\text{B}_2] + D_2 \rightarrow \text{temp} ; \)
\( b. \) \(\{\text{temp}\}_{26-31} \rightarrow 0 \)
\( \{R_1\}_{1-31} \)
* SLDAR₁, D₂(B₂)
  a. check that R₁ is even [assembler does this]
  b. \([B₂] + D₂ \rightarrow \text{temp} \uparrow\)
  c. \(\{\text{temp}\}_{26-31} \leftarrow 0\)
  \(\{R₁, R₁+1\}_{1-63}\)

* SRDR₁, D₂(B₂)
  a. check that R₁ is even [assembler does this]
  b. \([B₂] + D₂ \rightarrow \text{temp} \uparrow\)
  c. \(\{\text{temp}\}_{26-31} \rightarrow \{R₁\}\)
  \(\{R₁, R₁+1\}_{1-63}\)

ST  R₁, D₂(X₂, B₂)
  \((R₁) \rightarrow [B₂] + [X₂] + D₂\)

STH R₁, D₂(X₂, B₂)
  \(\{(R₁)\}_{16-31} \rightarrow [B₂] + [X₂] + D₂\)

STM R₁, R₃, D₂(B₂)
  a. \([B₂] + D₂ \rightarrow \text{temp} \uparrow\)
  b. determine n such that \(R₁ +₁₆(n) = R₃ \uparrow\)
  c. \((R₁) \rightarrow (\text{temp})\uparrow\)
     \((R₁ +₁₆ 1) \rightarrow (\text{temp}) + 4 \uparrow\)
     \((R₁ +₁₆ 2) \rightarrow (\text{temp}) + 8 \uparrow\)
     \(\vdots\)
     \((R₁ +₁₆(n)) \rightarrow (\text{temp}) + 4n(n)\)

* CR R₁, R₂
  if \((R₁) = (R₂)\) then \(0 \rightarrow \text{cc}\)
  else if \((R₁) < (R₂)\) then \(1 \rightarrow \text{cc}\)
  else \(2 \rightarrow \text{cc}\)
* C

\( R_1, D_z(x_2, B_z) \)

a. \([B_z] + [x_2] + D_z \rightarrow \text{temp} ;\)

b. if \((R) = (\text{temp}) \) then 0 \(\rightarrow\) CC
else if \((R) < (\text{temp}) \) then 1 \(\rightarrow\) CC
else 2 \(\rightarrow\) CC

* CH

\( R_1, D_z(x_2, B_z) \)

a. \([B_z] + [x_2] + D_z \rightarrow \text{temp} ;\)

b. if \((R) = (\text{temp}) \) then 0 \(\rightarrow\) CC
else if \((R) < (\text{temp}) \) then 1 \(\rightarrow\) CC
else 2 \(\rightarrow\) CC

CVB

\( R_1, D_z(x_2, B_z) \)

a. \(\{([B_z] + [x_2] + D_z)\}_{0-63} \overset{\text{convert 10-2}}{\rightarrow} \text{tem} \)

b. \(\{\text{temp}\}_{32-63} \rightarrow R_1 \)

CVD

\( R_1, D_z(x_2, B_z) \)

\( (R) \overset{\text{convert 2-10}}{\rightarrow} \{([B_z] + [x_2] + D_z)\}_{0-6} \)
**Floating Point Instructions**

**NB**: Operand register specifications refer to floating point registers and should be 0, 1, 4, or 6.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LER</td>
<td>(R_1, R_2)</td>
<td>((R_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>LDR</td>
<td>(R_1, R_2)</td>
<td>((R_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>LE</td>
<td>(R_1, D_2(\frac{B_2}{X_2}B_2))</td>
<td>(([B_2]+[X_2]+D_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>LD</td>
<td>(R_1, D_2(\frac{B_2}{X_2}B_2))</td>
<td>(([B_2]+[X_2]+D_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>LTER</td>
<td>(R_1, R_2)</td>
<td>(a. (R_2) \rightarrow R_1);</td>
</tr>
<tr>
<td>LTDR</td>
<td>(R_1, R_2)</td>
<td>(b. \text{if } (R_1) = 0 \text{ then } 0 \rightarrow \text{CC} );</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{else if } (R_1) &lt; 0 \text{ then } 1 \rightarrow \text{CC} );</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{else } 2 \rightarrow \text{CC} )</td>
</tr>
<tr>
<td>LCEER</td>
<td>(R_1, R_2)</td>
<td>((R_2)' \rightarrow R_1)</td>
</tr>
<tr>
<td>LCEDR</td>
<td>(R_1, R_2)</td>
<td>((R_2))</td>
</tr>
<tr>
<td>LPER</td>
<td>(R_1, R_2)</td>
<td>(</td>
</tr>
<tr>
<td>LPDR</td>
<td>(R_1, R_2)</td>
<td>(</td>
</tr>
<tr>
<td>LNER</td>
<td>(R_1, R_2)</td>
<td>(</td>
</tr>
<tr>
<td>LNDR</td>
<td>(R_1, R_2)</td>
<td>(</td>
</tr>
<tr>
<td>AER</td>
<td>(R_1, R_2)</td>
<td>(&lt;(R_1) + (R_2)&gt; \rightarrow R_1)</td>
</tr>
<tr>
<td>ADR</td>
<td>(R_1, R_2)</td>
<td>(&lt;(R_1) + (R_2)&gt; \rightarrow R_1)</td>
</tr>
<tr>
<td>AE</td>
<td>(R_1, D_2(\frac{B_2}{X_2}B_2))</td>
<td>(&lt;(R_1) + ([B_2]+[X_2]+D_2)&gt; \rightarrow R_1)</td>
</tr>
<tr>
<td>AD</td>
<td>(R_1, D_2(\frac{B_2}{X_2}B_2))</td>
<td>(&lt;(R_1) + ([B_2]+[X_2]+D_2)&gt; \rightarrow R_1)</td>
</tr>
<tr>
<td>AUR</td>
<td>(R_1, R_2)</td>
<td>((R_1) + (R_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>AWR</td>
<td>(R_1, R_2)</td>
<td>((R_1) + (R_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>Au</td>
<td>(R_1, D_2(\frac{B_2}{X_2}B_2))</td>
<td>((R_1) + ([B_2]+[X_2]+D_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>Aw</td>
<td>(R_1, D_2(\frac{B_2}{X_2}B_2))</td>
<td>((R_1) + ([B_2]+[X_2]+D_2) \rightarrow R_1)</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>SER</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>SDR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>SE</td>
<td>$R_1, D_2(X_1, B_1)$</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>$R_1, D_2(X_1, B_2)$</td>
<td></td>
</tr>
<tr>
<td>SUR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>SWR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td>$R_1, D_2(X_2, B_1)$</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>$R_1, D_2(X_2, B_2)$</td>
<td></td>
</tr>
<tr>
<td>HER</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>MDR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>ME</td>
<td>$R_1, D_2(X_1, B_2)$</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>$R_1, D_2(X_2, B_2)$</td>
<td></td>
</tr>
<tr>
<td>DER</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>DDR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>$R_1, D_2(X_2, B_2)$</td>
<td></td>
</tr>
<tr>
<td>DD</td>
<td>$R_1, D_2(X_2, B_2)$</td>
<td></td>
</tr>
<tr>
<td>HER</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>HDR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>STE</td>
<td>$R_1, D_2(X_1, B_2)$</td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td>$R_1, D_2(X_2, B_2)$</td>
<td></td>
</tr>
<tr>
<td>CER</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
<tr>
<td>CDR</td>
<td>$R_1, R_2$</td>
<td></td>
</tr>
</tbody>
</table>

For the operation $\langle R_1 \rangle - \langle R_2 \rangle \rightarrow R_1$, the following conditions apply:

- \( \langle R_1 \rangle = \langle R_2 \rangle \) then $O \rightarrow CC$.
- \( \langle R_1 \rangle < \langle R_2 \rangle \) then $4 \rightarrow CC$.
- Otherwise, $2 \rightarrow CC$.
* CE  \( R_1, D_2(x_2, B_2) \)  
* CD  \( R_1, D_2(x_2, B_2) \)

\[ \begin{align*}
  a. & \quad ([B_2] + [X_2] + D_2) \rightarrow \text{temp}; \\
  b. & \quad \text{if } (R_i) = (\text{temp}) \quad \text{then } 0 \rightarrow \text{CC} \\
  & \quad \text{else if } (R_i) < (\text{temp}) \quad \text{then } 1 \rightarrow \text{CC} \\
  & \quad \text{else } 2 \rightarrow \text{CC}
\end{align*} \]
PROGRAM EXCERPT 1:

```
COMPUTE ((A+B)×(C+D))/(A-D) WHERE A, B, C, AND D ARE FULL WORD FIXED POINT NUMBERS STORED IN CONSECUTIVE FULL WORDS IN STORAGE AND THE ADDRESS OF A IS IN GPR 15.
```

```
J 3,0,15
A 3,1,15
L 2,6,15
A 2,12,15
MR 2,2
L 10,0,15
S 10,12,15
DR 2,10
ST 3,16,15
```

```
A TO GPR 3
ADD B TO GPR 3; GPR 3 = A+B
C TO GPR 2
ADD D TO GPR 2; GPR 2 = C+D
(A+B)×(C+D) TO GPRS 2 & 3
A TO GPR 10
SUBTRACT D FROM GPR 10; GPR 10 = A-D
RESULT TO GPR 3
STORE RESULT AFTER D IN STORAGE
```

PROGRAM EXCERPT 2:

```
SAME AS PROGRAM EXCERPT 1 EXCEPT THAT A, B, C, AND D ARE NOW FULL WORD FLOATING POINT NUMBERS.
```

```
LE 0,0,15
AE 0,4,15
LE 6,8,15
AE 6,12,15
MR 0,6
LE 6,0,15
SE 6,12,15
DER 0,6
ST 0,16,15
```

```
A TO FPR 0
ADD B TO FPR 0; FPR 0 = A+B
C TO FPR 6
ADD D TO FPR 6; FPR 6 = C+D
FPR 0 = (A+B)×(C+D)
A TO FPR 6
SUBTRACT D FROM FPR 6; FPR 6 = C-D
RESULT TO FPR 0
STORE RESULT AFTER D IN STORAGE
```

NOTE: IN ABOVE TWO EXCERPTS, FIELDS B AND X ALWAYS REFER TO GPRS SINCE ADDRESS COMPUTATION IS DONE IN GPRS FOR ALL INSTRUCTIONS REQUIRING ADDRESSES IN STORAGE. THE R FIELDS IN FIXED POINT INSTRUCTIONS SPECIFY GPRS, SINCE FIXED POINT COMPUTATION IS DONE THERE, BUT THE R FIELDS IN FLOATING POINT INSTRUCTIONS REFER TO FPRS, SINCE FLOATING POINT COMPUTATION IS DONE THERE. IN EITHER CASE, OPERAND LENGTH IS INHERENT WITH THE SPECIFIC MACHINE INSTRUCTION USED.
BRANCHING INSTRUCTIONS:

**BCR** $M, R_z$

1. **if** $cc = 002$ **then** $10002 \rightarrow cond$
2. **else if** $cc = 012$ **then** $01002 \rightarrow cond$
3. **else if** $cc = 102$ **then** $00102 \rightarrow cond$
4. **else** $00012 \rightarrow cond$

**if** $M \land (\text{cond}) \neq 0$ and $R_z \neq 0$

- **then** $(R_z)_{31} \rightarrow IAR$

**BC** $M, D_{z}(X_z, B_z)$

1. calculate cond as above
2. **if** $M \land (\text{cond}) \neq 0$
   - **then** $(B_z) + (X_z + D_z)_{31} \rightarrow IAR$

**BALR** $R_i, R_z$

1. $(R_z)_{31} \rightarrow temp$
2. $(PSW)_{32-63} \rightarrow R_i$
3. **if** $R_z \neq 0$ **then** $(temp) \rightarrow IAR$

**BAL** $R_i, D_{z}(X_z, B_z)$

1. $(B_z) + (X_z + D_z) \rightarrow temp$
2. $(PSW)_{32-63} \rightarrow R_i$
3. $(temp) \rightarrow IAR$

**EX** $R_i, D_{z}(X_z, B_z)$

1. $(B_z) + (X_z + D_z) \rightarrow \text{inst}$
   - note: the instruction (variable-length) is fetched from the effective address
2. **if** $R_i \neq 0$
   - **then** $(\text{inst})_{8-15} \lor (R_i)_{24-31} \rightarrow \{\text{inst}\}_{8-15}$
3. **c. execute inst**

**BCTR** $R_i, R_z$

1. $(R_z)_{8-31} \rightarrow temp$
2. $(R_i) - 1 \rightarrow R_i$
3. **if** $(R_i) \neq 0$ and $R_z \neq 0$
   - **then** $(temp) \rightarrow IAR$
III.2

BCT \( R_1, R_2(x, R_0) \)

a. \([B_2]+[X_2]+D_2 \rightarrow \text{temp} \)
b. \((R_1)-1 \rightarrow R_1 \)
c. if \((R_0) \neq 0\)
   then \((\text{temp}) \rightarrow \text{IAR} \)

BXH \( R_1, R_2, D_0(B_0) \)

a. \(R_0/2*2+1 \rightarrow N \)
b. \([B_0]+D_0 \rightarrow \text{temp} \)
c. \(N \rightarrow \text{comp} \)
d. \((R_0)+(R_0) \rightarrow R_1 \)
e. if \((R_0)=\text{comp}\)
   then \((\text{temp}) \rightarrow \text{IAR} \)

BXLE \( R_1, R_2, D_0(B_0) \)

a. \(R_0/2*2+1 \rightarrow N \)
b. \([B_0]+D_0 \rightarrow \text{temp} \)
c. \(N \rightarrow \text{comp} \)
d. \((R_0)+(R_0) \rightarrow R_1 \)
e. if \((R_0) \leq \text{comp} \)
   then \((\text{temp}) \rightarrow \text{IAR} \)
PROGRAM EXCERPT 3:

COMPUTE IN GPR 6 THE SUM OF THE 100 CONTIGUOUS FULL WORD FIXED POINT NUMBERS STORED BEGINNING AT THE ADDRESS IN GPR 9. LOOP CONTROLLED BY A BCTR.

```
LA       10,99
L        6,0(,9)
BALR     15,0
A        6,4(,9)
LA       9,4(,9)
BCTR     10,15
```

PROGRAM EXCERPT 4:

SAME AS EXCERPT 3, EXCEPT LOOP CONTROLLED BY A BXLE.

```
LR       2,9
SR       6,6
LA       4,4
LA       5,396(,9)
BALR     15,0
A        6,0(,2)
BXLE     2,4,0(15)
```

PROGRAM EXCERPT 5:

SAME AS EXCERPT 3, EXCEPT LOOP CONTROLLED BY A BXH.

```
SR       2,2
LA       5,396
LA       4,4
L        6,0(,9)
BALR     15,0
BXH      2,4,10(15)
A        6,0(2,9)
BCHR     15,15
```
PROGRAM:

a. macro definitions

b. text with references to:
   1. subroutines
      a. in a library
      b. in another assembly
      c. already in core
         i. in MTS
         ii. in UMMPS
   2. macros

MACRO-PROCESSOR (part of ASSEMBLER)

expand

MACRO Libraries
*SYSMAC
*OSMAC and *1
private

MACRO-PROCESSOR

PROGRAM:

text with references to subroutines:

a. in a library
b. in another assembly
c. already in core
   i. in MTS
   ii. in UMMPS

(rest of) ASSEMBLER

process symbolic names
translate

text of macro's

*MACGEN
ASSEMBLER SOURCE:
  1. SYMBOLIC MACHINE INSTRUCTIONS
  2. ASSEMBLER INSTRUCTIONS
  3. MACRO DEFINITIONS AND/OR INSTRUCTIONS

STATEMENT FORMAT FOR THE ASSEMBLER (DEFAULT):
  COLUMNS 1-71  STATEMENT
  COLUMN 72  CONTINUATION INDICATOR
  COLUMN 16  CONTINUATION COLUMN

NAME FIELD:  BLANK OR CONTAINS SYMBOL STARTING IN COLUMN 1;
             TERMINATED BY FIRST BLANK
OPERATION FIELD:  MANDATORY; 1-8 CHARACTERS; TERMINATED BY FIRST BLANK;
                   CANNOT START IN COLUMN 1
OPERAND FIELD:  CONTAINS OPERAND EXPRESSIONS IF NEEDED; TERMINATED BY
                 FIRST BLANK
COMMENT FIELD:  THROUGH COLUMN 71; MAY CONTAIN ANY CHARACTERS; IF COLUMN 1
                CONTAINS AN *, WHOLE STATEMENT IS TREATED AS COMMENT
ID FIELD:  COLUMNS 72-80; FOR IDENTIFICATION AND SEQUENCING

SYMBOLS:
  ORDINARY:  1-8 CHARACTERS, FIRST IS ALPHABETIC (A-Z,$,#,@), REST ARE
             ALPHABETIC OR DIGITS (0-9); RELOCATABLE OR ABSOLUTE;
             ATTRIBUTES INCLUDE VALUE AND LENGTH
  VARIABLE:  FIRST CHARACTER IS &, FOLLOWED BY 1-7 CHARACTER ORDINARY
             SYMBOL; PRIMARILY USED IN MACROS
  SEQUENCE:  FIRST CHARACTER IS ., FOLLOWED BY 1-7 CHARACTER ORDINARY
             SYMBOL; USED IN CONDITIONAL ASSEMBLY
OPERAND EXPRESSION:
EVALUATED AT ASSEMBLY TIME
RESULT IS 2\text{h} BIT ADDRESS, 4 BIT LENGTH OR REGISTER SPECIFICATION, 8 BIT LENGTH
OR MASK SPECIFICATION, OR 12 BIT DISPLACEMENT
EVALUATED WITH 32 BIT SIGNED ARITHMETIC AND TRUNCATED TO FIELD WIDTH
COMPOSED OF SINGLE TERM OR ARITHMETIC COMBINATION OF TERMS
OPERATORS: + - * / ( )
INTEGER DIVISION; NO MORE THAN 5 LEVELS OF PARENS
TERM: SYMBOL, SELF-DEFINING TERM, LOCATION COUNTER REFERENCE (*),
LITERAL, SYMBOL LENGTH ATTRIBUTE (L\text{symbol}), OR OTHER SYMBOL
ATTRIBUTE REFERENCE.
ABSOLUTE OR RELOCATABLE VALUE

SELF-DEFINING TERMS:
ABSOLUTE TERMS; LENGTH ATTRIBUTE = 1; MAXIMUM VALUE = 2^{24} - 1 = 16,777,215;
ASSEMBLED INTO INSTRUCTION RIGHT-JUSTIFIED IN FIELD
TYPES:
\begin{itemize}
  \item DECIMAL \hspace{1cm} \text{UNSIGNED DECIMAL INTEGER, 1-8 DIGITS IN LENGTH}
  \item HEXADECIMAL \hspace{1cm} X^{'n'}, n = 1-6 \text{ HEXADECIMAL DIGITS (0-9, A-F)}
  \item BINARY \hspace{1cm} B^{'n'}, n = 1-2\text{h} \text{ BINARY DIGITS (0,1)}
  \item CHARACTER \hspace{1cm} C^{'s'}, s = 1-3 \text{ CHARACTERS}
\end{itemize}

LITERALS:
SYMBOLIC DATA (OPERAND) REFERENCES; RELOCATABLE TERMS; ASSEMBLER COLLECTS
LITERALS AND STORES THEM IN ONE OR MORE LITERAL POOLS; ADDRESS OF LITERAL
IS ASSEMBLED INTO INSTRUCTION; LITERAL MAY BE USED WHERE ADDRESS IS EXPECTED,
UNLESS DATA MOVEMENT IS INTO ADDRESS.
FORMATS: =\text{key}'v' \hspace{1cm} \text{NON-ADDRESS LITERALS; \text{v} = 1 OR MORE CONSTANT VALUES}
=\text{key}(a) \hspace{1cm} \text{ADDRESS LITERALS; \text{a} = 1 OR MORE EXPRESSIONS}
IN EITHER CASE, \text{key} IDENTIFIES THE SPECIFIC LITERAL TYPE
SAMPLE TYPES, DEFAULT BYTE LENGTHS, AND DEFAULT Alignments:
\begin{itemize}
  \item A ADDRESS \hspace{1cm} 4 \text{ BYTES} \hspace{1cm} FW
  \item V ADDRESS \hspace{1cm} 4 \text{ BYTES} \hspace{1cm} FW
  \item H FIXED POINT \hspace{1cm} 2 \text{ BYTES} \hspace{1cm} HW
  \item F FIXED POINT \hspace{1cm} 4 \text{ BYTES} \hspace{1cm} FW
  \item E FLOATING POINT \hspace{1cm} 4 \text{ BYTES} \hspace{1cm} FW
  \item D FLOATING POINT \hspace{1cm} 8 \text{ BYTES} \hspace{1cm} DW
  \item P PACKED DECIMAL \hspace{1cm} VARIABLE \hspace{1cm} BYTE
  \item Z ZONED DECIMAL \hspace{1cm} VARIABLE \hspace{1cm} BYTE
  \item C CHARACTER \hspace{1cm} VARIABLE \hspace{1cm} BYTE
  \item X HEXADECIMAL \hspace{1cm} VARIABLE \hspace{1cm} BYTE
  \item B BINARY \hspace{1cm} VARIABLE \hspace{1cm} BYTE
\end{itemize}
SIMPLIFIED ASSEMBLY PROCESS (IGNORING MACROS AND ERRORS):
PASS 1: DEFINE SYMBOLS, COLLECT LITERALS, COMPUTE CSECT LENGTHS AND INITIAL ADDRESSES (DOUBLE WORD ALIGNED)
PASS 2: PRODUCE OBJECT CODE ORDERED BY CSECTS

CONCEPTUAL PASS 1 FLOW:

START

\[ \downarrow \]

INITIALIZE

0 \[ \rightarrow \] LOC CTR

\[ \downarrow \]

GET NEXT STATEMENT

\[ \downarrow \]

MACHINE INSTRUCTIONS

\[ \downarrow \]

ALIGN LOC CTR TO HALF WORD
DEFINE NAME FIELD SYMBOL
VALUE = LOC CTR
LENGTH = INST LENGTH
COLLECT LITERAL IF THERE
ADD INST LENGTH TO LOC CTR

\[ \downarrow \]

OUTPUT INFORMATION FOR PASS 2

\[ \downarrow \]

ASSEMBLER INSTRUCTIONS

\[ \downarrow \]

END CLEAN UP & GO TO PASS 2
EQU DEFINE NAME FIELD SYMBOL
\{ DC ALIGN LOCATION COUNTER, THEN \}
\{ DS UPDATE FOR NEEDED BYTES \}
CSECT SAVE CURRENT LOC CTR & GET
DSECT NEW ONE
\{ USING UPDATE USING INFORMATION \}
\{ DROP \}
LTORG ESTABLISH LITERAL POOL
CNOP ADJUST LOC CTR TO SPECIFIED ALIGNMENT
\{ ENTRY IDENTIFY ESD ENTRIES \}

LOC CTR ALWAYS CONTAINS ADDRESS OF NEXT BYTE OF CSECT TO BE USED.
SOME ASSEMBLER INSTRUCTIONS DEFINE NAME FIELD SYMBOLS ALSO (CSECT, DSECT,
DC, DS, LTORG, AND SO ON)
* USED AS A TERM HAS VALUE ATTRIBUTE = LOC CTR AND LENGTH ATTRIBUTE = 1
NOTE: THE FLOW SHOWN FOR PASS 1 AND PASS 2 ARE CONCEPTUAL ONLY; ACTUAL ASSEMBLER PROCESSING IS CONSIDERABLY MORE COMPLEX AND DIFFERS GREATLY FROM THAT SHOWN.
ADDRESSING:

EXPLICIT ADDRESSING:
USER SUPPLIES BASE, INDEX, AND DISPLACEMENT FOR ADDRESS FORMATS:

\[
\begin{align*}
RX & : D(X, B); D(B); D \\
RS & : D(B); D \\
SI & : D(B); D \\
SS & : D(L, B); D(, B); D \\
\end{align*}
\]

\[D, B, X, L \text{ ARE ABSOLUTE EXPRESSIONS}\]

IMPLIED LENGTH = L'D

IMPLICIT ADDRESSING:
USER SUPPLIES RELOCATABLE EXPRESSION -- ASSEMBLER CONVERTS IT TO THE PROPER BASE AND DISPLACEMENT AND INSERTS THEM INTO THE INSTRUCTION.

FOR THIS TO WORK, MUST:

1. AT ASSEMBLY TIME, INFORM ASSEMBLER WHICH GPRs MAY BE USED AS BASE REGISTERS AND SPECIFY WHAT VALUES THESE REGISTERS WILL CONTAIN AT EXECUTION TIME;

2. AT EXECUTION TIME, MUST ENSURE THAT PROPER VALUES ARE IN BASE REGISTERS AS PROMISED DURING ASSEMBLY.

EXAMPLE USINGS:

1. THESE \(l_i\) SEQUENCES ARE EQUIVALENT:

\[
\begin{align*}
\text{BALR } & 5,0 \quad S \quad \text{BALR } 5,0 \quad \text{USING } S+2,5 \quad \text{USING } S,5 \\
\text{USING } & *,5 \\
\text{IM} & 2,1, \text{ADDR} \quad \text{USING HERE,2,3,4} \\
\cdot & \\
\cdot & \\
\text{ADDR} & A(\text{HERE,HERE+4096,HERE+6192}) \\
\end{align*}
\]

2. \(\text{IM}\) \(10,11,=A(\text{BASE+100,BASE+4196})\)

3. \(\text{LA } 5, \text{SY}\)

4. \(\text{USING SY,5}\)

NOTE: NO RELOCATABLE EXPRESSION MAY BE USED IN A PROGRAM (INCLUDING * AS A TERM AND LITERALS) UNLESS THERE IS A USING REGISTER WHOSE VALUE IS WITHIN 4095 OF THE RELOCATABLE EXPRESSION VALUE.

FORMATS:

\[
\begin{align*}
RX & : S(X); S \\
RS & : S \\
SI & : X, L = \text{ABSOLUTE EXPRESSIONS} \\
SS & : S(L); S \\
\end{align*}
\]

IMPLIED LENGTH = L'S
MACHINE: INSTRUCTION PROCESSING:

NAME FIELD SYMBOL:
VALUE IS ADDRESS OF LEFT-MOST BYTE OF INSTRUCTION
LENGTH IS INSTRUCTION LENGTH
RR 2
RX,RS,SI 4
SS 6

ASSEMBLER CHECKING DURING PROCESSING:
LEGAL OPERATION MNEMONICS ONLY
HALF WORD ALIGNED (FILLS WITH X'OO')
OPERAND ALIGNMENTS CHECKED, IF POSSIBLE
REGISTER SPECIFICATIONS CHECKED
FPRS 0,2,4,6 ONLY
GPRS 0 THROUGH 15 ONLY, EVEN-ODD PAIRS CHECKED WHERE NEEDED

EXTENDED MNEMONICS FOR BC AND BCR:

<table>
<thead>
<tr>
<th>MASK</th>
<th>CONDITION</th>
<th>MNEMONICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>NOP, NOPR</td>
</tr>
<tr>
<td>1</td>
<td>OF, 1s</td>
<td>BC, BOR</td>
</tr>
<tr>
<td>2</td>
<td>&gt;</td>
<td>BH, BHR, BP, BPR</td>
</tr>
<tr>
<td>4</td>
<td>&lt;</td>
<td>BL, BLR, BM, BMR</td>
</tr>
<tr>
<td>7</td>
<td>≠</td>
<td>BNE, BNER, BNZ, BNR</td>
</tr>
<tr>
<td>8</td>
<td>=</td>
<td>BE, BER, BZ, BZR</td>
</tr>
<tr>
<td>11</td>
<td>≥</td>
<td>BNI, BNL, BM, BMR</td>
</tr>
<tr>
<td>13</td>
<td>≤</td>
<td>BNH, BNHR, BNF, BNR</td>
</tr>
<tr>
<td>14</td>
<td>NOT 1s</td>
<td>BNO</td>
</tr>
<tr>
<td>15</td>
<td>ALL</td>
<td>B, BR</td>
</tr>
</tbody>
</table>

PROGRAM EXCERPT 6:

REPEAT OF PROGRAM EXCERPT 1 USING IMPLIED ADDRESSING. ASSUME A, B, C, AND D
NAME THE FOUR FIXED POINT NUMBER WHICH ARE IN CONTIGUOUS LOCATIONS.

.
.
RAIR 12,0
USING *,12

ESTABLISH ADDRESSABILITY -- ASSUME
IT COVERS WHOLE PROGRAM AND DATA

.
.
L 3,A
A 3,B
L 2,C
A 2,D
MR 2,2
L 10,A
S 10,D
DR 2,10
ST 3,D+4
.
.
A TO GPR 3
A+B TO GPR 3
C TO GPR 2
C+D TO GPR 2
PRODUCT TO GPRS 2 & 3
A TO GPR 10
A-D TO GPR 10
RESULT TO GPR 3
STORE RESULT -- ADDRESS D+4 COULD ALSO BE
WRITTEN AS C+8, B+12, OR A+16
PROGRAM EXCEPT 7:
REPEAT OF PROGRAM EXCEPT 4 USING IMPLIED ADDRESSING. WE NOW ASSUME THE 100 FIXED POINT NUMBERS ARE STORED CONTIGUOUSLY IN A VECTOR NAMED Vect AND THE SUM IS TO BE PLACED IN A FULL WORD NAMED SUM.

BALR 10,0  
ESTABLISH ADDRESSABILITY
USING *\*10
L 2,=A(VECT)  
FIRST ADDRESS TO GPR 2
SR 6,6  
0 TO GPR 6
LA L,4  
4 TO GPR L
LA 5,VECT+396  
LAST ADDRESS TO GPR 5
A 6,0(2)  
SUM TO GPR 6
EXE 2,=L-*L  
LOOP ON COUNT OF NUMBERS
ST 6,SUM  
STORE SUM

PROGRAM EXCEPT 8:
SAME AS PROGRAM EXCEPT 7 EXCEPT WE NOW ASSUME THE COUNT OF THE ENTRIES IN Vect IS IN A HALF WORD NAMED COUNT; THE COUNT IS GREATER THAN OR EQUAL TO ZERO.

BALR 10,0  
ESTABLISH ADDRESSABILITY
USING *\*10
LH 6,COUNT  
COUNT TO GPR 6
LTR 7,6  
COUNT TO GPR 7 AND SET CC
BE STORE  
BRANCH ON ZERO COUNT - GPRs 6,7 = 0
SR L,1  
ZERO TO GPR 1
L 6,VECT  
FIRST NUMBER TO GPR 6
B STORE-L  
ENTER THE SUM LOOP
LOOP LA L,1(L,1)  
UPDATE THE INDEX
A 6,VECT(1)  
ADD NEXT NUMBER TO SUM
BCT 7,LOOP  
LOOP ON COUNT OF NUMBERS
STORE ST 6,SUM  
STORE SUM
DEFINE CONSTANT (DC) AND DEFINE STORAGE (DS) ASSEMBLER INSTRUCTIONS:

<table>
<thead>
<tr>
<th>NAME FIELD SYMBOL PROCESSING:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE = ADDRESS OF FIRST CONSTANT OR FIRST STORAGE AREA, AFTER ALIGNMENT</td>
</tr>
<tr>
<td>LENGTH = LENGTH OF FIRST CONSTANT OR FIRST STORAGE AREA</td>
</tr>
</tbody>
</table>

EACH OPERAND IS COMPOSED OF UP TO FOUR SUBFIELDS:
1. DUPLICATION FACTOR -- OPTIONAL SUBFIELD
2. KEY SUBFIELD -- REQUIRED SUBFIELD
3. MODIFIERS -- OPTIONAL SUBFIELD
4. NOMINAL VALUES -- REQUIRED SUBFIELD FOR DC, OPTIONAL FOR DS

NOTE: IN FOLLOWING, dae STANDS FOR "UNSIGNED DECIMAL SELF DEFINING TERM OR PREDEFINED ABSOLUTE EXPRESSION ENCLOSED IN PARENTHESES, WITH VALUE GREATER THAN OR EQUAL TO ZERO".

DUPLICATION FACTOR: SPECIFIED AS dae; DEFaulTS TO 1 IF NOT GIVEN; A VALUE OF ZERO FORCES ALIGNMENT BUT RESERVES NO STORAGE

KEY SUBFIELD: DEFINES TYPE OF CONSTANT TO BE HANDLED AND SELECTS THE IMPLICIT (DEFAULT) LENGTH AND ALIGNMENT. THE KEYS, TYPES, DEFAULT ALIGNMENTS, DEFAULT LENGTHS, MAXIMUM LENGTHS, AND CONSTANTS PER OPERAND ARE AS FOLLOWS:

| C | CHARACTER | BYTE | AS NEEDED | 256 | 1 |
| X | HEXADECIMAL | BYTE | AS NEEDED | 256 | 1 |
| B | BINARY | BYTE | AS NEEDED | 256 | 1 |
| P | PACKED | BYTE | AS NEEDED | 16 | MULTIPLE |
| Z | ZONED | BYTE | AS NEEDED | 16 | MULTIPLE |
| F | FIXED PT | FW | 4 | 8 | MULTIPLE |
| H | FIXED PT | HW | 2 | 8 | MULTIPLE |
| E | FLOAT PT | FW | 4 | 8 | MULTIPLE |
| D | FLOAT PT | DW | 8 | 8 | MULTIPLE |
| L | FLOAT PT | DW | 16 | 16 | MULTIPLE |
| A | ADDRESS | FW | 4 | h | MULTIPLE EXPRESSIONS |
| V | ADDRESS | FW | 4 | h | MULTIPLE EXT SYMBOLS |
| Y | ADDRESS | HW | 2 | 2 | MULTIPLE EXPRESSIONS |
| S | BASE/DISPL | HW | 2 | 2 | MULTIPLE |
| Q | ADDRESS | FW | 4 | 4 | MULTIPLE |

MODIFIERS:
LENGTH: L,n BIT LENGTH \{ n GIVEN AS dae
Ln BYTE LENGTH IF GIVEN, OVERIDES DEFAULT LENGTH AND CANCELS ALIGNMENT

SCALE: Sn n GIVEN AS dae WITH OR WITHOUT SIGN
SCALES F,H,E,D,L TYPES OF CONSTANTS AFTER CONVERSION TO BINARY

EXPONENT: En n GIVEN AS dae WITH OR WITHOUT SIGN
SCALES F,H,E,D,L TYPES OF CONSTANTS BEFORE CONVERSION TO BINARY

NOMINAL VALUES:
FOR ADDRESS TYPES, SPECIFIED AS (E1,E2,...,En)
FOR V TYPE, EACH EL IS AN EXTERNAL RELOCATABLE SYMBOL
FOR A TYPE, EACH EL IS A RELOCATABLE OR ABSOLUTE EXPRESSION
FOR ALL OTHER TYPES, SPECIFIED AS 'C1' OR 'C1,C2,...,Cn'
TYPES C,X,B: 'C' ONLY
TYPES P,Z,F,H,E,D,L: MULTIPLE VALUES ALLOWED
* AS A TERM HAS VALUE OF LEFT MOST BYTE ADDRESS IN CONSTANT

FOR THE DS ASSEMBLER INSTRUCTION:
NOMINAL VALUES OPTIONAL SAME ALIGNMENT/LENGTH, BUT NOTHING LOADED
TYPES X,C MAY HAVE MAXIMUM LENGTH OF 65535 BYTES
DS EXAMPLES:

DS OH  }  ALIGNS TO HALF, FULL, OR DOUBLE WORD, RESPECTIVELY, BUT RESERVES
DS OF  }  NO STORAGE; LENGTHS = 2, 4, 8, RESPECTIVELY
DS OD  

DS nH  }  ALIGNS TO HALF, FULL, OR DOUBLE WORD, RESPECTIVELY, THEN RESERVES
DS nF  }  n HALF, FULL, OR DOUBLE WORDS, RESPECTIVELY; LENGTHS = 2, 4, 8,
DS nD  }  RESPECTIVELY

DS Cln  RESERVES n BYTES; LENGTH IS n; NO ALIGNMENT  }  FOR ALIGNMENT,
DS mCln  RESERVES m*n BYTES; LENGTH IS n; NO ALIGNMENT  }  MAY PRECEDE WITH
DS nC  RESERVES n BYTES; LENGTH IS l; NO ALIGNMENT;  }  DS OH, OF, OD
EXAMPLE OF FIELD OVERLAP USING DS:

<table>
<thead>
<tr>
<th>CARD</th>
<th>DS</th>
<th>OCL80</th>
<th>LENGTH = 80; NO STORAGE RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD</td>
<td>DS</td>
<td>OCL10</td>
<td>LENGTH = 10; NO STORAGE RESERVED</td>
</tr>
<tr>
<td>FIELD</td>
<td>DS</td>
<td>OCL20</td>
<td>LENGTH = 20; NO STORAGE RESERVED</td>
</tr>
<tr>
<td>MIDD</td>
<td>DS</td>
<td>OCL40</td>
<td>RESERVE 40 BYTES OF STORAGE</td>
</tr>
<tr>
<td>FIELD</td>
<td>DS</td>
<td>C120</td>
<td>LENGTH = 20; 20 BYTES OF STORAGE RESERVED</td>
</tr>
<tr>
<td>FIELD</td>
<td>DS</td>
<td>1OCL2</td>
<td>LENGTH = 2; 20 BYTES OF STORAGE RESERVED</td>
</tr>
</tbody>
</table>

FOR LITERALS:

NO MULTIPLE OPERANDS -- SINGLE OPERAND MAY HAVE MULTIPLE VALUES, IF ALLOWED
DUPLICATION AND LENGTH MUST BE UNSIGNED DECIMAL SELF DEFINING TERMS ONLY
DUPLICATION MUST BE GREATER THAN ZERO
SCALE AND EXPONENT MUST BE DECIMAL SELF DEFINING TERM ONLY (SIGN ALLOWED)
TYPES S, Q NOT LEGAL AS LITERALS
* AS A TERM HAS AS VALUE THE ADDRESS OF THE LEFT-MOST BYTE OF THE INSTRUCTION
IN WHICH THE LITERAL OCCURS
PROGRAM SECTIONING AND CONTROL SECTIONS (CSECTS):

CSECT:

BASIC ASSEMBLY/LINKEDIT/LOAD UNIT IN SYSTEM
SOURCE/OBJECT MODULE = 1 OR MORE CSECTS
LOADER PROCESSING:
CSECT LOADED WHEREVER THERE IS SUFFICIENT ROOM
CSECT LOADED IN CONTIGUOUS LOCATIONS BEGINNING ON DW BOUNDARY
ALL CSECT NAMES APPEAR IN LOAD MAP

CSECT DEFINITION DURING ASSEMBLY:

[sym] START org = BLANK OR SELF DEFINING TERM
NAMES FIRST OR ONLY CSECT IN ASSEMBLY; CSECT STARTS AT org OR ZERO;
sym DEFINED (LENGTH = 1) AND MADE EXTERNAL

[sym] CSECT
RESETS LOCATION COUNTER TO THAT OF CSECT sym; sym DEFINED (LENGTH = 1)
AND MADE EXTERNAL

sym DSECT
RESETS LOCATION COUNTER TO THAT OF DSECT sym; EACH DSECT MUST BE NAMED;
sym IS VALID RELOCATABLE INTERNAL SYMBOL WITH LENGTH OF 1; EACH DSECT I
ASSEMBLED BUT IS NOT PART OF OBJECT MODULE; DSECTS USED TO ESTABLISH A
MNEMONIC DESCRIPTION FOR USING PURPOSES OF A STORAGE AREA

FIRST CSECT IS ASSEMBLED AT org OR AT ZERO -- REST ARE ASSEMBLED ON DW BOUNDARIES
IN ORDER OF OCCURRENCE DURING ASSEMBLY; LITERALS NOT CAPTURED BY LTORG'S
GO TO LITERAL POOL AT END OF FIRST CSECT; CSECT NAMES ARE EXTERNAL --
DSECT NAMES ARE INTERNAL

FOR CSECTS ASSEMBLED TOGETHER:

ALL SYMBOLS ARE DEFINED IN ALL CSECTS;
USING IN CSECT COVERS ONLY THAT CSECT -- BUT USING RANGE DURING ASSEMBLY IS
INDEPENDENT OF CSECT BOUNDARIES;
NO EXTERNAL SYMBOLS NEEDED FOR COMMUNICATION;
IN EACH CSECT, ADDRESSABILITY MUST BE ESTABLISHED SEPARATELY FOR EACH OTHER
CSECT REFERENCED SYMBOLICALLY;
ADDRESSABILITY OVER LITERALS MUST BE MAINTAINED OVER CSECTS

SYMBOLIC LINKAGES -- INDEPENDENTLY ASSEMBLED CSECTS:

IN EACH CSECT ASSEMBLED, CSECT NAMES AND ENTRY NAMES (VIA ENTRY ASSEMBLER
INSTRUCTION) ARE INTERNALLY DEFINED BUT ARE MADE EXTERNALLY KNOWN--
ALL OTHER DEFINED SYMBOLS ARE PRESUMED INTERNAL ONLY

IN EACH CSECT ASSEMBLED, SYMBOLS OCCURRING IN V-TYPE ADDRESS CONSTANTS OR IN
THE EXTNR OR WXTRN ASSEMBLER INSTRUCTIONS ARE PRESUMED TO BE EXTERNALLY
DEFINED (THEY MUST NOT BE INTERNALLY DEFINED) -- ALL OTHER SYMBOLS MUST
BE INTERNALLY DEFINED

EXAMPLE LINKAGES:

1. L 15,v(SUB)
   BALR 1h,15

2. EXTRN SUB
   L =A(SUB)
   BALR 1h,15

3. L 12,v(DATA)
   USING DATAD,12

4. EXTRN DATA
   L 12=A(DATA)
   USING DATAD,12

3. AND 4, PREASSUME DESCRIPTIONS OF DATA PROVIDED DURING ASSEMBLY
   BY DSECT NAMED DATAD

5. L 12,v(DATA) \{ ASSUMING DATA FW ALIGNMENT, LOADS GPR 10
   L 10,h(12) \} WITH SECOND FW OF DATA
EXAMPLE SYMBOLIC LINKAGE:

NAME CSECT
LR 10,15
USING NAME,10
L 5,=A(D1)
USING D1,5

L 15,=V(SUB)
BALR 14,15

EXTERN S1
L 15,=A(S1)
BALR 14,15

D1 CSECT

END

S CSECT
ENTRY S1

USING *,15
S1 L 11,=V(D1)

(REFERENCES TO D1 VIA DISPLACEMENTS ON GPR 11)

BR 14
END

SUB CSECT
USING DATA,5

DATA DSECT

(SYMBOLIC DESCRIPTION OF DATA CSECT)

END
DECIMAL ARITHMETIC INSTRUCTIONS:

For the following instructions, the operands are in either packed or zoned decimal format and occupy a field which starts at the specified address and extends for the specified number of bytes.

Operands which are of unequal length are made of equal length by extending the shorter with high order zeroes.

All operands must be in packed decimal format except the operand to the PACK instruction, which must be in zoned decimal format.

All results will be in packed decimal format except the result of the UNPK instruction, which will be in zoned decimal format.

The results of all instructions always fill the entire target field, being extended with high order zeroes if necessary.

* AP

\[ D_1(L_1, B_1), D_2(L_2, B_2) \rightarrow ([B_1]+D_1) + ([B_2]+D_2) \rightarrow [B_1]+D_1 \]

* SP

\[ D_1(L_1, B_1), D_2(L_2, B_2) \rightarrow ([B_1]+D_1) - ([B_2]+D_2) \rightarrow [B_1]+D_1 \]

MP

\[ D_1(L_1, B_1), D_2(L_2, B_2) \]

if \( L_2 \leq 7 \) and \( L_2 < L_1 \)

then \( ([B_1]+D_1) \times ([B_2]+D_2) \rightarrow [B_1]+D_1 \)

UP

\[ D_1(L_1, B_1), D_2(L_2, B_2) \]

if \( L_2 \leq 7 \) and \( L_2 < L_1 \)

then \( ((B_1)+D_1) \times (B_2)+D_2) \rightarrow q+r \times 3 \)

\( q \rightarrow [B_1]+D_1 \times 3 \)

\( r \rightarrow [B_1]+D_1 \times 1-L_2 \)

end

MVO

\[ D_1(L_1, B_1), D_2(L_2, B_2) \]

\[ \min(L_1,L_2)-1 \rightarrow L_0 ; L_0 \rightarrow L_1 ; L_2-1 \rightarrow L_2 \]

\( \{([B_1]+D_1)+(L_0)\} \times 3 \rightarrow [B_1]+D_1+(L_0) \times 3 \)

\( \{([B_1]+D_1)+(L_1)\} \times 3 \rightarrow [B_1]+D_1+(L_1) \times 3 \)

\( \{([B_1]+D_1)+(L_2-1)\} \times 3 \rightarrow [B_1]+D_1+(L_2-1) \times 3 \)

\( \{([B_1]+D_1)+(L_2)\} \times 3 \rightarrow [B_1]+D_1+(L_2) \times 3 \)

\( \{([B_1]+D_1)+(L_2+1)\} \times 3 \rightarrow [B_1]+D_1+(L_2+1) \times 3 \)

\( \{([B_1]+D_1)+(L_2+2)\} \times 3 \rightarrow [B_1]+D_1+(L_2+2) \times 3 \)

\( \{([B_1]+D_1)+(L_2+3)\} \times 3 \rightarrow [B_1]+D_1+(L_2+3) \times 3 \)

where if \( L_1 < L_2 \) then

\( L_0 \) is 4-7 and \( k \cdot m \) is 0-8

and if \( L_1 > L_2 \) then \( L_0 \) is 4-7 and \( k \cdot m \) is 9-7
\[ \min (L_1, L_2) \rightarrow L_0 \]

\[ \{B_1\} + D_1 \rightarrow \{B_1\} + D_1 \]

\[ \{B_2\} + D_2 \rightarrow \{B_2\} + D_2 \]

\[ \{B_1, B_2\} + D_3 \rightarrow \{B_1, B_2\} + D_3 \]

**CP**

\( D_1(L_1, B_1), D_2(L_2, B_2) \)

\( \{B_1\} + D_1 \rightarrow V_1 \)

\( \{B_2\} + D_2 \rightarrow V_2 \)

\[ \begin{array}{l}
\text{if} \ (V_1 = V_2) \ \text{then} \ 0 \rightarrow \text{CC} \\
\text{else} \ 1 \rightarrow \text{CC} \\
\text{else} \ 2 \rightarrow \text{CC}
\end{array} \]

**PACK**

\( D_1(L_1, B_1), D_2(L_2, B_2) \)

\[ \min (2 \cdot L_1 - 1, L_2) \rightarrow L_0 \]

\[ \{B_1\} + D_1 \rightarrow \text{addr}_1 \]

\[ \{B_2\} + D_2 \rightarrow \text{addr}_2 \]

\[ \begin{array}{l}
\{\text{addr} + L_1 - 1\} \rightarrow \{\text{addr} + L_1 - 1\} \\
\{\text{addr} + L_2 - 1\} \rightarrow \{\text{addr} + L_2 - 1\} \\
\text{for} \ i = 2 \ \text{step} \ 2 \ \text{until} \ (L_2 - 1) \ \text{do} \begin{array}{l}
\{\text{addr} + L_1 - i\} \rightarrow \{\text{addr} + L_1 - i\} \\
\{\text{addr} + L_2 - i\} \rightarrow \{\text{addr} + L_2 - i\} \\
\text{end}
\end{array} \\
\frac{(L_0)}{2} \times 2 = \{L_0\}
\end{array} \]

**UNPK**

\( D_1(L_1, B_1), D_2(L_2, B_2) \)

\[ \min (2 \cdot L_2 - 1, L_1) \rightarrow L_0 \]

\[ \{B_1\} + D_1 \rightarrow \text{addr}_1 \]

\[ \{B_2\} + D_2 \rightarrow \text{addr}_2 \]

\[ \begin{array}{l}
\{\text{addr} + L_2 - 1\} \rightarrow \{\text{addr} + L_2 - 1\} \\
\{\text{addr} + L_1 - 1\} \rightarrow \{\text{addr} + L_1 - 1\} \\
\text{for} \ i = 2 \ \text{step} \ 2 \ \text{until} \ (L_1 - 1) \ \text{do} \begin{array}{l}
\{\text{addr} + L_2 - i\} \rightarrow \{\text{addr} + L_2 - i\} \\
\{\text{addr} + L_1 - i\} \rightarrow \{\text{addr} + L_1 - i\} \\
\text{end}
\end{array} \\
\frac{(L_0)}{2} \times 2 = \{L_0\}
\end{array} \]
**Logical Instructions:**

**MVI**  \( D_1(B_1), I_x \)  
\[ I_2 \rightarrow [B_1] + D_1 \]
*note: \( I_2 \) is a 16 bit field*

**MVC**  \( D_1(L,B_1), D_2(B_2) \)

a. \( L-1 \rightarrow \text{counter} \)

b. \([B_1] + D_1 \rightarrow \text{addr} \)
\([B_2] + D_2 \rightarrow \text{addr} \)

c. \{ ((\text{addr}) \times 4 \rightarrow \text{addr} + 1 \}
\{ ((\text{addr}) + 1 \times 4 \rightarrow \text{addr} + 2 \}
\{ ((\text{addr}) + 2 \times 4 \rightarrow \text{addr} + 3 \}
\{ ((\text{addr}) + 3 \times 4 \rightarrow \text{addr} + 4 \}
\{ ((\text{addr}) + 4 \times 4 \rightarrow \text{addr} + 5 \}
\{ ((\text{addr}) + 5 \times 4 \rightarrow \text{addr} + 6 \}
\{ ((\text{addr}) + 6 \times 4 \rightarrow \text{addr} + 7 \}

**MVN**  \( D_1(L,B_1), D_2(B_2) \)

a. \( L-1 \rightarrow \text{counter} \)

b. \([B_1] + D_1 \rightarrow \text{addr} \)
\([B_2] + D_2 \rightarrow \text{addr} \)

c. \{ ((\text{addr}) \times 4 \rightarrow (\text{addr}) \div 4 \}
\{ ((\text{addr}) + 1 \times 4 \rightarrow (\text{addr}) + 1 \}
\{ ((\text{addr}) + 2 \times 4 \rightarrow (\text{addr}) + 2 \}
\{ ((\text{addr}) + 3 \times 4 \rightarrow (\text{addr}) + 3 \}
\{ ((\text{addr}) + 4 \times 4 \rightarrow (\text{addr}) + 4 \}

**MVZ**  \( D_1(L,B_1), D_2(B_2) \)

a. \( L-1 \rightarrow \text{counter} \)

b. \([B_1] + D_1 \rightarrow \text{addr} \)
\([B_2] + D_2 \rightarrow \text{addr} \)

c. \{ ((\text{addr}) \times 4 \rightarrow \text{addr} \div 4 \}
\{ ((\text{addr}) + 1 \times 4 \rightarrow \text{addr} + 1 \}
\{ ((\text{addr}) + 2 \times 4 \rightarrow \text{addr} + 2 \}

**IC**  \( R_1, D_2(X_2,B_2) \)
\([B_2] + [X_2] + D_2 \rightarrow [R_1]_{24-31} \)

**STC**  \( R_1, D_2(X_2,B_2) \)
\([R_1]_{24-31} \rightarrow [B_2] + [X_2] + D_2 \)
* CLR

\[ R_1, R_2 \]

\[
\begin{align*}
\text{IF } (R_1) = (R_2) \text{ then } 0 \rightarrow \text{CC} \\
\text{else if } (R_1) < (R_2) \text{ then } 1 \rightarrow \text{CC} \\
\text{else } 2 \rightarrow \text{CC}
\end{align*}
\]

Note: comparison is over all 32 bits without any interpolation to fields within the 32-bit

* CL

\[ R_1, D_2(\text{x}, \text{B}_2) \]

CC set as above but comparison is between \((R_1)\) and \((I_1 + \text{[x]} + \text{D}_2)\)

* CLI

\[ D_1(\text{B}_1), I_2 \]

CC set as above but comparison is between \((\text{B}_1 + \text{D}_1)\) and \((I_2)\)

Note: 8 bits are compared

* LLC

\[ D_1(\text{L}, \text{B}_1), D_2(\text{B}_2) \]

CC set as above but comparison is between the two character strings of length 1, which start at \([\text{B}_1] + \text{D}_1\) and \([\text{B}_1] + \text{D}_2\)

* NR

\[ R_1, R_2 \]

\[(R_1) \land (R_2) \rightarrow R_1\]

* N

\[ R_1, D_2(\text{x}, \text{B}_2) \]

\[(R_1) \land ([\text{B}_1] + [\text{X}] + \text{D}_2) \rightarrow R_1\]

* NI

\[ D_1(\text{B}_1), I_2 \]

\([\text{B}_1] + \text{D}_1 \land I_2 \rightarrow [\text{B}_1] + \text{D}_1\]

* NC

\[ D_1(\text{L}, \text{B}_1), D_2(\text{B}_2) \]

\[
\begin{align*}
a. \text{L-1} & \rightarrow \text{counter} \\
b. [\text{B}_1] + \text{D}_1 & \rightarrow \text{addr}_1 \\
[\text{B}_1] + \text{D}_2 & \rightarrow \text{addr}_2 \\
c. ([\text{addr}_1]) \land ([\text{addr}_2]) & \rightarrow (\text{addr}) \\
([\text{addr}_1] + 1) \land ([\text{addr}_2] + 1) & \rightarrow (\text{addr}) + 1 \\
([\text{addr}] + \text{counter}) \land ([\text{addr}] + \text{counter}) & \rightarrow (\text{addr}) + \text{counter}
\end{align*}
\]

* OK

\[ R_1, R_2 \]

* D

\[ R_1, D_2(\text{x}, \text{B}_2) \]

* DL

\[ D_1(\text{B}_1), I_2 \]

* DC

\[ D_1(\text{L}, \text{B}_1), D_2(\text{B}_2) \]

Operations are the same as with AND and exclusive or except operations ∧ ∨ (inclusive OR)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XR</strong></td>
<td>$R_1, R_2$</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>$R_1, D_2(X_2, B_2)$</td>
</tr>
<tr>
<td><strong>XI</strong></td>
<td>$D_1(B_2), I_2$</td>
</tr>
<tr>
<td><strong>XC</strong></td>
<td>$D_1(L, B_1), D_2(B_2)$</td>
</tr>
<tr>
<td><strong>SLL</strong></td>
<td>$R_1, D_2(B_2)$</td>
</tr>
<tr>
<td></td>
<td>${[B_2] + D_2}_{26-31}$</td>
</tr>
<tr>
<td></td>
<td>$\leftarrow O$</td>
</tr>
<tr>
<td></td>
<td>${R_1}_{0-31}$</td>
</tr>
<tr>
<td><strong>SRL</strong></td>
<td>$R_1, D_2(B_2)$</td>
</tr>
<tr>
<td></td>
<td>${[B_2] + D_2}_{26-31}$</td>
</tr>
<tr>
<td></td>
<td>$O \rightarrow {R_1}_{0-31}$</td>
</tr>
<tr>
<td><strong>SDDL</strong></td>
<td>$R_1, D_2(B_2)$</td>
</tr>
<tr>
<td>a. check that $R_1$ is even</td>
<td></td>
</tr>
<tr>
<td>b. ${[B_2] + D_2}_{26-31}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\leftarrow O$</td>
</tr>
<tr>
<td></td>
<td>${R_1, R_1 + 1}_{0-63}$</td>
</tr>
<tr>
<td><strong>SRDL</strong></td>
<td>$R_1, D_2(B_2)$</td>
</tr>
<tr>
<td>a. check that $R_1$ is even</td>
<td></td>
</tr>
<tr>
<td>b. ${[B_2] + D_2}_{26-31}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$O \rightarrow {R_1, R_1 + 1}_{0-63}$</td>
</tr>
</tbody>
</table>

**TM**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1(B_2), I_2$</td>
</tr>
</tbody>
</table>

a. $([B_2] + D_2) \rightarrow \text{temp}$

b. If $([\text{temp}] \land I_2 = 11111111_2$ then $0 \rightarrow \text{cc}$
   else if $(\text{temp}) \land I_2 = 00000000_2$ then $3 \rightarrow \text{cc}$
   else $1 \rightarrow \text{cc}$

**TR**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1(L, B_2), D_2(B_2)$</td>
</tr>
</tbody>
</table>

a. $L-1 \rightarrow \text{counter}$
b. $[B_2] + D_1 \rightarrow \text{addr}_1$
   $[B_2] + D_2 \rightarrow \text{addr}_2$
c. $([\text{addr}_1] + ((\text{addr}_1))) \rightarrow (\text{addr}_1)$
   $([\text{addr}_1] + ((\text{addr}_1))) \rightarrow (\text{addr}_1 + 1)$
   $([\text{addr}_1] + ((\text{addr}_1))) \rightarrow (\text{addr}_1 + 1)$
   $([\text{addr}_1] + ((\text{addr}_1))) \rightarrow (\text{addr}_1 + 1)$
   $([\text{addr}_1] + ((\text{addr}_1))) \rightarrow (\text{addr}_1 + 1)$
TRT  \( D_1(L,B), D_2(B) \)

a. \( L-1 \rightarrow \text{ctr} \)
   \( 0 \rightarrow \text{index} \)
b. \( [B_1]+D_1 \rightarrow \text{addr}_1 \)
   \( [B_2]+D_2 \rightarrow \text{addr}_2 \)
c.
   \[\text{while } (\text{index}) \leq (\text{ctr}) \]
   \[\text{and } (\text{addr}_2+(\text{addr}_1+\text{index})) = 0 \]
   \( (\text{index})+1 \rightarrow \text{index} \)
   \[\text{if } (\text{index}) \leq (\text{ctr}) \]
   \[\text{then } 0 \rightarrow \text{CC} \]
   \[\text{else begin} \]
   \[\text{if } (\text{index}) = (\text{ctr}) \text{ then } 2 \rightarrow \text{CC} \]
   \[\text{else } 1 \rightarrow \text{CC} \]
   \[(\text{addr}_1)+(\text{index}) \rightarrow \{GR1\}_{B,31} \]
   \[(\text{addr}_2)+((\text{addr}_1)+(\text{index})) \rightarrow \{GR2\} \]
   \[\text{end} \]

note that \( \{GR1\}_{B,31} \) and \( \{GR2\}_{B,23} \) are unchecked

ED  \( D_1(L,B), D_2(B) \)

if interested, see POOP manual.

EDMK  \( D_1(L,B), D_2(B) \)
LOGICAL MACHINE INSTRUCTION EXAMPLES:

1. MVI  BUFF, C'1'
   MVC  BUFF-1(79),BUFF
   PUTS BLANKS IN 80 BYTES STARTING AT LOCATION BUFF

2. MVI  A, 0
   MVC  A+1(L'A-1),A
   ZEROS OUT AREA A, ASSUMING L'A IS AT MOST 256

3. CLI  BYTE,'X'F5'
   BE   EQUAL
   BRANCHES TO EQUAL IF BYTE CONTAINS A HEXADECIMAL F5

4. STM  0,10,GPRS
   CLC  GPRS(lt),VALS
   BE   CHECKS
   BRANCHES TO CHECKS IF GP RS 0 - 10 CONTAIN THE SAME VALUES AS THE 11 FULL WORDS STARTING AT VALS

5. TM   BYTE,'X'FO'
   BO   SET
   BRANCHES TO SET IF BITS 0 - 3 OF BYTE ARE ALL ONES

6. TM   BYTE,1
   BZ   UNSET
   BRANCHES TO UNSET IF BIT 7 OF BYTE IS ZERO

7. NI   BYT,'X'00'
   NI   BYT,'X'77'
   N    6,'X'00FFFF00'
   OI   BYT,'X'FF'
   OI   BYT,'X'EE'
   XI   BYT,'X'00'
   XI   BYT,'X'FF'
   XI   BYT,'X'88'
   SETS BITS 0 - 7 OF BYT TO ZERO
   ZEROS BITS 0 AND 4 OF BYT
   ZEROS FIRST BYTE OF GP RS 6
   SETS BITS 0 - 7 OF BYT TO ONE
   SETS BITS 3 AND 7 OF BYT TO ONE
   BYT UNCHANGED
   ALL 8 BITS OF BYT INVERTED
   BITS 0 AND 4 OF BYT INVERTED

8. THE TRANSLATE (TR) INSTRUCTION:
   TR  D1(L,B1),D2(B2)
   \rightarrow \{ \text{ARGUMENT} \}
   \rightarrow \{ \text{256 FUNCTION BYTES} \}
   \rightarrow \{ \text{BYTES} \}

   EACH OF THE L ARGUMENT BYTES IS PROCESSED, LEFT TO RIGHT, AS FAROWS:
   1. COMPUTE ADDRESS $A = D_2(B_2) + \text{ARGUMENT BYTE} \ (I. \ E., \ \text{USE ARGUMENT}$
      \text{BYTE AS INDEX})
   2. REPLACE ARGUMENT BYTE BY FUNCTION BYTE AT ADDRESS A

   FOR EXAMPLE, TO CONVERT EBCDIC STRING OF LENGTH L IN STR TO A STRING
   IN WHICH C'0'...C'9' BECOMES X'00'...X'09' AND ALL OTHER CHARACTERS
   BECOME X'FF', COULD USE:

   TR  STR(L),TAB
   .
   .
   TAB  DC  (C'0'..X'FF',X'0001203040506070809',X'6X'FF')
   .
   .
THE TRANSLATE AND TEST (TRT) INSTRUCTION:

\[ \text{TRT } D_1(L, B_1), D_2(B_2) \]

The argument bytes are processed, one by one, from left to right, as follows:

1. Compute address \( A = D_2(B_2) + \text{argument byte} \) (i.e., use argument byte as index)

2. If function byte at location \( A \) is zero, continue at step 1 with next argument byte

3. If function byte at location \( A \) is non-zero, then:
   a. Argument byte address is placed in bits 8-31 of GPR 1; bits 0-7 are unchanged;
   b. The non-zero function byte is placed in bits 24-31 of GPR 2; bits 0-23 are unchanged;
   c. The TRT is terminated, setting the CC.

4. At termination of a TRT, the CC is set as follows:
   0 ALL argument bytes were used and no function bytes selected were non-zero
   1 A non-zero function byte has been selected and one or more argument bytes have not yet been used
   2 The last argument byte produced a non-zero function byte (i.e., there are no more argument bytes)

EXAMPLE TRT:

To test string STR, of length L, for the characters +, -, and digits only, could use:

\[ \begin{align*}
\text{TRT} & \quad \text{STR, TAB} \\
\text{BNZ} & \quad \text{ILLEGAL} \\
\text{TAB} & \quad \text{DC} \quad \text{(C+'+'X'FF','X'00',}(C'-C'+1)X'11','X'00',
\quad \text{(C0'-C'-1)X'OF','10X'00','6X'FO'})
\end{align*} \]

When control reaches ILLEGAL, the function byte in GPR 2 will indicate the part of the table indexed by the argument byte.

EXAMPLE TRT:

The following code test the 8 byte string in EX and converts it to a fixed point number in GPR 15 if the string is a hex character string (consists of hex characters only)

\[ \begin{align*}
\text{TRT} & \quad \text{EX(8), TEST} \\
\text{BNZ} & \quad \text{NOGOOD} \\
\text{MVC} & \quad \text{TEMP, EX} \\
\text{TR} & \quad \text{TEMP, TRANS-C'1',A'} \\
\text{PACK} & \quad \text{TEMP(9), TEMP} \\
\text{L} & \quad 15, \text{TEMP} + h
\end{align*} \]

Test for illegal characters
Move string into TEMP
Convert string to binary values
Convert to binary number
Put into GPR 15
TEMP DS D
   DS CL1
TEST DC (C'1A')X'FF', 6X'00', (C'0'-'C'1')X'FF', 10X'00', 6X'FF'
TRANS DC 6ALL(=*-TRANS+10), (C'0'-'C'1')X'FF', 10ALL(=*-TRANS-47)
SAMPLE PROGRAM 1: SUMSQ

ROUTINE SUMSQ COMPUTES THE SQUARE ROOT OF THE SUM OF THE SQUARES OF A VECTOR OF
SHORT FLOATING POINT NUMBERS AND RETURNS THE RESULT IN FPR 0. SUMSQ USES THE
FOLLOWING R-TYPE CALLING SEQUENCE:

GPR 0   NUMBER OF ENTRIES IN VECTOR; MUST BE GREATER THAN ZERO
GPR 1   ADDRESS OF FIRST FULL WORD FLOATING POINT NUMBER IN VECTOR

THE RETURN CODES FROM SUMSQ ARE:
0    RESULT IS IN FPR 0
4    SOMETHING WENT WRONG -- NO RESULT

<table>
<thead>
<tr>
<th>SUMSQ</th>
<th>CSECT</th>
<th>LTR</th>
<th>TEST COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>GOOD-SUMSQ(,15)</td>
<td></td>
<td>BRANCH IF OK</td>
</tr>
<tr>
<td>LA</td>
<td>15,4</td>
<td></td>
<td>SET ERROR RC = 4</td>
</tr>
<tr>
<td>BR</td>
<td>14</td>
<td></td>
<td>AND RETURN</td>
</tr>
<tr>
<td>USING</td>
<td>SUMSQ,10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GOOD

| STM    | 14,12,12(13) | STANDARD ROUTINE ENTRY SEQUENCE WITH |
|--------|--------------| SAVE AREA CHAINING AND REGISTER SAVING |
| LR     | 10,15        |                                          |
| LA     | 2,SAVE       |                                          |
| ST     | 2,6(13)      |                                          |
| ST     | 13,4(,2)     |                                          |
| LR     | 13,2         |                                          |
| SER    | 0,0          |                                          |

LOOP

<table>
<thead>
<tr>
<th>LE</th>
<th>2,0(,1)</th>
<th>O TO FPR 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MER</td>
<td>2,2</td>
<td>NEXT NUMBER TO FPR 2</td>
</tr>
<tr>
<td>AER</td>
<td>0,2</td>
<td>SQUARE THE NUMBER</td>
</tr>
<tr>
<td>LA</td>
<td>1,4(,1)</td>
<td>ADD SQUARE TO SUM</td>
</tr>
<tr>
<td>BCT</td>
<td>0,LOOP</td>
<td>UPDATE ADDRESS</td>
</tr>
<tr>
<td>LA</td>
<td>15,0</td>
<td>COUNT DOWN</td>
</tr>
<tr>
<td>EZ</td>
<td>EXIT</td>
<td>SET RC TO ZERO</td>
</tr>
<tr>
<td>STE</td>
<td>0,HOLD</td>
<td>ALL DONE IF SUM = 0</td>
</tr>
</tbody>
</table>

| LA     | 1,A(HOLD)   | CALL SQRT FOR FINAL RESULT -- LEAVES SQUARE |
|--------|-------------| ROOT IN FPR 0 AND SETS RC = 0 OR 4, WE |
| L      | 15,V(SQRT)  | ASSUME HERE |
| BALR   | 14,15       |                                          |

| EXIT   | 14,4(,13)   | STANDARD ROUTINE RETURN SEQUENCE -- RC |
|--------|-------------| ALREADY SET IN GPR 15 |
| L      | 14,12,12(13)|                                          |
| IM     | 0,12,20(13) |                                          |
| BR     | 14          |                                          |

| HOLD   | DS F        | HOLDS SUM IN SQUARE ROOT CALL |
|--------|-------------| SAVE AREA |
| SAVE   | DS 18F      |            |
| END    |             |            |
SAMPLE PROGRAM 2: FDELE

Routine FDELE searches a doubly-linked list structure, ignoring sublists, for a node containing a specified datum field. If the node is found, it is deleted from the list structure and a pointer to the node is returned. In the list structure each node is 12 bytes long and full word aligned. The node format is:

Byte 0  Key field: 0 = head node; 1 = sublist reference; 2 = datum node
Bytes 1-3  address of successor node
Bytes 4-7  address of predecessor node
Bytes 8-11  sublist address (key = 1) or 32-bit datum (key = 2)

Sample list structure:

EMPTY STRUCTURE:

FDELE uses an R-type calling sequence:

GPR 0  32-bit comparand -- searching for node with this datum value
GPR 1  Address of head of list structure

Possible return codes from FDELE:

0  Node found & deleted; node address in GPR 1
4  List structure is empty
8  Unsuccessful search of non-empty list structure

FDELE
CSECT  FDELE,15
USING   NODE,1
C  1,SLINK
BNE  SAVE
LA  15,4
BR  4
SAVE  L  1,SLINK
CLI  KEY,1
BE  SAVE
BH  LOOP
LA  15,6
BR  4

Program addressability
List structure addressability
Test for empty structure
Branch if not empty structure exit -- set RC = 4
Get next node pointer
Test node key
Branch on sublist reference
Branch on datum node
We're back to the head node
Exit with RC = 8
<table>
<thead>
<tr>
<th>LOOP</th>
<th>C</th>
<th>0, DATUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE</td>
<td>SAVE</td>
<td></td>
</tr>
<tr>
<td>SWPR</td>
<td>2,2</td>
<td></td>
</tr>
<tr>
<td>LM</td>
<td>2,3, SLINK</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>3,l,(2)</td>
<td></td>
</tr>
<tr>
<td>MVC</td>
<td>1</td>
<td>(3,3),SLINK+1</td>
</tr>
<tr>
<td>SWPR</td>
<td>2,2</td>
<td></td>
</tr>
<tr>
<td>SR</td>
<td>15,15</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

**NODE**

<table>
<thead>
<tr>
<th>DSECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF</td>
</tr>
</tbody>
</table>

**KEY**

<table>
<thead>
<tr>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

**SLINK**

<table>
<thead>
<tr>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

**PLINK**

<table>
<thead>
<tr>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

**DATUM**

<table>
<thead>
<tr>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

**END**

**TEST DATUM VALUE**

BRANCH IF NOT EQUAL TO COMPARAND

GOT IT -- SAVE GPRS 2 & 3

GET NODE'S POINTERS

UPDATE SUCCESSOR NODE

UPDATE PREDECESSOR NODE

RESTORE GPRS 2 & 3

SET RC = 0

AND RETURN

**DSECT DESCRIPTION OF NODE FOR IMPLICIT ADDRESSING PURPOSES**
SAMPLE PROGRAM 3: FDELE

THIS EXAMPLE IS THE SAME AS THE PREVIOUS EXAMPLE, EXCEPT THAT HERE NO DSECT IS USED AND ALL ADDRESSING IS EXPPLICIT VIA DEFINED SYMBOLS FOR EASE OF READING AND WRITING THE PROGRAM.

FDELE CSBCT
NODE EQU 1
KEY EQU 0
SLINK EQU 0
PLINK EQU 4
DATUM EQU 8
USING FDELE, 15
C NODE, SLINK(NODE)
BNE SAVE
LA 15, 4
BR 1h
SAVE L NODE, SLINK(NODE)
CLI KEY(NODE), 1
BE SAVE
PH LOOP
LA 15, 8
BR 1h
LOOP C 0, DATUM(NODE)
BNE SAVE
SWFR 2, 2
LM 2, 3, SLINK(NODE)
ST 3, PLINK(.2)
MVC 1(3, 3), SLINK+1(NODE)
SWFR 2, 2
SR 15, 15
BR 1h
END

BASE REGISTER CONTAINING NODE ADDRESS
KEY FIELD DISPLACEMENT
SUCCESSOR FIELD DISPLACEMENT
PREDECESSOR FIELD DISPLACEMENT
DATUM FIELD DISPLACEMENT
PROGRAM ADDRESSABILITY
TEST FOR EMPTY STRUCTURE
BRANCH IF NOT EMPTY STRUCTURE EXIT -- SET RC = 4
GET NEXT NODE POINTER
TEST NODE KEY
BRANCH ON SUBLIST REFERENCE
BRANCH ON DATUM NODE
WE'RE BACK TO THE HEAD NODE
EXIT WITH RC = 8
TEST DATUM VALUE
BRANCH IF NOT EQUAL TO COMPARAND
GOT IT -- SAVE GPRS 2 & 3
GET NODE'S POINTERS
UPDATE SuCCESSOR NODE
UPDATE PREDECESSOR NODE
RESTORE GPRS 2 & 3
SET RC = 0
AND RETURN
PROGRAM SAMPLE 4: CONVERT

GIVEN A STRING OF CHARACTERS OF LENGTH N, ROUTINE CONVERT SEARCHES THE STRING FOR OCCURRENCES OF INTEGER STRINGS AND CONVERTS SUCH INTEGER STRINGS TO FIXED POINT BINARY NUMBERS STORED IN A VECTOR GIVEN AS AN ARGUMENT. CONVERT OPERATES WITH THE FOLLOWING ASSUMPTIONS:
1. EACH INTEGER STRING ENDS IN ONE OR MORE NON-DIGITS;
2. NO INTEGER STRING CAUSES AN OVERFLOW DURING CONVERSION;
3. INTEGER STRINGS ARE CORRECTLY WRITTEN WITHOUT ARITHMETIC SIGNS.

CONVERT USES AN S-TYPE CALLING SEQUENCE WITH PARAMETERS:
ST ADDRESS OF THE CHARACTER STRING
N FULL WORD LENGTH OF THE STRING (TRUE LENGTH)
V ADDRESS OF THE FULL WORD FIXED POINT RESULT VECTOR

THE RETURN CODES USED BY CONVERT ARE:
0 ONE OR MORE NUMBERS PLACED IN V; NUMBER OF ENTRIES IN V IS IN GPR 0
1 NO ENTRIES STORED IN V

THE ALGORITHM USED BY CONVERT IS AS FOLLOWS:
1. FIND NEXT DIGIT IN CHARACTER STRING USING TRT
2. IF NO MORE DIGITS IN STRING, THEN ALL DONE
3. FIND NEXT NON-DIGIT IN CHARACTER STRING USING TRT
4. CONVERT THE INTEGER STRING TO FIXED POINT
5. STORE THE FIXED POINT RESULT IN THE NEXT RESULT VECTOR ENTRY
6. UPDATE THE RESULT VECTOR INDEX
7. RETURN TO STEP 1.

CONVERT USES GPRS AS FOLLOWS DURING PROCESSING:
1 ADDRESS OF REMAINING PART OF CHARACTER STRING TO BE PROCESSED
2 USED BY THE TRTS
3 INDEX ON THE RESULT VECTOR V
4 ADDRESS OF THE RESULT VECTOR V
5 ADDRESS OF THE LAST BYTE IN THE CHARACTER STRING
6 CURRENT TRT LENGTH OR MOVE LENGTH
8 255 - FOR COMPARISON PURPOSES
9 HOLDS FIRST DIGIT ADDRESS OF INTEGER STRING
10 PROGRAM ADDRESSABILITY

CONVERT CSECT
USING *,10
STM 11,12,12(13)
LR 10,15
LA 12,SAVE
ST 12,(13)
ST 13,(12)
LR 13,12
SR 3,3
L 4,8(1)
L 6,4(1)
L 6,0(6)
LTR 6,6
BNE 12,DONE
L 1,0(1)
AR 6,1
BCTR 6,0
LA 8,255

{ STANDARD ENTRY SEQUENCE }
0 TO INDEX
ADDRESS OF V TO GPR 4
ADDRESS OF N TO GPR 6
N TO GPR 6
IF N LESS THAN ZERO, OR ZERO,
BRANCH HERE - WE'RE DONE
ADDRESS OF ST TO GPR 1
ADDRESS OF LAST BYTE OF ST TO GPR 6
255 TO GPR 8 FOR COMPARISONS
LAST BYTE ADDRESS TO GPR 7
LR
IBM LENGTH TO GPR 7
SR
IF NEGATIVE LENGTH, ALL DONE
BM
DONE
IF LENGTH GREATER THAN 255,
CR
**6
BNH
LR
8,8
EX
7,EXDIG
BNZ
DIGIT
LA
1,256,(1)
B
LOOP
DIGIT
LR
9,1
LR
7,6
SR
7,1
CR
7,8
BNH
**6
LR
7,8
EX
7,EXCHR
LR
7,1
SR
7,9
ECTR
7,0
EX
7,PACK
CVB
5,NUM
ST
5,0(3,4)
LA
3,4,(3)
B
LOOP
DONE
LA
15,4
LTR
0,3
BEZ
EXIT
LA
15,0
SRA
0,2
EXIT
L
13,4(13)
L
11,12(13)
LM
1,12,24(13)
BR
11
EXDIG
TRT
0,(1),DIGIT
EXCHR
TRT
0,(1),CHR
PACK
PACK
NUM(8),O(9)
SAVE
DS
18F
NUM
DS
D
DIGIT
DC
16,010X'00',10X'FF',6X'00'
CHRT
DC
16,010X'FF',10X'00',6X'FF'
END

USE 255
LOOK FOR A DIGIT
BRANCH IF FOUND ONE
COMPUTE NEW INITIAL ADDRESS
AND THEN TRY AGAIN
SAVE DIGIT ADDRESS IN GPR 9
LAST BYTE ADDRESS TO GPR 7
IBM LENGTH TO GPR 7
IF LENGTH GREATER THAN 255,
USE 255
FIND NEXT NON-DIGIT
NON-DIGIT ADDRESS TO GPR 7
COMPUTE INTEGER STRING LENGTH
COMPUTE IBM INTEGER STRING LENGTH
CONVERT INTEGER STRING TO PACKED DECIMAL
CONVERT PACKED DECIMAL TO FIXED POINT
STORE RESULT IN RESULT VECTOR
UPDATE RESULT VECTOR INDEX
ROUND AND ROUND WE GO, ........
ALL DONE --- SET RC = 4
TEST RESULT VECTOR INDEX
BRANCH IF NO ENTRIES IN RESULT VECTOR
HAVE ENTRIES --- SET RC = 0
COMPUTE NUMBER OF ENTRIES IN RESULT VECTOR

STANDARD RETURN SEQUENCE

FINDS NEXT DIGIT IN CHARACTER STRING
FINDS NEXT NON-DIGIT IN CHARACTER STRING
PKACS INTEGER STRING INTO NUM
SAVE AREA
HOLDS PACKED DECIMAL NUMBERS
PROGRAM SAMPLE 5

PROBLEM: Prepare, in assembly code, a program which can process a data deck which is structured as follows. The first part of the deck defines a number of LISP elements, giving them a name and exhibiting their structure. E.g.,

```
LIST1  LIST    ( (A (B C)) D E F )
ATOM1  ATOM    B
NIL     LIST    ()
LIST2  LIST    ( A ((B C) D) (E F) )
ATOM2  ATOM    K
```

The second part of the data deck specifies some composed functions which use the now-defined elements as arguments. The program is to read these, one at a time, and evaluate them. E.g.,

```
CONS ( ATOM2, CAR ( LIST1 ) )
CAAADR ( LIST2 )
```

when following the declarations given above would produce the output:

```
(K A (B C))
B
```

a. The program is divided into several pieces which are to be assembled separately.

1. The main part of the program is a MAIN program which is to control the reading of the data cards and the sequencing of the subprograms which will individually calculate one of the LISP primitive functions. In this program is therefore all of the logic for the reading of the first part of the data deck and the construction of the defined elements and the necessary tables so that a reference to the name of an element can be translated to an address of that element. The program also contains the logic for the reading and decoding of the functions in the second part of the data. This part of the program assures that calls are placed to the function subprograms in the appropriate order and that the final result is printed.

This part of the program is also the place in which the pools are defined and initialized as necessary. A very simple organization is used -- the individual data items are used from the pool in sequential order, and linking among the available data items in a pool is therefore not necessary. Whenever an available data item is used, the address of the next available one may be determined by adding the length of the data item to the address of the one just used.

2. The other parts of the program are each one of the primitive LISP functions. They are called by the main program as needed. There is also a garbage collection routine that is called upon to organize the available storage into pools whenever the previously available pools are depleted.

b. The following register usages rules are determined in advance of code preparation.

1. OS (1) 5 calling conventions will be followed and this fixes the use of register 13 as a pointer to the current save area, register 14 as the return address for a call, register 15 as the entry point for a called routine, register 1 as a pointer to the parameter list, and register 0 as the value returned by the functions.
2. Register 12 will be used as the base register for all control sections.

3. Registers 9, 10, and 11 will be used to cover the storage area used for the pools.

(Note: the programs presented below are not complete. Neither are all of the function subprograms specified, nor is the full text of the main program given. Rather, the partial programs are used to give examples of inter-program and inter-assembly referencing and the use of system macros and system subroutines.)

c. First assembly (MAIN program and data pools) written without macros.

<table>
<thead>
<tr>
<th>MAIN</th>
<th>CSECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>USING</td>
<td>*,12</td>
</tr>
<tr>
<td>STM</td>
<td>14,12,12(13)</td>
</tr>
<tr>
<td>LR</td>
<td>12,15</td>
</tr>
<tr>
<td>LA</td>
<td>11,SAVEAREA</td>
</tr>
<tr>
<td>ST</td>
<td>11,8(0,13)</td>
</tr>
<tr>
<td>ST</td>
<td>13,4(0,11)</td>
</tr>
<tr>
<td>LR</td>
<td>13,11</td>
</tr>
</tbody>
</table>

LM 9,11,ADDRS | establish the base registers for the data pools |

LM POOLS,9,10,11 |

* |

L 1,A(SCPMLIST) | part of the data card reading |
L 15,V(SCARDS) | code, showing a call to the |
BALR 14,15 | SCARDS system subroutine |

* |

CALL TO THE CAR SUBROUTINE |

ST 3,PARM | get parameter value stored |

L 1,A(PARMLIST) | activate |
L 15,V(CAR) | the car |
BALR 14,15 | function |

B **+A(15) | accept the return |
B CAROKAY | code and branch |
B CARERROR | to appropriate code |

* |

CAROKAY : |

CARERROR : |

* |

DATA FOR THIS PROGRAM |

SAVEAREA DS 18F | normal save area |
PARM DC F | holds parameter to car |
PARMLIST DS OF | (not really required) parm list |
DC X '80' | (the one and only parm) for |
DC AL3(PARM) | (parm's address) car function |
SCRMLIST DC A(BUFF) | parameter list |
DC A(LENGTH) | for |
DC A(MODS) | SCARDS |
DC A(LINENBR) | subroutine |

(continued)
BUFF DS CL80 area for input from SCARDS
LENGTH DS H number of bytes read
MODS DC F'0' special input modifiers
LINENBR DS F
ADDRS DC A(POOLS,POOLS+4096,POOLS+8192)

* THIS ASSEMBLY ALSO CONTAINS THE DEFINITION OF THE POOLS FOR THE DATA

POOLS CSECT
    DS OD (not really necessary)
    NXTN DC A(NPOOL) pointer to next avail. node
    NXTP DC A(PPOOL) pointer to next avail. pointer
    NPOOL DS 8000C
    PPOOL DS 4000C
    END MAIN
d. First assembly written with the aid of system macros.

MAIN CSECT
    ENTER 12,SA=SAVEAREA use calling conventions
    LM 9,11,ADDRS establish the base registers
    USING POOLS,9,10,11 for the data pools
    :
    SCARDS BUFF,LENGTH,MODS,LINENBR (will set up parm list)
    :
    * CALL TO THE CAR SUBROUTINE
    ST 3,PARM get parameter value stored
    CALL CAR,(=A(PARM)),VL (will also set up parm list)
    EXTRN CAR
    *
    B +=4(15) accept the return
    B CAROKAY code and branch
    B CARError to appropriate code
    *
    CAROKAY :
    CARError :
    *
    * DATA FOR THIS PROGRAM
    SAVEAREA DS 16F normal save area
    PARM DS F holds parameter to car
    BUFF DS CL80 area for input from SCARDS
    LENGTH DS H number of bytes read
    MODS DC F'0' special input modifiers
    LINENBR DS F
    ADDRDS DC A(POOLS,POOLS+4096,POOLS+8192)
    *
    * THIS ASSEMBLY ALSO CONTAINS THE DEFINITION OF THE POOLS FOR THE DATA
    POOLS CSECT
    DS OD (not really needed)
    NXTN DC A(NPOOL) pointer to next available node
    NXTP DC A(PPOOL) pointer to next avail. pointer
    NPOOL DS 8000C
    PPOOL DS 4000C
    END MAIN
e. Note that in the previous program, SCARDS could also have been invoked by the coding:

\[
\text{CALL \ SCARDS, (BUFF, LENGTH, MODS, LINENBR)}
\]

f. Second assembly (CAR function) with strict adherence to the calling conventions.

```
CAR
CSECT
ENTER 12,SA=SAVEAREA

* + L 3,0(0,1)  \text{pick up the address of the list argument (value of \texttt{parm})}
  L 3,0(0,3)

* + LTR 3,3  \text{check to see that it isn't an atom or a null list}
  BM ERROR

* \text{VALID ARGUMENT RECEIVED}
  L 0,0(0,3)
  L 13,4(0,13)
  LM 14,15,12(13)
  LM 1,12,24(13)
  SR 15,15
  BR 14

* \text{INVALID ARGUMENT RECEIVED}
  L 0,0(0,3)
  L 13,4(0,13)
  LM 14,15,12(13)
  LM 1,12,24(13)
  SR 15,15
  BR 14

* \text{ERROR}
  RETURN (14,12),RC=4  \text{RC=4 so that \texttt{MAIN}'s code will work}

SAVEAREA DS 18F
END
```

g. Second assembly may be shortened because CAR doesn't 1) call any other routine and 2) use any registers except 3 as work registers.

```
CAR
CSECT
USING *,15
ST 3,SAVEAREA

* + L 3,0(0,1)
  L 3,0(0,3)

* + LTR 3,3
  BM ERROR

* \text{VALID ARGUMENT RECEIVED}
  L 0,0(0,3)
  L 3,SAVEAREA
  SR 15,15
  BR 14

* \text{INVALID ARGUMENT RECEIVED}
  L 3,SAVEAREA
  LA 15,4(0,0)
  BR 14

* \text{SAVEAREA DS F}
END
```
h. Third assembly (CONS function) showing use of a dummy control section.

CON S
CSECT
ENTER 12,SA=MYSAVE
USING POOLAREA,9,10,11

* LM 7,8,0(1)
   L 7,0(0,7)
   L 8,0(0,8)

* L 6,NEXTNODE
   LA 5,PTRPOOL
   CR 5,6
   BH OKAY
   L 15,=V(GARBAGE)
   BALR 14,15
   LTR 15,15
   EZ OKAY
   RETURN (14,12),RC=4

* OKAY LTR 8,8
   BM NNNLIST

* SECOND ARGUMENT IS A LIST
LISTARG STM 7,8,0(6)
LR 5,6
LA 6,8(0,6)
ST 6,NEXTNODE
L 4,4(0,13)
ST 5,20(0,4)
RETURN (14,12),RC=0

* SECOND ARGUMENT IS AN ATOM OR A NULL LIST
NNNLIST L 4,=X'80000000'
CR 4,8
BE LISTARG
ST 8,0(0,6)
ST 4,4(0,6)
LR 8,6
LA 6,8(0,6)
CR 5,6
BH LISTARG
L 15,=V(GARBAGE)
BALR 14,15
LTR 15,15
EZ LISTARG
RETURN (14,12),RC=4

* MYSAVE DS 18F

* POOLAREA DSECT
   DS 0D

* NEXTNODE DS F
   DS 8004C
   DS 4000C

END

be conventional
these cover data pools
get the value of the
two arguments
find out the address of the
next available node in the
pool and make sure that
it is really a legal node.
If no nodes are available
then call collector routine
(note that it's a parameter-
less call) and interpret the
return code.
check the type of the
second argument
put ptrs side-by-side in a node
save pointer to node, it's fnc value
update the pointer to the
next available node in pool
save function's value into caller's
savearea (reg 0 location)
if the second arg is a null
list then it can be handled
above
turn second argument into a list
composed of the single atom
which was original second arg
try to get another node in which
to stick the two parts of
the result list. If the node
can be found, then let the
code above handle the
processing. Otherwise,
return error return code.

a grey flannel savearea
this dummy control section
maps out all that needs to
be known about the pattern
of this storage area.
I. References


II. Macro-assembler language is a special programming language, embedded in regular assembler language, the instructions of which are obeyed during assembly and which have the effect of producing sequences of assembler instructions which are later translated into machine instructions. The attributes of a macro-assembler language in general and the IBM O.S. version in particular are:

- it is basically a symbol manipulation language
- it is a reasonably primitive language, with many restrictions (not all of which are reported here) and very little error checking
- it allows for conditional assembly — the ability to perform arithmetic, logical and character manipulations at assembly time in order to control the sequence of code which is assembled
- it has special conventions for denoting variables, labels, etc., since it is embedded in another programming language
- it normally is used to write "subroutines" which are invoked during assembly, but the instructions in the language may also be used within the main body of the program

III. The rest of this overview will use the following (IBMese) terms. The list is not meant to be complete — many more terms are defined in the course of discussion — but rather to collect together some often used terms some of which, of necessity, must be used before they can be fully and properly defined. Also, the definitions indicate some of the synonyms which will be used in the discussion.

- variable symbols: these are the identifiers of the macro-assembler language; they consist of 1-7 characters, the first being alphabetic, preceded by ‘&’ . Caution: the meaning of some identifiers is predefined and their values are automatically maintained by the system; all of these begin with ‘ASYS’ and the user should refrain from using names which begin with these characters.
sequence symbols: these are the labels of the macro-assembly language; they consist of 1-7 characters, the first being alphabetic, preceded by ' '.

symbolic parameters: these variable symbols are the formal parameters which receive the values of the arguments upon invocation of a macro.

ordinary symbols: symbols which are valid within normal assembler language.

model statement: a macro-assembly statement which specifies the format and partial contents for a statement which is to be generated.

conditional assembly statements: those statements of the macro-assembly language which do not directly cause the generation of an assembler language statement but rather affect the sequence in which model statements or regular assembler statements are reached.

macro definition: these are the subroutines of the macro-assembler language.

macro instructions: these are the call statements of the macro-assembler language.

keyword parameter: in form, this is a symbolic parameter followed by the character '=' followed by almost any string of 0-255 characters; the use of a keyword parameter is explained below.

V. Notation

[...] indicates that the item enclosed in square brackets is optional.
the word 'blank' indicates that a field of an instruction is to be left blank.

V. Macro Definition. Basically, this is a subroutine which may be invoked during assembly for the purpose of generating the assembler statements which are to be translated into machine code. Its format is:

```
Header:                blank   MACRO   blank
Prototype:             A statement giving the name of the macro and its symbolic parameters; this is also the entry point when the definition is invoked by a macro instruction.
Body:
Declaration:           Zero or more declarations of the variables to be used in assembler time calculations or global variable declarations and recursive those of local variables.
Commands:              A list, perhaps empty, composed of model statements, copy, move, and TYPE statements, comments, conditional assembly statements, and macro instructions.
Trailer:               [sequence symbol]   MPND   blank
```
A COPY statement with the format

\[
\text{blank COPY ordinary symbol}
\]

may be used within the body in order to copy model, MEXIT, MNRTT, and
conditional assembly statements and comments into the definition from
a file.

VI. Program Structure. Basically, macro definitions may not be nested and must
all appear first in the program.

- Program Outline

  - First may come any sequence of listing control statements
    (EJECT, PRINT, SPACE, and TITLE), ICIT, or ISQL statements,
    and comments.

  - Then comes the macro definitions, if any. Interspersed
    between the macro definitions may be any sequence of listing
    control statements, ICIT statements, and comments.

  - Then comes the declarations of the variable symbols, first
    global ones and then local ones, which will be used within
    the model statements and conditional assembly statements
    appearing within the main body of the program.

  - Finally comes the main body of the program including the
    macro instructions, model statements, and conditional
    assembly statements.

  - Note: the macro-assembler language facilities may be used within the
    main body of the program as well as within the macro definitions.

  - Macro definitions may appear within a macro library (under MTS,
    the library is hooked in by assigning it to DSN 0). Definitions
    appearing in the program are scanned first and take precedence
    over any definitions which have the same name appearing in the
    library.

VII. Invoking Macro Definitions. Macro definitions are "called" during assembly
by use of a macro instruction. The macro instruction specifies a set
of arguments which are passed over to the corresponding symbolic parameters
at the time of invocation.

- The prototype in the macro definition of the invoked macro specifies
  the format of the macro instruction which may be used to invoke
  the definition. The format of the prototype is:

\[
[symbolic \text{ parameter}] \text{ name} [\text{ parameter list}] [\text{ comment}]
\]

where:

  - the name is any ordinary symbol which is not already the
    name of an assembler opcode

  - the parameter list is a comma-list of 0-200 symbolic
parameters or keyword parameters.
. the comment may not appear if there is no parameter list
. the parameter list and comment may be interleaved on
  successive lines -- see the manual for details

A macro instruction specifies the arguments which are to be assigned
as the character string values of the symbolic parameters. The
format of a macro instruction is:
  [ordinary symbol] opcode [operand list] [comment]

where:
. the operand list and comment may be interleaved on successive
  lines -- see the manual for details
. the comment may not appear if the operand list does not appear
. the operand list is a comma-list of 0-200 operands each of
  which is a string of 0-255 characters which conforms to the
  following rules:
    . it may not be a sequence symbol
    . it must conform to the usual rules for having apostrophies
      within a quoted character string
    . parentheses must be balanced except within a quoted
      character string
    . the character '=' may appear only
      . as the first character, or
      . within a quoted character string, or
      . inside balanced parentheses
    . any sequence of ampersands must have an even number
      except that a single ampersand may appear as the first
      character
    . commas may appear only within a quoted character string
      or within balanced parentheses
    . blanks may appear only within a quoted character string

  Argument value passing. When a macro definition is invoked by a
  macro instruction the values of the operands are set as the character
  string values of the corresponding symbolic parameters in the proto-
  type of the macro definition.

  Positional correspondence. This is the simplest case, in
  which the ith symbolic parameter in the parameter list takes
  on the value of the ith operand in the operand list.
  Any operand may be omitted and its trailing '=' may
  also be omitted if it becomes trailing in the operand
list. If a positional operand is omitted then the corresponding symbolic parameter is given the null string as its value.

- **Keyword correspondence.** In this case the name of the symbolic parameter which is to receive the value is also stated in the operand list. Since the correspondence is explicitly denoted, the order of the operands need not be the same as the order of the symbolic parameters.

- An element of the parameter list now has the form
  
  symbolic parameter \[\text{value}\]

  where \text{value} has the same form as an operand (given above) except that it may not itself be a symbolic parameter.

- An element of the operand list now has the form
  
  name = value

  where \text{name} is the same name as one of the symbolic parameters but without the '!' character and \text{value} has the same form as an operand (given above).

- If the argument is omitted from the operand list, then the value of the symbolic parameter is set to \text{value}.

- If, in addition, \text{value} was not specified then its value of the symbolic parameter is set to the null character string.

- **Mixing positional and keyword correspondences.** These two modes may be used together in a macro definition but all positional elements must precede all keyword elements in both the operand list and parameter list. The order for the definition of operands stated above still holds. The only stipulation that a macro is left for in the optional portion of an operand list if there are no positional elements following it.

VIII. Model Statements. These instructions in the macro assembler language serve to generate regular assembler statements which are subsequently translated into machine code by the later phase of the assembler.

- The format of a model statement is:
  
  \[[\text{name}\] \text{operator} [\text{operands}] \[\ldots\]]

- \text{name} may be any of the following:
  
  - ordinary symbol
  - variable symbol
. sequence symbol

. the concatenation of an ordinary symbol with a variable symbol
or a variable symbol with one or more additional variable
symbols.

N.B. Concatenation may usually be denoted by
juxtaposition. But if the second item in the
concatenation begins with a letter, digit, '(' or
'.', then the two items must be separated by a
concatenation operator which is a '\'.

. name may not appear when the operator is either ACTR, COPY, END,
ICTL, ISEQ, or ORSYN

. operator may be any of the following:
  . a regular assembler opcode
  . any assembler pseudo-opcode except END, ICTL, ISEQ, ORSYN,
or PRINT
  . the name of another macro definition
  . variable symbol

. operands is a string composed of:
  . ordinary symbols
  . or variable symbols
  . or ordinary and variable symbols concatenated together (The
'.' concatenation operator must be used when an ordinary
symbol beginning with a letter, a digit, '(' or '.' follows
a variable symbol.)

IX. Some simple examples. Figures 1, 2, and 3 show three versions of a simple
entry point macro, first with positional parameters, then with keyword
parameters, and finally with a mixture. Figure 4 shows examples of
concatenation. Figure 5 shows the same macro as in Figure 4 except that
it has been peppered with comments. Note that:

. ordinary comments generate a copy of themselves whenever the macro
is invoked.

. a comment preceded by '/*' in columns 1 and 2 does not generate
anything, i.e. is a comment within the macro-assembler language.

. comments given at the end of model statements appear in the
assembler statements generated from the model statement.

. a variable symbol appearing in any comment is never replaced by
its value.
LOC  OBJECT  CODE  ADDR1  ADDR2  STMT  SOURCE  STATEMENT

1    MACRC
2    &L BRBACK &R1,&R2,&R3
3    &L  L   13,4(0,13)
4    *  B.OUS USING AND DROP -- BELIEVE ME FOR AWHILE, IT'S CORRECT
5    &R3SYSNDX EQU 0
6    USING &R3SYSNDX,13
7    &R1,&R2,&R3SYSNDX+12+4*((&R1+2)-((&R1+2)/16*16))
8    DROP 13
9    LA  15,&R3.(C,O)
10   BR  14
11   MEND
12 *
13  EXIT BRBACK 15,7,0
14+EXIT L   13,4(0,13)
15+BR0001 EQU 0
16+    USING BR0001,13
17+    LM  15,7,BR0001+12+4*((15+2)-((15+2)/16*16))
18+    DROP 13
19+   LA  15,0(0,0)
20+   BR  14
21 *
22  EXIT1 BRBACK 4,9,4
23+EXIT1 L   13,4(0,13)
24+BR0002 EQU 0
25+    USING BR0002,13
26+    LM  4,9,BR0002+12+4*((4+2)-((4+2)/16*16))
27+    DROP 13
28+   LA  15,4(0,0)
29+   BR  14
30 *
31   END

FIGURE 6: Use of the &R3SYSNDX system variable
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MACRO</td>
</tr>
<tr>
<td>2</td>
<td>&amp;LAB</td>
<td>SAFECALL &amp;OPD1,&amp;OPD2,&amp;CALL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&amp;LAB</td>
<td>STM &amp;OPD1,&amp;OPD2,&amp;SYSECT&amp;SYNDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INVOKE</td>
<td>&amp;CALL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LM</td>
<td>&amp;OPD1,&amp;OPD2,&amp;SYSECT&amp;SYNDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>J&amp;SYNDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&amp;SYSECT&amp;SYNDX DS 16F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>J&amp;SYNDX</td>
<td>EQU *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MEND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>MACRO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&amp;LAB</td>
<td>INVOKE &amp;TARGET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ST</td>
<td>13,&amp;SYSECT&amp;SYNDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>LA</td>
<td>13,&amp;SYSECT&amp;SYNDX+4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>BAL</td>
<td>14,&amp;TARGET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>L</td>
<td>13,&amp;SYSECT&amp;SYNDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>B</td>
<td>J&amp;SYNDX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>&amp;SYSECT&amp;SYNDX DS 19F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>J&amp;SYNDX</td>
<td>EQU *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>MEND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>ONE CSECT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>LSING</td>
<td>*+15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>SAFECALL</td>
<td>14,12,BOGUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>STM</td>
<td>14,12,ONE0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>ST</td>
<td>13,ONE0002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>LA</td>
<td>13,ONE0002+4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>BAL</td>
<td>14,BOGUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>L</td>
<td>13,ONE0002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>B</td>
<td>J00002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>31+ONE0002 DS 19F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>J00002</td>
<td>EQU *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>LM</td>
<td>14,12,ONE0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>B</td>
<td>J00001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>35+ONE0001 DS 16F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>J0001</td>
<td>EQU *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>BOGUS</td>
<td>EQU *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 7:** Use of the &SYSECT system variable and an example of the use of both &SYNDX and &SYSECT with nested macro invocations.
XI. Lists of Parameters. It is frequently convenient to think of several arguments to a macro invocation as forming an ordered collection. The possibility of having a variable sized collection is the most interesting aspect; more on that later.

- The arguments are grouped in a comma-list, enclosed in parentheses, and given as one element in the operand list of the macro instruction. Only one level of grouping is allowed, i.e. a list may not appear as an element of a list.
- The corresponding parameter, either positional or keyword, serves as a name for the entire list. Individual elements of the list may be selected by a single integer-valued subscript enclosed in parentheses.

    If the subscript value is positive and selects a non-existent element then the null character string value is used.

- The entire positional portion of an operand list may be referenced as a list by using the system variable &SYSLIST.

    - A single subscript, i.e. &SYSLIST(n), selects the nth positional argument.
    - A double subscript, i.e. &SYSLIST(n,m), selects the mth element of the list which appears as the nth positional argument.

- Figures 8 and 9 give examples.
FIGURE 8: Lists as arguments to an invocation.
OBJECT ADDR1 ADDR2 STMT SOURCE STATEMENT

1     MACRO
2     &L    BRBACK
3     * CAN'T PUT A COMMENT IN THE COMMENT FIELD OF THE PROTOTYPE SINCE
4     * THERE ISN'T ANY PARAMETER LIST -- HAVE TO PUT IT HERE
5     &L    L    13,4(0,13)
6     * BOGUS LSING AND SAVE -- DONE BETTER IN LATER EXAMPLE
7     BR,&SYSNDX EQU 0
8     USING BR,&SYSNDX,13
9     LM    &SYSLIST(1,1),&SYSLIST(1,2),BR,&SYSNDX+12+4*((&SYSLIST(1,1)+2)-((&SYSLIST(1,1)+2)/16*16))
10    DROP   13
11    LA    15,&SYSLIST(2),(0,0)
12    BR    14
13    MEND
14    *
15    EXIT   BRBACK (15,7),0
16+EXIT  L    13,4(0,13)
17+BR0001 EQU 0
18+    LSING BR0001,13
19+    LM    15,7,BR0001+12+4*((15,2)-((15,2)/16*16))
20+    DROP   13
21+    LA    15,0(0,0)
22+    BR    14
23    *
24    EXIT1  BRBACK (4,9),4
25+EXIT1  L    13,4(0,13)
26+BR0002 EQU 0
27+    LSING BR0002,13
28+    LM    4,9,BR0002+12+4*((4,2)-((4,2)/16*16))
29+    DROP   13
30+    LA    15,4(0,0)
31+    BR    14
32    *
33    END

FIGURE 9: Use of the &SYSLIST system variable.
XII. Conditional Assembly Statements. The statements in this category allow
arithmetic, logical and character manipulations to be performed at assembly
time with the context and sequence of the assembled instructions being
affected by the results of the manipulations. These statements may appear
in the main body of the program as well as in macro definitions.

SET-symbols are the user defined variables of the macro-assembler
language. The type of a SET-symbol is either:

- arithmetic (for SETA-symbols) with 32-bit signed integer value
- binary (or really logically; for SETB-symbols) with a value
  of either 0 (for false) or 1 (for true)
- character (for SETC-symbols) with a character string value

which has at most 8 characters.

Conversion among the types is done whenever necessary and possible,
and produces the expected results. Another attribute of a SET-symbol
is its scope which may be either:

- local - the name is defined only within the program segment
  (either a macro definition or the main program) where it is
  declared and its initial value (which is either zero or null
  depending upon its type) is set every time the program segment
  is reached during assembly, which may be several times for
  a macro definition which is invoked several times.

- global - the name is defined within every segment where it
  is declared and its initial value is set only once at the
  beginning of the assembly. (note that the word global is
  used almost but not quite the way it would be for a block-
  structured language)

The final attribute of a SET-symbol is whether it is a simple variable
or a dimensioned variable. If it is dimensioned:

- only a single dimension is allowed
- its size must be between 1 and 2500

Declaration of SET-symbols:

- local SET-symbols are declared by an instruction of the form:

  blank  l-decl-op  namelist

where:

- l-decl-op is either LCLA, LC1B, or LCLS depending on
  the type of the variables being declared
- namelist is a comma-list of variable symbols


. global SET-symbols are declared within each program segment which wants to reference their value by an instruction of the form:

    blank g-decl-on namelist

where namelist is as defined above and g-decl-on is either GBLA, GRLB, or GRLC depending on the type of the variables being declared.

. if the SET-symbol is to be dimensioned then it appears in the namelist of its declaration with its size, given as an integer constant, enclosed in parentheses following it. If the SET-symbol is global then every declaration of it must quote the same size.

. in any set of declarations, the ones for global SET-symbols must precede those for local SET-symbols.

Attributes. These are essentially predefined functions which may be used to query characteristics such as type, length, scaling, etc., of SET-symbols, ordinary symbols, and symbolic parameters.

. The attributes of a symbolic parameter are those inherited from its corresponding argument.

. The attributes of a list are generally those of the first item in the list.

. The available attributes are:

. T' - returns a single character denoting the type; A indicates an A-type address constant; B indicates a binary constant; C indicates a character constant; etc.

. L' - returns an integer number indicating the number of bytes allocated to the symbol

. S' - scale factor of the symbol

. T' - width of the integer field of the symbol

\[ \text{Fixed: } T' = 2 \times (T' - 1) - S' \]
\[ \text{Floating: } T' = 2 \times (T' - 1) - S' \]
\[ \text{Packed: } T' = 2 \times (T' - 1) - S' \]
\[ \text{Zoned: } T' = T' - S' \]

. K' - number of characters in an argument, when the argument is a list then the count includes parentheses and commas
- \( N' \) - number of items in a list
- \( K' \) and \( N' \) may be used only inside macro definitions

Operations on SETA-symbols
- all evaluation is done in 32-bit signed integer arithmetic
- character strings are converted to integers if necessary; if the string does not represent an integer, then an error is raised
- when a SETA-symbol is used in a model statement, its absolute value is converted to a string and leading blanks are deleted
- the assignment statement for setting values of SETA-symbols has the form

\[
\text{SETA-symbol} \quad \text{SETA expression}
\]

where the expression is composed of variable symbols combined by the operators +, -, *, and / with parentheses allowed for readability or overriding of default precedence

- Figures 10 and 11 illustrate the use of SETA-symbols by giving two final versions of the BRACK macro of previous examples, first by using just local SET-symbols and then by using local and global SET-symbols,
<table>
<thead>
<tr>
<th>LCC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STM</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&amp;LABEL</td>
<td>BRBACK</td>
<td>ER=(14,12),&amp;RC=0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>* * NOW WE CAN DO IT RIGHT</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>LCLA</td>
<td>OFF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&amp;LABEL</td>
<td>L</td>
<td>13+4(0,13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&amp;OFF</td>
<td>SETA</td>
<td>12+4*((ER(1)+2)-((ER(1)+2)/16*16))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>LM</td>
<td>&amp;R(1),&amp;R(2),&amp;OFF*(13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>LA</td>
<td>15+&amp;RC*(C,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>BR</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>MEND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>BOTTOM</td>
<td>BRBACK</td>
<td>RC=4,R=(7,12)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>&amp;BOTTOM</td>
<td>L</td>
<td>134(0,13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>LM</td>
<td>7,12+8(13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>LA</td>
<td>154(0,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>BR</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>ROCK</td>
<td>BRBACK</td>
<td>R=(4,9),RC=4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>&amp;ROCK</td>
<td>L</td>
<td>134(0,13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>LM</td>
<td>4,9,36(13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>LA</td>
<td>154(0,0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>BR</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>END</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 10:** Use of SETA symbols.
LOC OBJECT CODE ADDR1 ADDR2 STMT SOURCE STATEMENT

1          MACRO
2       &LAB  BRBACK &RC
3 .  * THE RIGHT WAY AGAIN, BUT THIS TIME USING GLOBAL SYMBOLS TO
4 .  * PASS SOME OF THE ARGUMENTS
5       GBLA   &R1,&R2
6       LCLA    &OFF
7       &LAB    L13,4(0,13)
8 &OFF     SETA 12+4*((&R1+2)-((&R1+2)/16*16))
9       LM     &R1,&R2,&OFF.(13)
10      LA     15,&RC.(0,0)
11      BR     14
12      MEND
13      GBLA   &R1,&R2
14      GBLA   &R1,&R2
15      &R1     SETA 7
16      &R2     SETA 12
17      BOTTOM  BRBACK 4
18      BOTTOM  L13,4(0,13)
19      L13,4(0,13)
20      LM     7,12,48(13)
21      LA     15,4(0,0)
22      BR     14
23      &R1     SETA 4
24      &R2     SETA 9
25      ROCK   BRBACK 4
26      ROCK   L13,4(0,13)
27      L13,4(0,13)
28      LM     4,9,36(13)
29      LA     15,4(0,0)
30      BR     14
31      END
32

FIGURE 11: The example given in Figure 10, redone with the use of some global variables.
Operations on SETC-symbols

- concatenation is performed as before -- juxtaposition can normally be used, but the ',' concatenation operator must be used whenever the second item in the concatenation begins with a letter, digit, ',(', or '}'.

- for the rest of this discussion we need the following definition:

  A character expression is a string of not more than 255 characters enclosed in parentheses, which is composed of ordinary and variable symbols (both symbolic parameters and SETC-symbols) concatenated together; any instance of a variable symbol will be replaced by the value of the variable symbol.

- a substring selection operation may be performed by using:

  character-expression(first,length)

  where first and length may not be expressions and must evaluate to integer values; note that the subscript selectors may only be applied to character expressions and that the numbering of the character positions in a character expression begins with 1

- the assignment statement for setting values of SETC-symbols has the form:

  SETC-symbol SETC c-expr

  where c-expr is the concatenation of one or more character expressions and substring selections

- Figure 12 gives a simple example of the use of SETC-symbols; it's admittedly ad hoc (better examples will follow) but it shows most of the operations mentioned above.
**FIGURE 12:** Illustrations of the use and misuse of SETC symbols.
Operations involving SETB-symbols

- the value of a SETB-symbol is either 0 or 1, having the meaning false or true respectively
- these values are produced by
  - the comparison of arithmetic or character values by using the comparison operators EQ, NE, LT, GT, LE, and GE
  - the combination of SETB-symbols, 0's, 1's, and comparisons by using the logical operators NOT, AND, and OR
  - these operators must always be preceded and followed by at least one break character (blank, parentheses, or single quote marks)
  - conversion is done as necessary and as possible with the intuitively expected results
- the assignment of a value to a SETB-symbol is accomplished by an instruction of the form:
  \[
  \text{SETB-symbol SETB (l-expr)}
  \]
  where l-expr is either a logical constant (0 or 1), a SETB-symbol, a comparison, or a logical expression formed from any of these by using the logical operators.

- SETB-symbols and logical valued expressions are most naturally used in the control statements which are taken up next -- examples of their use are therefore delayed until after that discussion.
Conditional Transfer and Iteration -- the final set of instructions in the macro-assembly language are those for effecting conditional transfer and iteration.

. the ANOP instruction has the form

    sequence symbol   ANOP   blank

and serves exactly the same function as the Fortran continue statement.

. the AGO instruction has the form

    [sequence symbol]   AGO   sequence symbol

and is used for unconditional transfer of control.

. the AIF statement has the form

    [sequence symbol]   AIF   (l-expr)sequence symbol

where l-expr is as defined on the previous page; the instruction may be used to effect the conditional transfer of control.

. there are no instructions for the direct expression of iteration and loops must be programmed by using the conditional transfer statement. There is one instruction for use in controlling the number of times a loop is traversed, but it is designed for error checking rather than iteration. It has the form

    blank   ACTR   expression

where expression is a SETA expression as defined in a previous section. It is used to set an upper limit on the number of AIF and AGO instructions that should be executed local to the appearance of the ACTR instruction, i.e. within the same macro definition or within the main program -- the default value for this limit is 4096. When that limit is reached, transfer is automatically made to the next program instruction following the macro instruction at the highest level of currently nested macro instructions -- if the limit is reached within the main program rather than within a macro definition, then transfer is made to the END statement. Note that the limit is a local value and follows all the rules for such.

. the MEXIT instruction has the form

    [sequence symbol]   MEXIT   blank

and is the same as a Fortran return statement.

. the MNOTE instruction has the form
[sequence symbol] MNOTE [code] character-string

where character-string is as defined in the section on SFTC symbols and code is a number between 1 and 255 (1 is assumed if code is omitted) or an * . When this instruction is encountered within a macro definition (and it may appear only within a macro definition), instances of variable symbols within the character-string are replaced by their values and the result is printed as an error message. The value of code is figured into the running account of the severity of the errors (which affects the subsequent processing of the program after assembly is finished) except when the code is *, in which case the modified character string is just printed as a comment.

The remaining figures illustrate this final set of instructions.

Figure 13 shows a macro for a generalized move-character operation and illustrates the use of the conditional transfer instructions. Figures 14, 15 and 16 give three versions of a macro for calculating the factorial of a number: first by constructing code to do the calculation, then by doing the calculation at assembly time with an iterative algorithm, and finally by doing the calculation at assembly time with a recursive algorithm.
FIGURE 13: A generalized move-character macro illustrating the use of conditional macro-assembler statements.

continued...
DIAGNOSTICS

**IMT**   **ERROR CODE**  **MESSAGE**

54      ASM0271     MNORE STATEMENT.

1 STATEMENT FLAGGED IN THIS ASSEMBLY

**FIGURE 13: **...continued
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$LABEL</td>
<td>FACT</td>
<td>$EN, &amp;REG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>GBLA</td>
<td>$TEMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>LCLA</td>
<td>$EVEN, &amp;CDD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>LCLB</td>
<td>$ISEVEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.* GET AN EVEN-CDD REGISTER PAIR</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&amp;ISEVEN SETB (GREG/2*2 EQ &amp;REG)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$EVEN SETA &amp;REG</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AIF (&amp;ISEVEN).SKIP</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$EVEN SETA $EVEN+1</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*SKIP ANCP</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&amp;CDD SETA $EVEN+1</td>
</tr>
</tbody>
</table>
| 13  |          |       |       |       | *MNOTE *
| 14  |          |       |       |       | $EVEN AND &CDD ARE USED FOR CALCULATION OF THE FACTORIAL |
| 15  |          |       |       |       | $LABEL LA &CDD, 1 |
| 16  |          |       |       |       | LA $TEMP, $EN |
| 17  |          |       |       |       | C $TEMP, =F'0' |
| 18  |          |       |       |       | RE &E8SYSNDX |
| 19  |          |       |       |       | &E8SYSNDX MR $EVEN, $TEMP |
| 20  |          |       |       |       | BCT $TEMP, &E8SYSNDX |
| 21  |          |       |       |       | &E8SYSNDX LR &REG, &CDD |
| 22  |          |       |       |       | **END **|

FIGURE 14: A macro which constructs the code for the calculation of n!

continued....
23  *
24      GBLA   &TEMP
25      USING   *,15
26  *
27      TEMP
28      SETA   2
29      FACT   6,3
30      *4 AND 5 ARE USED FOR CALC'S
31      LA    5,1
32      LA    2,6
33      C     2, =F'.0'
34      BE    E0001
35      MR    4,2
36      ACT   2, L0001
37      LR    3,5
38      TEMP
39      SETA   5
40      FACT   7,2
41      *2 AND 3 ARE USED FOR CALC'S
42      LA    3,1
43      LA    5,7
44      C     5, =F'.0'
45      BE    E0002
46      MR    2,5
47      ACT   5, L0002
48      LR    2,3
49      END  =F'.0'
50

FIGURE 14: ....continued
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>SMT</th>
<th>SCLRCF</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MACRO</td>
</tr>
<tr>
<td>2</td>
<td>&amp;LABEL</td>
<td>FACT</td>
<td>EN, &amp;REG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>* DCNE</td>
<td>BY AN ITERATIVE ALGORITHM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LCLA</td>
<td>&amp;F, &amp;CTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&amp;F</td>
<td>SETA</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&amp;CTR</td>
<td>SETA</td>
<td>&amp;N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LCCF</td>
<td>AIF</td>
<td>(&amp;CTR EQ 0), END</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;F</td>
<td>SETA</td>
<td>&amp;F* &amp;CTR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>&amp;CTR</td>
<td>SETA</td>
<td>&amp;CTR-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>AGC</td>
<td>LOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>END</td>
<td>ANCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>&amp;LABEL</td>
<td>L</td>
<td>&amp;REG, =F* &amp;F*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>MEND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>USING</td>
<td>* ,15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>FACT</td>
<td>6,3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>L</td>
<td>3, =F'720'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>FACT</td>
<td>7,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>L</td>
<td>2, =F'5040'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>END</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td>=F'720'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td>=F'5040'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 15:** A macro which calculates $n!$ using an iterative algorithm.
FIGURE 16: A macro which calculates n! using a recursive algorithm.
PROGRAM STATUS INSTRUCTIONS

All of these instructions, except Set Program Mask, Supervisor Call, and Test and Set are privileged.

* LPSW  D1(B1)  \(((B1)_D1)_o_{63} \rightarrow PSW\)

* SPM  R1  \{R\}_1{-7} \rightarrow \{PSW\}_D3{_{34-31}}

DSM  D1(B1)  \(((B1)_D1)_o_{63} \rightarrow \{PSW\}_D3{_{0-7}}

SVC  I  \{PSW\}_D3{_{24-31}}

invoke supervisor call interrupt routine

SSK  R1R2  set and load key for

ISK  R1R2  storage protection; (R2) addresses

a byte on some 2048 boundary

of memory. R1 is either source

or target of the storage key.

or this piece of memory

* TS  D1(B1)  \(((B1)_D1)_o_{63} \rightarrow \{CC\}_D3{_{31-16}}

0 \rightarrow \{CC\}_D3{_{0}}

11111111 \rightarrow \{(B1)_D1\}_o_{63}

N.B. operation is indivisible
STATUS SWITCHING:

CPU STATES:

Problem/Supervisor
- privileged instructions executed only in supervisor state
  (I/O, direct control, LPSW, SSM, SSK, ISK, DIAGNOSE)
- state change requires whole new PSW

Wait/Running
- in wait state:
  - no instruction fetch, but timer runs
  - unmasked interrupts accepted
- state change requires whole new PSW

Masked/Interruptible
- system mask
  - 1/0 & external interrupt masks
- machine check mask
  - PSW bit 13
  - masks all system machine checks
- program mask
  - PSW bits 36-39
  - masks for fixed point and decimal overflow, exponent underflow, significance
- in all cases, a bit = 0 indicates "masked off"
- state changed by new PSW, or:
  - SSM changes system mask
  - SPM changes program mask

Stopped/Operating (no PSW bit)
- stopped state entered by machine malfunction or manually;
  - no instruction fetch, no timer update, no interrupts accepted;
  - left only by manual control
- in operating state, instructions are executed (if not in wait state)
  and interrupts accepted (if not masked)

Storage Protection:
- each 2048 byte block of storage has 5 bit protection key:
  - bits 0-3 store protect
  - bit 4 fetch protect
- operation:
  - store reference:
    - PSW/channel key compared to storage block key (4 bits);
    - store is permitted if:
      1. keys match, or
      2. PSW/channel key is zero
      - storage unchanged on mismatch and protection interrupt generated
    - fetch reference:
      - fetch bit = 0 fetch not monitored
      - fetch bit = 1 fetch permitted if storage keys match (as above)
- CPU/channel generated addresses not monitored (PSWs, CSW, ... )
- may protect against store or against store-&-fetch, but not against
  fetch only
- storage key set by SSK & may be inspected by ISK
Interrupt System:

When interrupt occurs and CPU is masked:

a. I/O and external interrupts remain pending
b. program and machine check interrupts ignored

When interrupt occurs and CPU is not masked:

a. current instruction completed, terminated, or suppressed
b. current PSW stored in old PSW for interrupt class
c. new PSW for interrupt class becomes current PSW for CPU -
   new PSW is effective immediately and is not checked until
   actually used by CPU

Interrupt classes (by priority):

<table>
<thead>
<tr>
<th>class</th>
<th>old/new PSWs</th>
<th>interrupt code</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine check</td>
<td>30/70</td>
<td>model dependent</td>
</tr>
<tr>
<td>SVC</td>
<td>20/60</td>
<td>'ooii' ii from SVC</td>
</tr>
<tr>
<td>program</td>
<td>28/68</td>
<td>'ooop' p=1...F</td>
</tr>
<tr>
<td>external</td>
<td>18/58</td>
<td>'ooee' ee=80-timer</td>
</tr>
<tr>
<td>I/O</td>
<td>38/78</td>
<td>'ocdd' c channel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{d d device</td>
</tr>
</tbody>
</table>

for I/O interrupts, additional information is also stored in
the CSW (channel status word), location 40, for the interrupt
routine to peruse

permanently allocated storage:

<table>
<thead>
<tr>
<th>0</th>
<th>dw</th>
<th>initial program loading</th>
<th>PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>dw</td>
<td>initial program loading</td>
<td>CCW1</td>
</tr>
<tr>
<td>10</td>
<td>dw</td>
<td>initial program loading</td>
<td>CCW2</td>
</tr>
<tr>
<td>18</td>
<td>dw</td>
<td>external</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>dw</td>
<td>SVC</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>dw</td>
<td>program</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>dw</td>
<td>machine check</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>dw</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>dw</td>
<td>CSW (channel status word)</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>fw</td>
<td>CAW (channel address word)</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>fw</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>fw</td>
<td>Timer</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>fw</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>dw</td>
<td>external</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>dw</td>
<td>SVC</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>dw</td>
<td>program</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>dw</td>
<td>machine check</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>dw</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>diagnostic scanout area</td>
<td></td>
</tr>
</tbody>
</table>
Software responsibility:

1. set up new PSW locations with appropriate mask bits and instruction addresses
2. load and prepare interrupt routines at specified addresses
3. set current PSW to appropriate value

Integral Timer:
- 32 bit word at location 50-treated as signed integer
- low resolution timer
  - bit positions 21,23 reduced by 1 every 1/60 sec, or bit positions 21,22 reduced by 1 every 1/50 sec.
  - resultant resolution = 1/300 seconds
  - updated between instructions when access permitted - may be delayed or omitted under certain conditions
- high resolution timer:
  - bit 31 reduced every ~ 13 μsec - counts at 300 cps in bit position 23
  - bits 24-31 in backup internal storage-location
    50 updated periodically - may be delayed or omitted under certain conditions
- in either case:
  1. external interrupt generated when timer goes negative-timer keeps ticking
  2. store/fetch reference to location 50 sets/gets full timer value
  3. to change timer without losing tick:
    - put new value in fullword at 54
    - MVC bytes 50-57 → bytes 4C-53
    - timer has new value; old value at 4C
Input/Output (I/O)

CPUs:
- execute I/O instructions
- initiate asynchronous I/O activity
- monitor I/O activity
- receive interrupts for I/O state changes

Channels:
- at most 7 - numbered 0 thru 6
  - multiplexor/selector channels
  - direct info flow between devices and storage - permit
  - concurrent CPU & I/O activity
  - provide standard I/O interface
  - converts CPU control info to proper signal sequences
  - assemble/disassemble data
  - provide registers for channel program execution -
    decode and control channel commands
  - may be one or more control units hooked to given channel -
    they share common bus

Control Units:
- provide logical capability to control and operate I/O device(s)
- interprets channel sequences for devices
- controls data transfer timing over I/O interface
- provides device status to channel - control info to device
- one cu may attach to 1 or 2 channels
- cu may attach one or more devices

I/O Devices:
- many of these: terminals, discs, drums, data cells, printers,
  readers, punches, other computers, etc.
- each has unique 16 bit address:
  - 0-3 zero
  - 4-7 channel (0-6)
    - 0 = multiplexor
    - 1-6=mpx or selector
  - 8-15 cu and device (and subchannel on mpx channel)
  - each path to device has unique address
  - contiguous address sets required for shared subchannels
  - or shared cus addresses set by physical connections

Diagram:

```
Storage
 info. ---------- Channels
 CPU(s) \    control ---- cu
       \ control unit
           \ I/O devices
```
Start I/O Instruction:

SIO \ D_j(B_j) \\
rightmost 16 bits of \ D_j(B_j) \\
taken as I/O address

CAW (location 48) provides key and channel program address:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Key: channel protection key 
CCW address: address of first (or only) CCW to be used

CC setting: 0 I/O in progress
1 CSW stored (bits 32-47 only)
2 channel or subchannel busy
3 not operational

Once started, things continue until:
channel end (ce)
device end (de)
control unit end (cue)
error or unusual condition

All of these are signalled via an I/O interrupt (or several of them)

program procedure:

. 
1) establish CCW program
2) turn off I/O interrupts
3) set CAW
4) set new I/O PSW, if necessary
5) issue SIO
6) test cond code - if nonzero, ...
7) restore I/O interrupts
. 
.
.

system operation (simplified):
1) CPU executes SIO 
2) CAW, first CCW \rightarrow channel 
3) proceed if path available 
4) device address \rightarrow all cus on channel 
5) addressed cu logically attaches and returns its address 
6) first command code \rightarrow cu 
7) device status byte \rightarrow channel 
8) SIO terminates - CPU freed 
9) subchannel continues responding to cu/device service requests and executing CCW program 
10) finally will receive ce and de (sometimes cue, also), possibly along with status bytes from device and channel
Other I/O instructions:

**TIO**  \( D_1(B_j) \)  test I/O  
  tests status of selected path; sets cc as for SIO

**HIO**  \( D_1(B_j) \)  halt I/O  
  stops current operation on specified path, if there is one;  
  sets cc to indicate effect and status

**TCH**  \( D_1(B_j) \)  test channel  
  tests specified channel only and sets cc; cu and device  
  status ignored
Channel Commands (CCWs)

Each command (except TIC) initiates or continues I/O operation
max info transferred = block, as defined by device
I/O op terminates on block or byte count
CCW format:

<table>
<thead>
<tr>
<th>COMM CODE</th>
<th>DATA ADDR</th>
<th>FLAGS</th>
<th>000</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7 8</td>
<td>31</td>
<td>32</td>
<td>36 37</td>
</tr>
</tbody>
</table>

DATA ADDR: address of first buffer byte
COUNT: true byte length of buffer (>0)
FLAGS: 32 CD=1 -> data chaining
        33 CC=1 -> command chaining
        34 SL1=1 -> suppress incorrect length interrupt
        35 SKIP=1 -> suppress info transfer (input only)
        36 PCI=1 -> give program controlled interrupt when
                     CCW used

COMM CODE: last 2 bits (if#0), or last 4 bits, determine command
all 8 bits + cu and device and may be used there as modifiers

Commands recognized:

READ       mmmmm10
sets channel for input and initiates
read at device

READ       BACKWARD mmm1100
as for READ, but backwards,
if device supports it

WRITE      mmmmm01
sets channel for output and initiates
write at device

CONTROL    mmmmm11
channel set up for output flow and
control op set up at device -
control function specified by
modifier bits or by data
transferred to device

TIC        (transfer in channel) xxxx1000
next CCW fetched from data
address - channel not informed -
cannot TIC to TIC or have TIC
as first CCW after S10

SENSE      mmm0100
sets channel for input and initiates
sense op at device; device
returns detailed status info
unique to device and cu, one
or more bytes; status is that
at end of last I/O op on device
for most devices, first 6 bits of sense info are as follows:
   0 command reject
   1 intervention required
   2 bus-out check
   3 equipment check
   4 data check
   5 overrun

Assembler CCW instruction:
[symb]  CCW CC,DA,F,C
       CC  abs exp = command code
       DA  exp = data address
treated as AL3(DA)
       F   abs exp = flags
           bits 37-39 must be set to 0
           assembler zeroes bits 40-47
       C   abs exp = true byte count

command is assembled in CCW format and aligned on double word - skipped bytes (if any) are zeroed

Examples:
For these examples, we assume:
   a. the device is similar to an IBM 1050, which is a typewriter-like device used in most System/360 systems as the operator's console
   b. the I/O address of the console is X'009'
   c. the device commands of interest are:
       READ  (X'0A') reads EBCDIC bytes from the console followed by a carriage return; input lines are assumed to be variable length up to 100 bytes
       WRITER (X'09') writes EBCDIC bytes on console followed by carriage return
       WRITEN (X'01') writes EBCDIC bytes on console and leaves carriage at line end (no carriage return)
   d. the proper masking and unmasking of I/O interrupts is provided by surrounding code
   e. the following symbols are defined as shown in the assembly:
       READ EQU X'0A'
       WRITER EQU X'09'
       WRITEN EQU X'01'
       CDATA EQU X'80'  data chain bit
       CCOMM EQU X'40'  comm chain bit
       SLI   EQU X'20'  suppress length bit
       CAW   EQU X'48'  CAW location
       OPCONS EQU X'009' op cons address
   f. the storage key for all I/O is 5
Example 1: read line from op console to AREA

```
MVC CAW(4,0),MYCAW
SIO OPCONS(0)
BZ OK

MYCAW DC X'50',AL3(COMM)
COMM CCW READ,AREA,SLI,L'AREA
AREA DS CL100
```

Example 2: as above, but put first 10 bytes read in AREA and rest in BUFF

```
MVC CAW(4,0),MYCAW
SIO OPCONS(0)
BZ OK

COMM CCW READ,AREA,CDATA,10
    CCW READ,BUFF,SLI,90
MYCAW DC X'50',AL3(COMM)
AREA,BUFF definitions
```

Example 3: write the string

```
TOO MANY TASKS STARTED
```

on the operator's console:

```
MVC CAW(4,0),SETC
SIO OPCONS(0)
BZ STARTED

SETC DC X'50',AL3(CCW)
CCW CCW WRITER,STR,0,L'STR
STR DC C'\ TOOT MANY TASKS STARTED'
```

Example 4: write the string

```
prefix MOUNT TAPE:
```

with no carriage return, followed by a read from the console;

prefix is string stored in PREFIX

```
MVC CAW(4,0),MYCAW
SIO OPCONS(0)
BZ GOOD

MYCAW DC X'50',AL3(COMS)
COMS CCW WRITEN,PREFIX,CDATA,L'PREFIX
    CCW WRITEN,MESS,COMM,L'MESS
    CCW READ,BUFF,SLI,100
MESS DC C'\ MOUNT TAPE: \'
PREFIX,BUFF definitions
CSW (Channel Status Word): location 40
part, or all, of CSW is filled with I/O information during:

1. S10, T10, H10 if cc = 1; info pertains to addressed device
2. I/O interruption; info pertains to device whose address is in
interrupt code portion of old I/O PSW (bits 16-31 of double
word starting in location 38)

CSW format:

<table>
<thead>
<tr>
<th>Key</th>
<th>0000</th>
<th>cad</th>
<th>status</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 4</td>
<td>7 8</td>
<td>31 32</td>
<td>47 48</td>
</tr>
</tbody>
</table>

- **Key**: channel protection key
- **cad**: address of last CCW used +8
- **count**: residual count from last CCW = CCW count - # bytes transferred
- **status** information:
  - 32-39: device and cu status
  - 40-47: channel status

**unit status bits:**

32 Attention - Asynchronous signal from device;
   may accompany de or come during S10
33 Status Modifier - indicates cu and/or device cannot give
   status
34 Control Unit End - provided by shared cu if it was
   interrogated while busy, or by cu presenting unusual
   condition after ce
35 Busy - device/cu is either busy or has pending interrupt
   operation not started
36 Channel End - subchannel now free; arises after last CCW
   releases channel normally
37 Device End - arises at completion of I/O operation by
   device; may accompany or follow ce
38 Unit Check - device/cu has sensed unusual condition;
   may or may not be an error; should issue SENSE command
   to device to get more info; this is a summary bit
39 Unit Exception - device has detected specific unusual
   condition (EOF, for example); only one such condition
   for each type of device; usually not an error condition

**channel status bits:**

40 PCI - (Program controlled interrupt) - CCW has been
   fetched by channel with PCI bit = 1
41 Incorrect Length - storage area length not same as # of
   bytes requested or presented by device - suppressed by
   SLI bit in CCW
42 Program check - channel has detected programming errors;
   examples:
   - Invalid CCW address
   - Invalid CCW or CAW
   (bad command code, count, data address, ...)
   - Invalid CCW sequence
   (TIC to TIC, ...)

43 Protection check - store or fetch protect by channel on
data/CCW reference "

44-46 Channel and interface equipment error indicators

47 Chaining Check - channel overrun during input with data
chaining
Dynamic Address Translation

Virtual Address

Control Register 0

0 7 8 31
STL STO

Segment Table

8 Associative Registers

0 11 12 19 20 31 36-38
SN PN PBA ACB

Interrupt:
- SN Too Large
- PN Too Large or PTA = 1
- PA = 1

Real Core Page
On Page Boundary
DAT CHART SYMBOLISM:

VIRTUAL ADDRESS:
SN SEGMENT NUMBER; 0 - 15 OR 0 - 4095
PN PAGE NUMBER; 0 - 255
D DISPLACEMENT; 0 - 4095; USED AS RIGHT HALF OF REAL CORE ADDRESS

CONTROL REGISTER 0:
STL SEGMENT TABLE LENGTH IN UNITS OF 16 ENTRIES
STO SEGMENT TABLE ORIGIN; BITS 26 - 31 MUST BE ZERO

SEGMENT TABLE:
MUST BE LOCATED ON 64 BYTE BOUNDARY; THERE WILL BE 1 SEGMENT TABLE FOR EACH USER; THERE WILL BE ONE SEGMENT TABLE ENTRY FOR EACH USER DEFINED SEGMENT.

PTL PAGE TABLE LENGTH (IBM LENGTH)
PTO PAGE TABLE ORIGIN - MUST BE HALF WORD ALIGNED IF IN REAL CORE
PTA PAGE TABLE AVAILABILITY; IF 1, PAGE TABLE NOT AVAILABLE

PAGE TABLE:
MUST BE LOCATED ON EVEN (HALF WORD) ADDRESS; THERE WILL BE ONE PAGE TABLE FOR EACH USER DEFINED SEGMENT (I. E., FOR EACH SEGMENT TABLE ENTRY); THERE WILL BE 1 PAGE TABLE ENTRY FOR EACH PAGE ALLOCATED TO THIS SEGMENT.

PBA PHYSICAL BLOCK ADDRESS = LEFT-MOST 12 BITS OF REAL CORE PAGE ADDRESS
PA PAGE AVAILABILITY; IF 1, REAL CORE PAGE IS NOT AVAILABLE
CB CONTROL BITS RESERVED FOR IBM; MUST BE ZERO

ASSOCIATIVE REGISTERS:
SN SEGMENT NUMBER FROM A VIRTUAL ADDRESS
PN PAGE NUMBER FROM A VIRTUAL ADDRESS
PBA PHYSICAL BLOCK ADDRESS FROM A PAGE TABLE
ACB ASSOCIATIVE CONTROL BITS
BIT 36 SET TO 0 WHEN CRO LOADED; SET TO 1 WHEN AR LOADED
BIT 37 SET TO 0 WHEN CRO LOADED AND WHEN 8TH AR LOADED; SET TO 1 WHEN AR LOADED AND WHEN AR USED IN DAT
BIT 38 MAY BE SET TO DISABLE THE AR; SET BY DIAGNOSE INSTRUCTION

DAT ALGORITHM, GIVEN VIRTUAL ADDRESS (VA) OF SN-PN-D:
1. SEARCH ARS FOR ONE WITH BIT 36 = 1 AND SN-PN MATCHING THAT OF VA;
2. IF NO SUCH AR, GO TO STEP 4;
3. SET AR BIT 37 TO 1; COMPUTE ADDRESS USING PBA FROM AR AND D FROM VA;
4. ADD SN FROM VA TO STO FROM CRO TO GET ADDRESS OF SEGMENT TABLE ENTRY;
5. ADD PN FROM VA TO PTO FROM SEGMENT TABLE ENTRY TO GET ADDRESS OF PAGE TABLE ENTRY;
6. CONCATENATE D FROM VA WITH PBA FROM PAGE TABLE ENTRY TO GET REAL ADDRESS;
7. PICK LOWEST AR WITH BIT 37 = 0 AND ENTER THE SN, PN, AND PBA USED INTO THE AR; SET AR BIT 37 TO 1; SET AR BIT 36 TO 1; END OF DAT.
VIRTUAL MEMORY (VM) USING DAT:

24 BIT ADDRESSING: 4,096 PAGE VM COMPOSED OF 16 SEGMENTS OF 256 PAGES EACH;
TOTAL VM IS 16,777,216 BYTES

32 BIT ADDRESSING: 1,048,576 PAGE VM COMPOSED OF 4,096 SEGMENTS OF 256 PAGES
EACH; TOTAL VM IS 4,294,967,296 BYTES

COMMENTS ON DAT:

1. DAT APPLIES TO PROGRAM GENERATED ADDRESSES ONLY - IT DOES NOT APPLY TO
   CHANNEL ADDRESSES OR TO HARDWARE GENERATED ADDRESSES

2. FOR INSTRUCTION SEQUENCING, INTERNAL REGISTER KEEPS THE REAL CORE ADDRESS
   OF THE NEXT INSTRUCTION TO BE EXECUTED - DAT NEEDED ONLY WHEN CROSSING
   PAGE BOUNDARIES OR WHEN BRANCHING OR STATUS SWITCHING

3. FOR MULTIPLE CPUS, EACH CPU HAS ITS OWN DAT HARDWARE

4. CPU ADDRESS PREFIXING IS APPLIED AFTER DAT

5. DAT TIMING COSTS:
   a. FOR DAT USING AN AR, TRANSLATION ADDS 150 NANOSECONDS TO EACH STORAGE
      REFERENCE
   b. FOR DAT USING SEGMENT AND PAGE TABLES, EACH ADDRESS TRANSLATED REQUIRES
      2 STORAGE REFERENCES (FOR PAGE AND SEGMENT TABLE REFERENCES) PLUS
      750 NANOSECONDS, PLUS THE DATUM REFERENCE ITSELF AFTERWARDS
REAL CORE SHARING:

Sharing of real core between two users (for re-entrant routines, for example) may be done in one of two ways with this segment-page table structure. In the first way, each user uses his own segment and page tables to reach the shared region. In this case, the virtual segment and page numbers used by each user may be the same or different. In diagram, this situation is as follows:

```
    VA1  ST  VA2
  ┌──────> PT ───────────┐
  │                   │
  │                   │
  │                   │
  └──────> REAL PAGE └──────

    USER 1
    USER 2
```

Note that in the above case individual real pages may be shared in one or more user segments. The second technique for sharing real pages is to share page tables. Here, each user has his own segment table, but some of the page tables are joint between the users sharing real core. By diagram, this situation is as follows:

```
    VA1  ST  VA2
  ┌──────> PT ───────────┐
  │                   │
  │                   │
  │                   │
  └──────> REAL PAGE └──────

    USER 1
    USER 2
```

In this case, the two users may use different segment numbers, but they will use the same page numbers to refer to the same real page. Here, however, they must also share the whole segment represented by the shared page table. On the other hand, if the real page is paged out there is only one page table to modify.
### EXTENDED PFSW FORMAT:

<table>
<thead>
<tr>
<th>BIT(S)</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>SPARE - MUST BE ALL ZERO</td>
</tr>
<tr>
<td>4</td>
<td>DAT ADDRESS MODE: 0 = 24 BIT VM ADDRESSES; 1 = 32 BIT VM ADDRESSES</td>
</tr>
<tr>
<td>5</td>
<td>DAT CONTROL: 0 = NO DAT; 1 = USE DAT ON PROGRAM GENERATED ADDRESSES</td>
</tr>
<tr>
<td>6</td>
<td>I/O SUMMARY MASK: 0 = ALL CHANNELS MASKED; 1 = SEE CR 4</td>
</tr>
<tr>
<td>7</td>
<td>EXTERNAL SUMMARY MASK: 0 = ALL MASKED; 1 = SEE CR 6</td>
</tr>
<tr>
<td>8-11</td>
<td>PROTECTION KEY - SAME AS FOR STANDARD PFSW</td>
</tr>
<tr>
<td>12-15</td>
<td>AMMP - AS FOR STANDARD PFSW (ASCII MODE, CPU MACHINE CHECK MASK, WAIT STATE, PROBLEM STATE)</td>
</tr>
<tr>
<td>16-17</td>
<td>INSTRUCTION LENGTH CODE (ILC) - AS IN STANDARD PFSW</td>
</tr>
<tr>
<td>18-19</td>
<td>CONDITION CODE</td>
</tr>
<tr>
<td>20-23</td>
<td>PROGRAM MASK - AS IN STANDARD PFSW</td>
</tr>
<tr>
<td>24-31</td>
<td>SPARE</td>
</tr>
<tr>
<td>32-63</td>
<td>NEXT INSTRUCTION ADDRESS (VM ADDRESS)</td>
</tr>
</tbody>
</table>

### INTERRUPTION CODES:

In extended PFSW mode, interruption codes are not stored as part of the old PFSW during interrupt processing. Rather, interruption codes are stored in memory as follows (hexadecimal addresses):

- **EXTERNAL**: E-F
- **SVC**: 10-11
- **PROGRAM**: 12-13
- **MACHINE CHECK**: 14-15
- **I/O**: 16-17

### EXTENDED INTERRUPTIONS:

**SPECIFICATION INTERRUPTS (CODE = 6):**

1. EXTENDED PFSW BIT 4 = 1 AND 32-BIT OPTION NOT INSTALLED
2. EXTENDED PFSW BITS 0-3 NOT ALL ZERO
3. CONTROL BITS 13-15 OF PAGE TABLE ENTRY NOT ALL ZERO

**DATA EXCEPTION (CODE = 7):**

Generated if bits 26-31 of control register 0 (the segment table register) are not all zero

**SEGMENT TRANSLATION EXCEPTION (CODE = HEX 10):**

1. SEGMENT NUMBER IN VIRTUAL ADDRESS TOO LARGE (32-BIT MODE ONLY)
2. BIT 31 OF SEGMENT TABLE ENTRY (PTA) IS 1

**PAGE TRANSLATION EXCEPTION (CODE = HEX 11):**

1. PAGE NUMBER IN VIRTUAL ADDRESS GREATER THAN PAGE TABLE LENGTH (PTL)
2. BIT 12 OF PAGE TABLE ENTRY (PA) IS 1

For both the segment translation exception and the page translation exception, the offending virtual address is stored in control register 2 as part of the interrupt processing.
CONTROL REGISTERS:

0  SEGMENT TABLE REGISTER FOR DAT
0-7  SEGMENT TABLE LENGTH IN MULTIPLES OF 16
8-31  SEGMENT TABLE ORIGIN

2  TRANSLATION EXCEPTION ADDRESS REGISTER - VIRTUAL ADDRESS IS STORED HERE
    WHEN A TRANSLATION EXCEPTION OCCURS DURING DAT

4  EXTENDED I/O CHANNEL MASKS:
  0-6  I/O MASKS FOR CHANNELS 0 THROUGH 6
  7  SUMMARY BIT - SET TO 0 IF BITS 0-6 ALL ZERO
  8-14  I/O MASKS FOR CHANNELS 7 THROUGH 13
  15  SUMMARY BIT - SET TO 0 IF BITS 8-14 ALL ZERO
  16-31  RESERVED - CURRENTLY UNUSED

6  EXTENDED INTERRUPT MASK BITS:
  0-1  MACHINE CHECK MASK EXTENSIONS FOR CHANNEL CONTROLLERS
  2-3  RESERVED
  4-7  UNASSIGNED
  8  EXTENDED CONTROL MODE - 1 = EXTENDED PSW MODE
  9  CONFIGURATION CONTROL BIT - DEFINES WHEN PARTITIONING MAY OCCUR
  10-23  UNASSIGNED
  24-31  EXTERNAL INTERRUPT MASKS:
  24  TIMER
  25  INTERRUPT KEY
  26  MALFUNCTION ALERT - CPU 1
  27  MALFUNCTION ALERT - CPU 2
  28  RESERVED
  29  RESERVED
  30  EXTERNAL INTERRUPT
  31  RESERVED

8-14  PARTITIONING SENSING REGISTERS - FOR READING CONFIGURATION CONSOLE

NOTE: IN THE ABOVE, "UNASSIGNED" MEANS NOT IMPLEMENTED; "RESERVED" MEANS
    IMPLEMENTED BUT NOT YET ASSIGNED SPECIFIC FUNCTION. CONTROL REGISTERS
    1, 3, 5, 7, AND 15 ARE UNASSIGNED.

NOTE: A CPU MAY RUN IN STANDARD PSW MODE OR IN EXTENDED PSW MODE - THIS MODE
    IS DETERMINED BY BIT 8 OF CONTROL REGISTER 6. IF RUNNING IN EXTENDED
    PSW MODE, THE CPU MAY OR MAY NOT USE DAT - THIS IS DETERMINED BY BIT 5
    OF THE EXTENDED PSW. FINALLY, IF USING DAT, THE VIRTUAL ADDRESSES
    MAY BE 24 OR 32 BITS IN LENGTH, AS DETERMINED BY BIT 4 OF THE EXTENDED
    PSW.
IBM 2846 CHANNEL CONTROLLER (CC):

THE 2846 CC IS USED IN 2067-2 CONFIGURATIONS AND PROVIDES:
1. CPU-CHANNEL INTERFACE
2. PROCESSOR STORAGE UNIT-CHANNEL COMMUNICATION INTERFACE

THE CC:
1. SCANS ATTACHED CHANNELS AND PROVIDES STORAGE CYCLES WHEN NEEDED
2. PROVIDES PREFIXING ON CHANNEL GENERATED STORAGE ADDRESSES
3. MONITORS PASSAGE OF I/O INTERRUPTS AND RELATED INFORMATION BACK TO A CPU
4. MONITORS SELECTION AND INITIATION OF CHANNEL OPERATIONS

A 2846 CC MAY INTERFACE WITH:
1. UP TO TWO 2067-2 CPUS
2. UP TO EIGHT 2365-12 PROCESSOR STORAGE UNITS
3. UP TO SEVEN PHYSICAL CHANNELS (ONE 2870 MULTIPLEXER CHANNEL AND UP TO TWO 2860 SELECTOR CHANNELS)
4. ONE 2167 CONFIGURATION UNIT

IBM 2860 SELECTOR/2870 MULTIPLEXER CHANNELS:

DATA IS TRANSFERRED IN ONE BYTE WIDTH BETWEEN I/O DEVICES AND A CHANNEL AND IN EIGHT BYTE WIDTHS BETWEEN CHANNELS, CHANNEL CONTROLLERS, AND STORAGE UNITS. A 2067-2 MAY HAVE UP TO 11 CHANNELS, 7 ON EACH OF 2 CHANNEL CONTROLLERS.

2860 SELECTOR CHANNEL:
SUPPORTS RATES UP TO 1.3 MILLION BYTES PER SECOND;
CHANNELS OPERATE WITH MINIMAL INTERFERENCE DUE TO OWN REGISTERS;
UP TO 8 CONTROL UNITS PER CHANNEL, WITH AT MOST ONE ACTIVE AT ONCE.

2870 MULTIPLEXER CHANNEL:
AT MOST ONE 2870 MAY BE ATTACHED TO EACH CHANNEL CONTROLLER;
BASIC MULTIPLEXER CHANNEL:
UP TO 192 SUBCHANNELS;
CAN ATTACH UP TO 8 CONTROL UNITS AND ADDRESS UP TO 192 I/O DEVICES;
CAN OVERLAP OPERATION OF SEVERAL I/O DEVICES IN MULTIPLEX MODE OR OPERATE SINGLE DEVICE IN BURST MODE;
AGGREGATE DATA RATE IS 110 KB.
AUGMENTED MULTIPLEXER CHANNEL:
UP TO 4 SELECTOR SUBCHANNELS MAY BE ADDED, FOR A TOTAL OF 196;
EACH SELECTOR SUBCHANNEL MAY ATTACH UP TO 8 CONTROL UNITS AND ADDRESS UP TO 16 I/O DEVICES;
SELECTOR SUBCHANNELS 1-3 HAVE MAXIMUM DATA RATES OF 180 KB, WHILE THE FOURTH SELECTOR SUBCHANNEL HAS A MAXIMUM DATA RATE OF 100 KB;
SELECTOR SUBCHANNELS MAY OPERATE CONCURRENTLY WITH EACH OTHER AND WITH THE MULTIPLEXER CHANNEL, BUT MAY OPERATE AT MOST ONE DEVICE AT A TIME;
ADDITION OF SELECTOR SUBCHANNELS REDUCES THE MAXIMUM DATA RATE OF THE BASIC MULTIPLEXER SUBCHANNEL, BUT INCREASES THE OVERALL DATA RATE OF THE MULTIPLEXER CHANNEL.
CHANNEL ADDRESSES AND PRIORITIES:
CHANNEL ADDRESSES ON EACH CHANNEL CONTROLLER:
0 2870 MULTIPLEXER CHANNEL
1-6 2860 SELECTOR CHANNELS
MUST USE CONSECUTIVE ADDRESSES STARTING AT 0 OR 1
COMPLETE I/O ADDRESS IS 16 BITS IN LENGTH AS FOLLOWS:
0-3 CHANNEL CONTROLLER ADDRESS (00, 01)
4-7 CHANNEL ADDRESS (00-06)
8-15 DEVICE ADDRESS ON CHANNEL:
ON 2860: 00 THROUGH FF
ON 2870:
00-EF ON BASIC MULTIPLEXER CHANNEL
00-CF ON SELECTOR SUBCHANNEL 1
00-DF ON SELECTOR SUBCHANNEL 2
00-EF ON SELECTOR SUBCHANNEL 3
00-PF ON SELECTOR SUBCHANNEL 4
CHANNEL CONTROLLER SERVICES ATTACHED CHANNELS ACCORDING TO THEIR ADDRESSES
IN THE PRIORITY ORDER OF 01, 02, 03, 04, 05, 06, 00; STORAGE UNIT
SERVICES CHANNEL CONTROLLERS IN ORDER OF ADDRESSES 00, 01; CPUs
SERVICED WHEN CHANNELS NOT REQUESTING SERVICE
DURING PROCESSING, ACTUAL DATA RATES DEPEND UPON:
1. CHANNEL PRIORITY
2. NUMBER OF CHANNELS OPERATING CONCURRENTLY
3. SPEED OF THE I/O DEVICES OPERATING CONCURRENTLY ON THE CHANNELS
4. TYPE OF CHANNEL PROGRAMMING USED

IBM 2365-12 PROCESSOR STORAGE:

UP TO 8 2365-12s MAY BE ATTACHED TO A 2067-2 SYSTEM;
2365 HAS BASIC STORAGE CYCLE OF 750 NANOSECONDS AND ACCESSES 8 BYTES IN PARALLEL;
EACH 2365 CONTAINS 64 PAGES = 256K BYTES = 262,144 BITS OF STORAGE HOUSED IN TWO
INDEPENDENT STORAGE ARRAYS OF 32 PAGES EACH; ALL EVEN DOUBLE WORDS ARE IN
ONE ARRAY AND ALL ODD DOUBLE WORDS ARE IN THE OTHER ARRAY;
EACH 2365-12 CAN ATTACH TO UP TO FOUR STORAGE BUSSES: ONE FROM EACH CPU AND ONE
FROM EACH CHANNEL CONTROLLER IN THE DIPLEX SYSTEM;
AT EACH 2365, THE CHANNEL CONTROLLER CLOSEST TO IT GETS FIRST PRIORITY, AS DOES
THE CPU CLOSEST TO IT; THEREFORE, PRIORITIES AT EACH STORAGE UNIT MAY NOT
BE THE SAME ORDERING OF CGs AND CPUs;
INTERLEAVING OF SUCCESSIVE DOUBLE WORD REQUESTS FROM A GIVEN STORAGE UNIT IS
POSSIBLE, REDUCING ACCESS TIME TO 375 NANOSECONDS IN SUCH CASES, AND
DEPENDING UPON OTHER CONFLICTING REQUESTS;
STORAGE KEY IS 7 BITS IN LENGTH AND COVERS 2048 BYTE BLOCKS:
0-3 STANDARD STORE PROTECTION KEY
4 STANDARD FETCH PROTECTION BIT
5 REFERENCE BIT: SET TO 1 EACH TIME THE CORRESPONDING STORAGE BLOCK
   IS REFERENCED (STORE OR FETCH)
6 CHANGE BIT: SET TO 1 EACH TIME THE CORRESPONDING STORAGE BLOCK
   RECEIVES A STORAGE REFERENCE (I.E., IS CHANGED)
IBM 2167 CONFIGURATION UNIT:

2167 IS REQUIRED IN A 2067-2 CONFIGURATION AND PROVIDES MANUAL SWITCHES FOR THE FOLLOWING FUNCTIONS:

1. PARTITIONING: ONE SWITCH FOR EACH CORE STORAGE UNIT INTERFACE, ONE FOR EACH CPU INTERFACE TO A CHANNEL CONTROLLER, AND ONE FOR EACH CHANNEL INTERFACE ON DUAL-CHANNEL-INTERFACE I/O CONTROL UNITS;

2. PREFIX ACTIVATION: ONE SWITCH PER CPU;

3. DIRECT CONTROL ACTIVATION: ONE SWITCH PER CPU;

4. PROCESSOR STORAGE UNIT FLOATING ADDRESSING: ROTARY SWITCH FOR EACH 2365-12 STORAGE UNIT IN THE SYSTEM; SWITCH PICKS THE STARTING CORE ADDRESS FOR THE UNIT; EACH UNIT CONTAINS 256K CONSECUTIVE BYTE ADDRESSES STARTING WITH THE ONE INDICATED BY THE UNIT'S SWITCH; TOTAL SET OF ADDRESSES IN THESE SWITCHES MUST BE CONSECUTIVE STARTING WITH ZERO;

5. CHANNEL CONTROLLER FLOATING ADDRESSING: ONE ROTARY SWITCH FOR EACH CPU; MAY BE USED TO ASSIGN THE CHANNEL CONTROLLER TO BE USED BY THE CPU IN STANDARD PSW MODE.

STATUS OF SWITCH SETTINGS MAY BE SENSED BY EITHER CPU BY STORING THE CONTROL REGISTERS; CONTROL REGISTERS 8-11 CONTAIN THE STATUS BITS REFLECTING THE SWITCH SETTINGS. ALSO, BIT 9 OF CONTROL REGISTER 6 MAY BE USED TO PREVENT SWITCH CHANGES EXCEPT WHEN ACCEPTABLE TO THE CONTROL PROGRAM.
INSTRUCTIONS MODIFIED UNDER EXTENDED PSW MODE:

WHEN BIT 8 OF CONTROL REGISTER 6 IS 1, THE CPU OPERATES IN EXTENDED PSW MODE.
IN THIS MODE, SOME INSTRUCTIONS ARE MODIFIED IN THEIR EXECUTION, AS FOLLOWS:

**BAI, BALR**
IF DAT IS ON, THE RIGHT-MOST 24 BITS OF THE VIRTUAL ADDRESS ARE
PLACED IN THE SPECIFIED REGISTER, AND THE FIRST BYTE OF THE REGISTER
IS SET TO THE ILC, CC, AND FM OF THE PSW

**BXH, EXLE**
IN 32 BIT DAT, AN ADDRESS IN ONE OF THE AFFECTED REGISTERS MAY
APPEAR TO BE A NEGATIVE NUMBER RATHER THAN A POSITIVE ADDRESS

**EDMK**
IN 32 BIT DAT, THE ADDRESS INSERTED INTO GPR 1 IS A FULL 32 BIT
VIRTUAL ADDRESS

**ISK**
A 7 BIT STORAGE KEY IS TAKEN FROM BITS 24-30 OF THE SPECIFIED
GENERAL REGISTER

**LA**
A VIRTUAL ADDRESS IS PLACED IN THE SPECIFIED GENERAL REGISTER;
IN 24 BIT DAT, THE FIRST BYTE OF THE REGISTER IS ZERODED; IN 32
BIT DAT, THE FULL 32 BIT ADDRESS IS PLACED IN THE REGISTER

**LPSW**
THE DOUBLE WORD LOADED AS THE NEW PSW MUST CONFORM TO THE FORMAT
OF THE EXTENDED PSW

**SSK**
THE STORAGE KEY IS PLACED INTO BITS 24-30 OF THE SPECIFIED GENERAL
REGISTER

**SSM**
SETS BITS 0-7 OF THE CURRENT PSW; THIS IS NOT THE SYSTEM MASK,
HOWEVER, IN EXTENDED PSW MODE, CAN BE USED TO SET THE DAT ADDRESS
MODE (BIT 4), DAT CONTROL (BIT 5), I/O SUMMARY MASK (BIT 6), AND
EXTERNAL SUMMARY MASK (BIT 7)

**SVC**
THE IMMEDIATE FIELD FROM THE SVC IS STORED IN LOCATIONS 10-11
(HEX) INSTEAD OF WITH THE OLD PSW

**TRT**
IN 32 BIT DAT, THE ADDRESS INSERTED INTO GPR 1 IS A FULL 32 BIT
VIRTUAL ADDRESS

NEW INSTRUCTIONS PROVIDED ON THE MODEL 67:

**BASR**  \( R_1, R_2 \)
IF IN 32 BIT DAT, THE FULL 32 BIT VIRTUAL ADDRESS OF THE
NEXT INSTRUCTION IS PLACED IN \( R_1 \). OTHERWISE, A 24 BIT
REAL OR VIRTUAL ADDRESS IS PLACED IN \( R_1 \) AND THE FIRST BYTE IS ZERODED. THEN BRANCHING OCCURS.

**BAS**  \( R_1, D_2(\times_2, B_2) \)

**IMC**
\( R_1, R_3, D_2(B_2) \)
PROVIDES MULTIPLE CONTROL REGISTER LOAD/STORE. REGISTERS
LOADED FROM/STORED INTO CONTIGUOUS FULL-WORDS STARTING
AT THE SPECIFIED ADDRESS. REGISTER WRAP-AROUND WITH 0
FOLLOWING 15. CRs 8-14 CANNOT BE LOADED. POSSIBLE
PROGRAM EXCEPTIONS INCLUDE M, A, S, AND P.

**STMC**
\( R_1, D_2(\times_2, B_2) \)
COMPUTES THE REAL ADDRESS (TRANSLATED ADDRESS) CORRESPO
TO THE SECOND OPERAND AND PLACES IT IN THE SPECIFIED
GENERAL REGISTER. CONDITION CODE IS SET:
0 SUCCESSFUL TRANSLATION
1 PTA = 1 OR SEGMENT NUMBER TOO LARGE
2 PA = 1 OR PAGE NUMBER TOO LARGE
TRANSLATION EXCEPTIONS ARE MASKED DURING AN IRA.
POSSIBLE PROGRAM EXCEPTIONS ARE M, A, AND S.
The following pages are excerpts from the MTS Manual, Volume 3. We have provided here copies of that Volume 3 information of greatest use in each of the two courses. This does not include all possible information of interest to the courses, but the need to reference the MTS manual should be considerably reduced. Needless to say, students are encouraged to peruse Volume 3, and other MTS manual volumes, to gain a better understanding of the many facilities provided by this system.
USING_SUBROUTINE_LIBRARIES

The Computing Center maintains a number of subroutine libraries in public files. In addition the user can construct and use his own libraries.

The two most common libraries, *LIBRARY and LCSYMBOL (the resident system subroutines), are automatically searched. If, after the user's explicitly specified object modules are loaded, unresolved external symbols remain, the system loader scans *LIBRARY and LCSYMBOL in an attempt to resolve them. This automatic library search can be prevented by issuing the command

SET LIBS=OFF

The other available subroutine libraries are not automatically searched to resolve undefined symbols. To be used, they must be concatenated onto the object file name when the program is run. For example, to use SSP (IBM's Scientific Subroutine Package) with a FORTRAN program that has just been compiled, one might specify

$RUN -LOAD##SSP

In this case the loader goes through the following steps:

1. The object modules in the file -LOAD# are loaded and linked together.

2. Then object modules are selectively loaded from SSP (since it is a library) to resolve external symbols (i.e., subroutine names) from -LOAD#.

3. Finally, if there are still unresolved external symbols, *LIBRARY and LCSYMBOL are searched for the appropriate object modules.

Note that this concatenation can be implicit as well as explicit. Instead of saying

$RUN OBJ##SSP

one could put

$CONTINUE WITH *SSP

in OBJ as the last line and then merely say

$RUN OBJ

to get the same effect.
The following public files contain subroutine libraries:

**LIBRARY**

All except the standard FORTRAN mathematical routines as described in this volume. See "IBM System/360 FORTRAN II Library: Mathematical and Service Subprograms", form C28-6410, for descriptions of the others.

**SSP**
**SSPMATH**
**SSPSSTAT**

**SSPMATH** and **SSPSSTAT** contain the mathematical and statistical routines, respectively, of the Scientific Subroutines Package, and **SSP** contains the one line

**SCONTINUE WITH **SSPMATH**SSPSSTAT**


Some of the functions performed by **SSP** modules are duplicated by subroutines available in **LIBRARY**. In general the **LIBRARY** versions are both faster and smaller and are therefore highly recommended for use in preference to the corresponding **SSP** subroutines. See the description of these subroutines in this volume.

**PL11IE**

This contains subroutines that may be invoked by PL/1 compilation code. For details, see "IBM System/360 CS PL/1 [?] Programmer's Guide", form C28-6594, and "IBM System/360 CS PL/1 Library Computational Subroutines", form C28-6590.

**PL360LIP**

This contains subroutines to support the external procedures READ, WRITE, PUNCH, and PAGE for PL360 programs. Further information may be found in the "PL360 User's Guide" in Volume 7 of the MTS manual.

**SLIP**

The SLIP (Symmetric List Processor) subroutine package is an implementation of Joseph Weizenbaum's IBM 7090 SLIP language. A complete description of the language may be found in the *Communications of the ACM*, 1963, vol. 6, pp. 534-544. See the "SLIP User's Guide" in Volume 6 of the MTS Manual.
CONSTRUCTING A SHERGUTHINE LIBRARY

The dynamic loader's library facility consists of four control records, namely LCS, LIB, RIP, and DIR records (named because the records have LCS, P, RIP, or DIR, respectively in columns 2 to 4 of the record). The LCS record causes symbols which are referenced but not yet defined to be defined in an in-core table if they exist there. The LIB record loads selectively a object module which follows it or to which the LIB record points only if the module name has been referenced but not yet defined. The RIP record holds forward references and multiple entry point problems in the one-pass binary scan. The DIR record is used to facilitate the loading of modules stored in a sequential file.

A library consists of the object modules the user desires in his library together with the LIB and RIP control records necessary to define the module name, entry points, and references for the selective loading feature of the loader. Although the user can construct such a library himself by inserting appropriate LIB and RIP records in both his object modules, this task has been formidable enough with large libraries that a program has been written to analyze the object modules for a library and generate the library file with all LIB and RIP records. A description of this program, SHERG, is given in Volume 2 of the UGS manual. Details of the use of E, RIP, and DIR records can be found in "The Dynamic Loader" section in June 5.
*GENLIB*

Contents: The object module of the library-generating program.

Purpose: To generate a loader library from object modules.

Usage: The program is invoked by the $RUN command.

Logical I/O Units Referenced:
- SCARDS - input object modules and/or libraries.
- SECOM - listing of diagnostics for conversational users.
- SPRINT - listing of diagnostics and library modules.

Parameters: Only the first character of each parameter need be given. Only one of the parameters SEQL, POINT, DIR, and LIB can be specified. The parameters must be separated by either a comma or a blank.

- XREF - Specifies that a complete cross-reference listing of load modules be produced. This will consist of three parts: load modules with entry points (if any), load modules with external references, and external references with load modules.

- NCXREF - Specifies that LIB records are to have no pointers so that the loader must proceed in the sequential order. The output library may be any kind of file or output device such as *PUNCH*, or magnetic tapes. The first record will be a COM (comment) record with the date and time indicated. Each load module follows its appropriate RIP records and LIB record. The parameter SEQL should be specified only for output devices, since the loader must process every record, and hence takes more time than for other types of libraries.

- SEQL - Specifies that LIB records are to have note-point information. The loader proceeds in sequential order, but skips over unneeded load modules without reading them. The output library must be a sequential file. POINT should be specified when a program loads many load modules from the library.

- POINT - Specifies that a DIR (directory) record be constructed. This record consists of 12-byte entries, an 8-byte defined symbol name and its 4-byte pointer referring to its first record in the library. Since the DIR record can have no more than 32767 bytes, there can be no more than
2730 different defined symbol names. Library load modules follow the LIR record. The DIR record has a pointer pointing to immediately after the last load module. The loader does not proceed in sequential order, but picks load modules as needed from the library. The output library must be a sequential file. The DIR parameter should be specified when only a few load modules need to be loaded from the library.

LINE - Specifies that LIE records are to have line number pointers. The output library must be a line file. The line number zero will be date-started. The necessary RIP and LIE records follow. Load modules start at the line number 99999; for each successive record, the line number is incremented by 1.00. The LINE parameter has two advantages: (1) it is generally fast when loading from a small library, (2) one can easily insert patches into the library. If many modules are to be loaded, the loading time is slow since the loader must use numerous indexed read operations on the file.

Examples:
$RUN *GENLIB SCARDS=SQUASH SPUNCH=LIR
$RUN *GENLIB SCARDS=A+B+C SPUNCH=SKLIB PAR=XREF,POINT
$RUN *GENLIB SCARDS=OLDLIB SPUNCH=NEWLIB PAR=D

scription:
Each input load module is analyzed for errors. Each module must have at least one ESL record, and the last record must be an END record. It also must have at least one defined symbol. The library module name will be the one defined on the previous LIR record, if any, or the first defined non-blank symbol in the module. If this library module name is a duplicate of an earlier name, the entire load module is skipped. If the module satisfies the above conditions, it is saved in the sequential file -SYSUT4. When all of the input has been processed, a complete cross-reference listing is produced if the XREF parameter is specified. Any duplicate or undefined symbols are printed. Then, the output library is produced, each load module copied intact from -SYSUT4. Any remaining records (such as LCS or LT records) immediately after the last load module in the input from SCARDS are added. The program then lists all library module names with the corresponding line numbers (if applicable) on which they begin. Finally, the processing time taken (in seconds and hundredths of seconds) is printed.

See "The Dynamic Loader" section in Volume 5 for the description of the DIR, RIP, and LIE loader records.

its:
(1) The output library file should be emptied before *GENLIB is run.
(2) To generate a new library from an old library of type LINE, use the following:

$RUN *GENLIB SCARDS=OLDLIB(-99999,-1) SPUNCH=NEWLIB

This reduces some of the indexed operations on OLDLIB and, thus, reduces the processing time in generating the new library.

(3) All libraries of type LIR and POINT must be copied as

$COPY OLDLIB=TRIM NEWLIB=TRIM

Both files must be sequential files.

(4) All libraries of type LINE must be copied as

$COPY OLDLIB(-99999) NEWLIB=!
CALLING CONVENTIONS

INTRODUCTION

A calling convention is a very rigid specification of the sequence of instructions to be used by a program to transfer control to another program (usually referred to as a subroutine). It is very desirable although not always practical to set up only one set of conventions to be used by all programs no matter what language they are written in so that FORTRAN programs may call MAD programs and assembly language programs and so forth. In the MTS system the OS type I calling conventions have been adapted as the standard. A complete specification of these standards can be found in the "IBM System/360 Operating System Publication, Supervisor and Data Management Services", form C28-6646. This description shall try to bring out pertinent details of these calling conventions.

Throughout this discussion we will refer to the terms calling program, called program, save area, and calling sequence. The calling program is the program which is in control and wants to now call another program (subroutine). The called program is the program (subroutine) which the calling program wants to call. The save area is an area belonging to the calling program which the called program uses to save and later restore general purpose registers. The save area has a very rigid format and is discussed in more detail later on. A calling sequence is the actual sequence of machine instructions which perform the tasks as specified by the calling conventions.

The facilities that must be provided by the calling conventions are:

1. Establish addressability and transfer to the entry point.
2. Pass parameters on to the called program.
3. Pass results back to the calling program.
4. Save and restore general purpose and floating point registers.
5. Re-establish addressability and return to the calling program.
6. Pass a return code (error indication) back to the calling program so it knows how things went.

The remainder of this description will describe the OS type I calling conventions to show how they are used and how the facilities listed above are provided for.

REGISTER AND STORAGE VARIANTS OF TYPE I CALLS

The OS type I calling conventions actually consist of two very similar calling conventions, referred to as OS (I) S Type calling conventions and OS (I) R Type calling conventions. The two differ only in the way parameters
and results are passed between the calling and called programs. The R refers to register and the S to storage.

The OS (I) R type calling conventions utilize the general purpose registers 0 and 1 for passing parameters and results. This allows only two parameters or results and cannot be generated in higher level languages as FORTRAN. Its advantages are that calling sequences are shorter and take less time to set up. These are very popular in lower-level system subroutines such as GETSPACE or GETFD. Fortran users needing to call subroutines that utilize R-type calling conventions can use the RCALL subroutine described in this volume.

The OS (I) S Type calling conventions require a pointer to a vector of address constants called a parameter list (in register 1). Since the parameter list can be of any required length, several parameters can be passed using OS (I) S Type calling convention. These conventions are used by system subroutines such as SCARDS or LINK and are generated by all function or subprogram references in FORTRAN. Results can be passed back by giving variables in the parameter list new values or via register 0.

PARAMETER LISTS

As stated above, a parameter list is a vector of address constants. The parameter list must be on a full-word boundary and the entries are each four bytes long. The address of the first parameter is the first word of the list, the address of the second parameter the second word of the list, and so on. For example the parameter list for the FORTRAN statement

\[
\text{CALL QSIV}(X,Y,Z)
\]

might be written in assembly code as:

\[
\begin{align*}
\text{PAR} & \quad \text{DC} & \quad A(X) & \quad \text{address of } X \\
& \quad \text{DC} & \quad A(Y) & \quad \text{address of } Y \\
& \quad \text{DC} & \quad A(Z) & \quad \text{address of } Z
\end{align*}
\]

Now this parameter list works well enough when the parameter list for the subroutine is of fixed length, but there is not enough information yet to allow a subroutine to determine the length of the parameter list and hence accept variable length parameter lists. For this reason there are two types of parameter lists, fixed length parameter lists as described above, and an extended form of parameter list called a variable-length parameter list which is described next.

Since a standard 360 computer uses 24 byte storage addresses the left-most byte of an address constant is usually zero. In a variable length parameter list bit zero of the left-most byte of the last parameter address constant is set to 1 to show that it is the last item in the list. The example above then would be written as:

\[
\begin{align*}
\text{PAR} & \quad \text{DC} & \quad A(X) & \quad \text{address of } X \\
& \quad \text{DC} & \quad A(Y) & \quad \text{address of } Y
\end{align*}
\]
DC XL'80' turn on bit zero.
DC AL3(Z) address of Z

if it generated a variable-length parameter list. As a matter of fact
FORTRAN does generate variable-length parameter lists. Note though that
programs expecting a fixed length parameter list will work with a variable-
length parameter list, provided it is at least as long as the fixed-length
list they are expecting, since they extract only the address part when they
use the parameters.

REGISTER ASSIGNMENTS

Of the sixteen general purpose registers, five are assigned for use in
the calling conventions. The use of the general registers differs slightly
depending upon whether an R or S type call is being made.
The following table specifies exactly what each register is used for during a call:

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Parameter to be passed in R type sequences.</td>
</tr>
<tr>
<td></td>
<td>Result to be passed back in R and S type sequences.</td>
</tr>
<tr>
<td>1</td>
<td>Parameter to be passed in R type sequences.</td>
</tr>
<tr>
<td></td>
<td>Address of a parameter list in S type sequences.</td>
</tr>
<tr>
<td>2-12</td>
<td>Not used as a part of the calling sequence. Must be saved and restored by the called program. The save area is usually used for this.</td>
</tr>
<tr>
<td>13</td>
<td>The address of the save area provided by the calling program to be used by the called program.</td>
</tr>
<tr>
<td>14</td>
<td>Address of the location in the calling program to which control should be returned after execution of the called program.</td>
</tr>
<tr>
<td>15</td>
<td>Address of the entry point in the called program at the time of the call.</td>
</tr>
<tr>
<td></td>
<td>A return code at the time of the return that indicates to the calling program whether or not an exceptional condition occurred during processing of the called program. The return code should be zero for a normal return or a multiple of four for various exceptional conditions.</td>
</tr>
</tbody>
</table>

General Purpose Register Linkage Conventions

Notice that it is the called program's responsibility to save and restore registers 2-12 in the save area provided by the calling program. There are two reasons for this. First of all only the called program knows how many of the registers from 2-12 it is going to use. Since a register need be saved and restored only if it is actually going to be changed, the called program may be able to save some time by saving and restoring only those registers which it will use. Secondly, the called program requires addressability over the area in which it will save registers upon entry, since any attempt to acquire the address of a save area would destroy some of the registers which are to be saved. Furthermore, the save area should not be a part of the called program since that would prevent it from being re-entrant (shareable). This means the calling program should provide the save area in which registers are saved and restored. And so we have the
called program saving and restoring registers 2-12 in a save area provided by the calling program.

The calling conventions are quite different with floating point registers. Since a large percentage of programs do not leave items in floating point registers across subroutine calls it seems rather wasteful to always save and restore the floating point registers. So the convention has been established that the calling program must save and restore those floating point registers which contain items which are wanted. Also, programs which return a single floating point result quite frequently do so via floating point register 0.

RETURNING RESULTS

There are in the OS Type I calling conventions four ways in which a subroutine can return a result. These are:

1. Value of result in general purpose register 0.
2. Value of result in general purpose register 1.
3. Value of result in floating point registers. (usually 0)
4. Value of a parameter from the parameter list changed.

The particular method used depends upon whether the R or S type convention is used and whether the called program can be used as a function in arithmetic statements.

The first three methods are used by R type calling conventions for all returned results. The contents of each of the registers depends upon the particular called program and are described in the subroutine description for each subroutine using the R type calling conventions.

The first, third, and fourth methods are used by S type calling conventions for all returned results. The first and third methods are used by function subprograms whose calls can be embedded in FORTRAN and MAD statements. The choice of general register 0 or floating point register 0 depends upon whether the result returned is integer or floating point mode, respectively. An example of subroutines which return results in this manner are the FORTRAN IV Library Subprograms, such as EXP, ALOG, or SIN. The fourth method can be used by a subprogram. An example would be a subprogram called by the statement

```
CALL MATADD(A,B,C,H,N)
```

which might add the MxN matrices A and B together and store the result in C.

SAVE AREA FORMAT

The save area is an area belonging to the calling program which the called program uses to save and later restore general purpose registers. The address of the save area is passed to the called program by the calling
program via general purpose register 13. The save area has a very rigid format and is described in the table:

<table>
<thead>
<tr>
<th>Word</th>
<th>Displacement</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Used by FORTRAN, PL/I, and other beasties for many devious purposes. Don't touch!</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Address of the save area used by the calling program. Forms a backward chain of save areas. Stored by calling program.</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Address of the save area provided by the called program for programs it calls. Forms a forward chain of save areas.</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>Return address. Contents of register 14 at time of call.</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>Entry point address. Contents of register 15 at time of call.</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>Register 0 contents.</td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td>Register 1 contents.</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>Register 2 contents.</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>Register 3 contents.</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
<td>Register 4 contents.</td>
</tr>
<tr>
<td>11</td>
<td>40</td>
<td>Register 5 contents.</td>
</tr>
<tr>
<td>12</td>
<td>44</td>
<td>Register 6 contents.</td>
</tr>
<tr>
<td>13</td>
<td>48</td>
<td>Register 7 contents.</td>
</tr>
<tr>
<td>14</td>
<td>52</td>
<td>Register 8 contents.</td>
</tr>
<tr>
<td>15</td>
<td>56</td>
<td>Register 9 contents.</td>
</tr>
<tr>
<td>16</td>
<td>60</td>
<td>Register 10 contents.</td>
</tr>
<tr>
<td>17</td>
<td>64</td>
<td>Register 11 contents.</td>
</tr>
<tr>
<td>18</td>
<td>68</td>
<td>Register 12 contents.</td>
</tr>
</tbody>
</table>

Save Area Format

There are two things to be noted about the save area format, namely who sets what parts of the save area and how these areas might be set up. The
calling program is responsible for setting up the second word of the save area. This is to contain the address of the save area which was provided when the calling program was itself called. Although this is technically set up by the calling program as a part of the call, most programs set up the save area they will provide to subroutines they call once and leave its address in general register 13. The work then does not need to be repeated for each call. The called program is responsible for setting up the third through eighteenth words of the save area. The called program usually saves the general registers which it will use as a part of its initialization procedure and restores the registers as a part of the return procedure. Notice that the save area format is amenable to use of the store multiple and load multiple instructions for saving and restoring blocks of registers. All of this will be made clearer in the examples at the end.

Some system subroutines (notably GETSPACE, FREESPAC, and a few others) do not require that a save area be provided for them. For these subroutines general register 13 need not be set up before a call and its contents are preserved by the called subroutine. The subroutines which need no save area are clearly marked as such in the MTS subroutine descriptions. Notice that it is all right to provide a save area to one of these subroutine; it will simply be ignored.

CALLING PROGRAM RESPONSIBILITIES AND CONSIDERATIONS

The calling program is responsible for the following:

1. Loading register 13 with the address of the save area and setting up the second word of the save area.
2. Loading register 14 with the return address.
3. Loading register 15 with the entry point address.
4. Loading registers 0 and 1 with the parameters in an R type call or loading register 1 with the address of the parameter list in an S type call.
5. Saving floating point registers, if necessary.
6. Transferring to the entry point of the subroutine.
7. Restoring floating point registers, if necessary.
8. Testing the return code in register 15, if desired.

After the return from a subroutine, the status of the program will be as follows:

1. In general, the contents of the floating point registers will be unpredictable unless saved and restored by the calling program.
2. The contents of general registers 2 through 14 will be restored to their contents at the time the called program was entered.
3. The program mask will be unchanged.
4. The contents of general registers 0, 1, and 15 may be changed.
5. The condition code may be changed.

Note that general registers 0 and 1 and floating point register 0 may contain results in the case of R type subroutine calls or a function subprogram. General register 15 will normally contain a return code.
indicating whether or not an exceptional condition occurred during processing of the called program.

CALLED PROGRAM RESPONSIBILITIES AND CONSIDERATIONS

The called program is responsible for the following:

1. Saving the contents of general registers 2 through 14 in the save area provided by the calling program. These registers need be saved only if the called program modifies these registers.
2. Setting up the third word of the save area with the address of the save area which will be provided to subroutines it will call.
3. Restoring the contents of general registers 2 through 14 before returning to the calling program.
4. Restoring the program mask if changed.
5. Loading general registers 0 and 1 or floating point register 0 with the result in the case of a type subroutine calls of a function subroutine.
6. Loading general register 15 with the return code.
7. Transferring to the return location.

EXAMPLE CALLING SEQUENCES

This section will describe and give the assembly language statements for the typical machine instructions necessary to implement OS Type I calling conventions.

A typical entry point might consist of the following statements:

```
USING SUBRA, 12

SUBRA STM 14, 12, 12 (13) 12 will be a base register
           LR   12, 15        save registers
           LA   11, SAV3      set up 12 as the base register
           ST   11, (0, 13)   this is save area provided for others
           ST   13, (0, 11)   set up forward pointer
           LR   13, 11      set up backward pointer
           LR   11, 1       get parameter pointer into non-volatile reg.

SAVE DS   18F      save area we provide for others

Inside a subroutine that began with the entry sequence given above, the value of the second parameter in the parameter list could be put into general purpose register 3 with the following sequence:

L    3, (0, 11)    pick up second adcon from parameter list
```
L 0.0(0,3)  pick up value of parameter

Inside a subroutine that began with the entry sequence given above, another subroutine, SUBRS, could be called using the following sequence. Remember that register 13 already points to the correct save area:

LA 1,PARLIST set up parameter list address
L 1b,=V(SUBRS) set up entry point address
SALR 14,15 set up return address and branch to the subroutine
B +s(15) test return code via a transfer table
B ACK RC=0
B BAD1 RC=4
B BAD2 RC=8

ACK normal return to here

PARLIST DC A(PAR1) first parameter address

Finally, a subroutine that began with the entry sequence given above could return to the program that called it with the following sequence:

LEV 0,RESULT floating point result to FPR 0.
L 13,4(0,13) use back pointer to get right save area.
LM 14,12,12(13) restore registers.
SR 15,15 indicate a zero return code (no errors).
BR 14 return to what called us

It should be pointed out that although the above sequence are typical of the instructions used to implement the calling conventions, many variations are possible.

MACROS FOR CALLING SEQUENCES

There are two sets of macro definitions in the MTS Macro Library which can be used to help generate calling sequences. These are the macros SAVE,
CALL, and RETURN; and the macros ENTER and EXIT. The more useful of these macros are ENTER, CALL, and EXIT. Besides these there is a set of macros which generate the entire calling sequences for many of the system subroutines and IOH/360. For details, see the macro descriptions in this volume.
The return codes that may result from a call on an input or output subroutine depend on the type of the file or the device used in the operation. In general, a return code of zero means successful completion of the input or output operation, and a return code of 4 means end-of-file for an input operation and end-of-file-or-device for an output operation. If the file or device being used was specified as part of an explicit concatenation (and is not the last member of that concatenation), a return code of 4 will cause progression to the next element of the concatenation, and that return code will not be passed back to the caller. Thus, for example, if

\[
\text{SCARDS} = A \times B
\]

then when the call is made to the SCARDS subroutine after the last line in A has been read, the file routines will signal an end-of-file, but this will be intercepted, and the first line in B will be read instead.

Return codes greater than 4 are normally not passed back to the caller but instead cause an error comment to be printed and control to be returned to command mode. There are two ways to suppress this action and gain control in this situation. First, the subroutines SETIOERR and SIGERR (see descriptions in this volume) are provided to permit a global intercept of all input/output errors. Secondly, specifying the ERRATN modifier on an I/O subroutine call will cause all return codes to be passed back.

A description of the return codes that may occur with a particular file or device will be found in the appropriate User's Guide. In addition, a summary is provided below:

**Files:**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful return</td>
</tr>
<tr>
<td>4</td>
<td>End-of-file (sequential read)</td>
</tr>
<tr>
<td></td>
<td>Line not in file (indexed read)</td>
</tr>
<tr>
<td>8</td>
<td>Error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful return</td>
</tr>
<tr>
<td>4</td>
<td>Size of file exceeded</td>
</tr>
<tr>
<td>8</td>
<td>Line numbers not in sequence (SEQ#L)</td>
</tr>
<tr>
<td>12</td>
<td>For future expansion, should not occur</td>
</tr>
<tr>
<td>16</td>
<td>For future expansion, should not occur</td>
</tr>
<tr>
<td>20</td>
<td>Sequential file written with indexed modifier, or written with starting line number other than 1</td>
</tr>
<tr>
<td>24</td>
<td>File not in catalog (system error)</td>
</tr>
<tr>
<td>28</td>
<td>Hardware or system error</td>
</tr>
</tbody>
</table>

I/O Routines' Return Codes 357
Magnetic Tape:

Input
0  Successful return
4  End-of-file mark sensed
8  Load point sensed on backspace control command
12 Logical end of labeled tape
16 Permanent read error (tape positioned past bad block) or improper control command
20 Should not occur
24 Fatal errors (may be due to equipment malfunctions, fatal label errors in which the position of the tape is uncertain, or pulling the tape off of the end of the reel). The tape must be rewound after any of these errors.
28 Volume and data set label errors (only on labeled tapes if label processing is enabled)
32 Error in sequence of I/O operations or commands
36 Detracking error

Output
0  Successful return
4  End of tape strip sensed
8  Load point sensed on backspace control command
12 Attempt to write more than five records in end-of-tape area
16 Permanent write error or improper control command
20 Attempt to write on file-protected tape
24 Fatal errors (may be due to equipment malfunctions, fatal label errors in which the position of the tape is uncertain, or pulling the tape off of the end of the reel). The tape must be rewound after any of these errors.
28 Volume and data set label errors (only on labeled tapes if label processing is enabled)
32 Error in sequence of I/O operations or commands
36 Blocking error

Paper Tape:

Input
0  Successful return
4  End-of-file
8  End-of-tape
12 Invalid control command
16 Hardware malfunction
Card input under HASP:

- Input 0: Successful return
- Input 4: End-of-file
- Input 8: Attempt to read in column binary mode

Output 8: Attempt to write on card reader

Printed output:

- Input 8: Attempt to read from printer
- Output 0: Successful return
- Output 8: Local page limit exceeded
  (user never gets control back for global limit exceeded)

Punched output:

- Input 8: Attempt to read from punch
- Output 0: Successful return
- Output 8: Local card limit exceeded
  (user never gets control back for global limit exceeded)

Most other devices:

- Input 0: Successful return
- Input 4: End-of-file
- Input 8: Error

Output 0: Successful return
- Output 4: End-of-file-or-device [if applicable]
- Output 8: Error
MODIFIERS

INTRODUCTION

Modifiers are used to modify the action of a specific I/O call or a general I/O usage. Modifiers may be used in I/O subroutine calls (SCARDS, SPRINT, READ, etc), in macro calls setting up the corresponding I/O subroutine calls, in calls to GETFD, or as parts of FNames given in MTS commands. Modifiers control such functions as upper or lower case conversion, logical carriage control, machine carriage control, record trimming, etc.

In general, there are three levels of precedence in the usage of modifiers. The first level of precedence is the modifiers specified on a call to one of the I/O subroutines. If the modifier is not specified by the subroutine call, or when a user generated subroutine call is not relevant (e.g., when a SCOPY command is issued), the second level of precedence, which consists of the modifier name as part of the FName, applies. Note that the group of modifiers which can only control the action of a specific I/O call (for example ERRRTN and NOTIFY) are not valid at this level of precedence. If the action of the modifier is not specified by the second level, the third level of precedence, which consists of the default specifications, applies. The default specification depends upon the type of FName referenced in the I/O call and the settings of global switches. These defaults are given in the explanation of modifier bits below. Modifier specifications given at the first level of precedence will override specifications given at the second and third levels. Modifier specifications given at the second level will override specifications given at the third level. This precedence process is illustrated in the diagram below. Each modifier pair is treated independently in the above precedence process.

Level 1: Subroutine Call Modifiers
Level 2: FName Usage Modifiers
Level 3: Defaults

Effective
Modifiers
The example below illustrates the three levels for controlling the TRIM modifier.

CALL SCARDS(REG,LEN,16384,LNUM) ---->
(16384 specifies -TRIM)
| SCARDS=FYLE@TRIM on $RUN command ...
| Default is TRIM for file ............
| -TRIM

CALL SCARDS(REG,LEN,0,LNUM) ...........
(0 makes no specification)
| SCARDS=FYLE@-TRIM
| Default is TRIM for file ............
| -TRIM

CALL SCARDS(REG,LEN,0,LNUM) ...........
(0 makes no specification)
| SCARDS=FYLE
| Default is TRIM for file ............
| TRIM

The action of the modifiers specified on a subroutine call is controlled by a fullword of modifier bits given as one of the parameters to the subroutine. The action of the modifiers on the subroutine call apply only to that specific call. There are two classes of modifiers.

(1) Bits 0-7 are referenced individually and specify the options for a specific I/O call. If the bit is set, the modifier's action is enabled. If the bit is not set, the default specification is used (which normally means the modifier action is disabled).

(2) Bits 8-31 are referenced in pairs and specify options for a general I/O usage. For each option, one bit is used as an "ON" bit and the other as an "OFF" bit. If either of the bits, but not both, is set, the modifier action is as specified. If neither or both of the bits is set, indicating a "don't care" condition at this level of precedence, the modifier name appended to the Fname is used. If there is no
modifier name appended to the FDname, the default specification is used. The normal programming practice is to leave the modifier bits zero on the subroutine call and apply the modifier names to the FDname referenced unless the program depends upon the modifier bits being set for a specific subroutine call. Here is an example done first in assembly language and then in FORTRAN: FORTRAN is:

```
CALL SCARDS,(REG,LEN,MOD,LNUM)
```

```
REG  DS  20F
LEN  DS  H
MOD  DC X'00004000'  Specifies no trimming of input lines
LNUM DS  F
```

```
INTEGER*2 LEN
DATA MOD/Z00004000/ Specifies no trimming
CALL SCARDS(REG,LEN,MOD,LNUM)
```

Note that if the subroutine call is set up by a macro call, the modifier names rather than the bits are used in the macro parameter list. Thus the above example would become

```
SCARDS  REG,LEN,2-TRIM,LNUM
```

The action of modifiers applied to the FDnames is controlled by the modifier name (preceded by @) appended to the FDname. The action of the modifiers appended to the FDname apply to all I/O calls referencing that usage of the file or device. If the modifier name is preceded with "-" or "-", the other bit of the bit pair is set, which negates the action of the modifier name. [The modifier applies only to the FDname to which it is attached.] If implicit or explicit concatenation to another FDname occurs, the modifiers must be applied to both FDnames even if the FDnames are the same. If the user at a terminal is prompted for an FDname, the full FDname including the modifiers and line number range must be given with each request. The order of modifier names appended to an FDname is unimportant. Some examples are

```
FILE1@IBC
FILE2@-TRIM
*SINK@NOC
*SINK@-CC
RDR@BIN
FILE3@FX@CC(10)
FILE3@FX@CC(1,10)+(20,30)@CC
```

Specifies indexed and upper case  
Specifies no trimming  
Specifies no carriage control  
Specifies no carriage control  
Specifies no BCD conversion  
Specifies prefix and carr control  
Specifies prefix and carr control for lines 1 to 10 and carr control for lines 20 to 30

If the modifier action is also specified on a subroutine call, the modifier action applied to the FDname is overridden.

I/O Modifiers 363
EXPLANATION OF MODIFIERS

The device types discussed below in the exceptions to the default modifier bit specifications are the device types as listed in the description of "Files and Devices" in Volume 1 of the MTS Manual. The device types discussed here are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Modifier</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>SEQUENTIAL, S</td>
<td>1 (dec) 00000001 (hex)</td>
<td>Printers via IBM 2703 Transmission Control</td>
</tr>
<tr>
<td>30</td>
<td>INDEXED, I</td>
<td>2 (dec) 00000002</td>
<td>2741 IBM 2741 and 1050 terminals via IBM 2703</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Transmission Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2260 IBM 2260 Display Unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PLP8 Terminals via the Data Concentrator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9TP 9 Track Magnetic Tape Drive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7TP 7 Track Magnetic Tape Drive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SDA Synchronous Data Adaptor (Remote batch via IBM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Controllers)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HPTH Printed output under HASP</td>
</tr>
</tbody>
</table>

The values indicated below with each bit specification are the values that the modifier word for a subroutine call would have if that modifier option only was specified.

In general, the INDEXED modifier is applied only to line files, while the SEQUENTIAL modifier is applied to line files, sequential files, and all types of devices. Note that the SEQUENTIAL modifier and the sequential file are not directly related. The paragraphs below describe the action of this modifier pair and the results that occur when these modifiers are not used in the normal manner.

With each logical unit (or FDUB) there is a current line pointer which contains the line number of the last record read or written. When an I/O operation is performed, the current line pointer is first set to the line number of the record to be read or written before the actual read or write occurs. After the read or write operation has occurred, the current line pointer will contain the line number of the record last read or written.

I/O operations involving line files may be done with either SEQUENTIAL or INDEXED specified. A SEQUENTIAL I/O operation occurs when the user specifies that the "next" record is to be read or written. For a read operation, "next" means the
record that is next in ascending line number order from the
current value of the line pointer (last line read or written)
of the same logical I/O unit (or FDUS). If, however, an
increment was explicitly given with the PNAME, the line
number read is the current value of the line pointer plus the
first multiple of the specified increment for which there is
a line in the file. For a write operation, "next" means the
current value of the line pointer (last line read or written)
plus the increment specified with the PNAME (defaults to 1)
of the same logical I/O unit (or FDUS). An INDEXED I/O
operation occurs when the user specifies the line number of
the record to be read or written. As an example, consider
the following FORTRAN program segment.

    INTEGER*2 LEN
    DATA MOD/2/    Specifies INDEXED
    1 CALL READ(REG,LEN,0,LNR,2,82)
    CALL WRITE(REG,LEN,MOD,LNR,3)
    GO TO 1
    2 CALL EXIT

This program will perform a read SEQUENTIAL and write INDEXED
using the line numbers from the read operation as the line
number specifications for the write operation. The command
(assuming compilation of the above into -LOAD#)

    $RUN -LOAD# 2=A 3=B

will be equivalent to

    $COPY A B@I

which will copy file A into file B preserving the line
numbers of file A as the line numbers for file B. If a
series of I/O operations involving a given usage of a line
file are interspersed with INDEXED and SEQUENTIAL operations,
the SEQUENTIAL operation will begin sequentially with the
line following the last line specified in the INDEXED
operation. INDEXED operations following SEQUENTIAL opera-
tions will use the line number given in that INDEXED
specification.

I/O operations involving sequential files must be done
SEQUENTIALLY. If the user specifies INDEXED on a sequential
file operation, an error message will be generated unless the
global switch SEQFCHK is OFF, in which case the operation
will be performed as if SEQUENTIAL was specified. Attempting
a sequential operation with a starting line number other than
1 (for example $COPY FYLE(2) ) will also give an error
comment if SEQFCHK is ON.

I/O operations involving sequential devices, such as cari
readers, printers, card punches, magnetic tape units, paper
tape units, and terminals, are inherently sequential and are normally done SEQUENTIALLY. If the SEQUENTIAL modifier is specified, the line number attached to the line is the current value of the line pointer plus the increment specified on the FDname. If the INDEXED modifier is specified, the line number attached to the line is the line number specified in the calling sequence. The INDEXED modifier is used primarily in conjunction with the PREFIX modifier. Note that the device will treat the I/O operation as if SEQUENTIAL were specified.

Bit 29

<table>
<thead>
<tr>
<th>EBCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value: 4 (dec) 00000004 (hex)</td>
</tr>
</tbody>
</table>

Default: EBCD

Exceptions: None

The EBCD/BINARY modifier pair is device dependent as to the action specified. For card readers and punches, the EBCD modifier specifies EBCDIC translation of the card image which means that each card column represents one of the 256 8-bit EBCDIC character codes. The BINARY modifier specifies that the card images are in column binary format which means that each card column represents two 8-bit bytes of information. The top six and bottom six punch positions of each column correspond to the first and second bytes respectively with the high order two bits of each byte taken as zero. The printers and file routines will ignore the presence of this modifier pair.

Other device support routines recognizing this modifier pair are:

1. The Data Concentrator routines
2. The 2703 routines
3. The Paper Tape routines
4. The Audio Response Unit routines

For information on the usage of this modifier pair in specifications involving the devices listed above, see the respective User's Guides in Volume 1 of the MTS Manual. The list of device support routines recognizing this modifier is volatile and subject to change without notice. Users who wish to keep their programs device independent should not specify this modifier.
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

Bit 27  LOWERCASE, LC  Value:  16 (dec) 00000010 (hex)
   CASECONV, UC  32  00000020

Default:  LOWERCASE
Exceptions:  None

The LOWERCASE/CASECONV modifier pair is not device dependent. If the LOWERCASE modifier is specified, the characters are transmitted unchanged. If the CASECONV modifier is specified, lower case letters are changed to upper case letters. This translation is performed in the user’s core region. On input operations, the characters are read into the user’s buffer area and then translated. On output operations, the characters are translated in the user’s buffer area and then written out. Only the alphabetic characters (a-z) are affected by this modifier. [Unlike IBM programming systems, MTS considers the characters  _ , " , and ! as special characters rather than "alphabetic extenders" and thus the UC modifier does not convert _ , " , and ! into @, #, and $, respectively.]

Bit 25  NOCARCNTRL, NOCC  Value:  64 (dec) 00000040 (hex)
   CC, STACKERSELECT, SS  128  00000080

Default:  NOCARCNTRL
Exceptions:  CC for PTR, TTY, 2741, 2260, PDP8, SDA, and NPTA

The NOCC/CC modifier pair is device-dependent. This modifier pair controls the presence or absence of logical carriage control (or stacker selection) on the output of records. For printer and terminal devices, the first character of each record is taken as logical carriage control if this character is a valid carriage control character and if the CC modifier is specified. If the first character is not valid as a carriage control character, the record is written as if NOCC were specified. For further information on logical carriage control, see the "Carriage Control" description in this volume. For card punches, the first character of each card image is taken as the stacker select character if it is a valid logical stacker select character (0, 1, or 2) and if the SS modifier is specified. If the first character is not valid as a stacker select character, the card image is punched as if NOCC were specified. The SS modifier is intended only for those users who are communicating directly with a physical punch (normally system programmers) and is not intended for normal batch usage under HASP. Note that the SS and CC modifiers reference the same modifier bit and thus may be used interchangeably.

I/O Modifiers 367
The Magnetic Tape routines also recognize the presence of this modifier pair. For this description, see the Magnetic Tape User's Guide in Volume 1 of the MTS Manual. The file routines will ignore the presence of this modifier pair.

Bit 23
22  PREFIX, PPX

Value:  256 (dec)  00000100 (hex)
       512  00000200

Default:  -PREFIX
Exceptions:  None

The PREFIX modifier pair controls the prefixing of the current input or output line with the current line number. On terminal input, the current input line number is printed before each input line is requested. The line number used is determined as specified in the description of the SEQUENTIAL and INDEXED modifiers. An example for terminal input is

$COPY *SOURCE*(6,,2)@PPX A(6,,2)
  6_ first input line
  8_ second input line

end of file indicator

Note that this would have the same effect with respect to line numbering as

$GET A
$NUM 6,2
  6_ first input line
  8_ second input line
  
  xx$NUM

The current (prefix) line number is not equivalent to the file line number. In the example above, the prefix line and the file line numbers were explicitly made to correspond by also specifying a line number range on the output PNAME (the file A). On input from card readers and files, the PREFIX modifier has no effect. On terminal output, the current line number is printed before each output line is written. The line number used is determined as specified in the description of the SEQUENTIAL and INDEXED modifiers. An example for terminal output is

$COPY A(1,10) *SINK*(100,,2)@PPX
  100_ first output line
  102_ second output line
Note again that the current line number is not equivalent to the file line number. On output to the printer or to a file, the PREFIX modifier has no effect.

If the INDEXED and PREFIX modifiers are given together for terminal output, the line numbers referenced by the INDEXED modifier will be the same as those produced by the PREFIX modifier. As an example, consider the following FORTRAN program segment:

```
INTEGER*2 LEN
DATA MOD/Z000000202/
   Turns on INDEXED and PREFIX
1 CALL READ(REG,LEN,0,LMR,2,82)
   CALL WRITE(REG,LEN,MOD,LMR,3)
   GO TO 1
2 CALL EXIT
```

This program will perform a read SEQUENTIAL and a write INDEXED and PREFIX. The command (assuming compilation of the above into -LOAD#)

```
$RUN -LOAD# 2=A 3=*SINK*
```

is equivalent to

```
$COPY A *SINK*@BPPFX
```

which is also similar to

```
$LIST A
```

with a slightly different formatting of the line numbers.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1024 (dec)</td>
<td>00000400</td>
</tr>
<tr>
<td>20</td>
<td>2048 (dec)</td>
<td>00000800</td>
</tr>
</tbody>
</table>

Default:   -PEEL
Exceptions: None

The PEEL modifier pair has two functions depending upon whether it is specified on input or on output. On input, if the PEEL (GETLINE#) modifier is specified, a line number is extracted from the front of the current input line. The line number is converted to internal form (external value times 1000) and returned in the line number parameter during the read operation. See the subroutine description of SCARDS and READ. The remainder of the line is moved into the input region specified. As an example, consider the following FORTRAN program segment:
MTS 3  SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

INTEGER*2 LEN
DATA MOD/2048/
1 CALL SCARDS(REG,LEN,MOD,LNR,62) Read with PEEL
CALL SPRINT(REG,LEN,0,LNA)
GO TO 1
2 CALL EXIT

The program will read an input line, extract the line number, and write out the line without its line number. The following sequence (assuming compilation of the above into -LOAD#)

$RUN -LOAD# SCARDS=*SOURCE* SPRINT=ABC
10AAA
12BBB

is equivalent to

$COPY *SOURCE*GETLINE# ABC
10AAA
12BBB

Listing the file ABC will produce

$LIST ABC
1  AAA
2  BBB

If the PEEL modifier is specified in conjunction with the INDEXED modifier, the line number of the input line can be used to control the destination of the line during output. For example:

INTEGER*2 LEN
DATA MOD1/2048/, MOD2/2/
1 CALL SCARDS(REG,LEN,MOD1,LNR,62) Read with PEEL
CALL SPRINT(REG,LEN,MOD2,LNA) Write INDEXED
GO TO 1
2 CALL EXIT

This program will read an input line, extract the line number, and write out the line with the extracted line number as the line number specification for an indexed write operation. The following sequence (assuming compilation of the above into -LOAD#)

$RUN -LOAD# SCARDS=*SOURCE* SPRINT=ABC
10AAA
12BBB

is equivalent to
$COPY *SOURCE*GETLINE* ABCAI
10 AAA
12 BBB

which is also equivalent to

$GET ABC
10 AAA
12 BBB

Listing the file ABC will produce

$LIST ABC
10  AAA
12  BBB

On output, if the PEEL (RETURNLINES) modifier is specified, the line number of the current output line is returned in the line number parameter of the subroutine call during the write operation. See the subroutine descriptions of SPRINT, SPUNCH, SERCCM, and WRITE. The line itself is written out and is unaffected by the presence or absence of this modifier. The modifier is used on output to aid the programmer in recording the line number of the current line written out.

**Bit 19 18 MACHCARCTRL, MCC**

Value: 4096 (dec) 00001000 (hex)
8192 00002000

Default: -MCC
Exceptions: None

The machine carriage control modifier pair is device-dependent. The MCC modifier is used for printing output (via printers or terminals) from programs producing output in which the first byte of each line is to be used as the command code in the Channel Command Word (CCW) used for output to a 1403 (or 1443) printer. If the MCC modifier is specified and the first byte of the output line is a valid 1403 CCW command code, the line is spaced accordingly and printing starts with the next byte as column 1. If the first byte is not a valid 1403 CCW command code, the entire line is printed using single spacing. Spacing operations performed by machine carriage control occur after the line is printed (as opposed to logical carriage control in which the spacing is performed before each line is printed). Most programs do not produce output using machine carriage control. The few programs that do (such as $ASMG and the TEXT360 programs) internally specify MCC for their output assuming that it is bound for a printer. Hence MCC need not be specified. If the user directs the output to a file, when the file is printed, MCC will have to be specified. For example:
$RUN *ASMG SCARDS=A SPRINT=B SPUNCH=C $COPY B TO *SINK*SMCC

The MCC modifier pair is ignored for files and all devices other than printers or terminals. For further information on machine carriage control, see the "Carriage Control" description in this volume.

Bit 17
16 TRIM

Value: 16364 (dec) 00004000 (hex)
        32768 00008000

Default: -TRIM
Exceptions: Line files, sequential files, and HPTS

The TRIM modifier pair is used to control the trimming of trailing blanks from input or output lines. If the TRIM modifier is specified, all trailing blanks except one are trimmed from the line. If -TRIM is specified, the line is not changed. A trimming operation does not physically delete the trailing blanks from the line, but only changes the line length count.

Bit 15
14 SPECIAL, SP

Value: 65536 (dec) 00010000 (hex)
        131072 00020000

Default: -SP
Exceptions: None

The SPECIAL modifier pair is reserved for device-dependent uses. Its meaning depends upon the particular device type with which it is used. The device support routines recognizing this modifier pair are:

1. The File routines (sequential files only)
2. The Data Concentrator routines
3. The Paper Tape routines
4. The Audio Response Unit routines

For information on the usage of this modifier pair in specifications involving the devices listed above, see the respective User's Guides in Volume 1 of the MTS Manual. The list of device support routines recognizing this modifier is volatile and subject to change without notice. Users who wish to keep their programs device-independent should not specify this modifier.
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

Bit 13
12
IC

Value: 262144 (dec) 00040000 (hex)
524288 00080000

Default: The setting of the IC global switch (usually ON)

Exceptions: None

The IC modifier pair controls implicit concatenation. If the IC modifier is specified, implicit concatenation will occur via the $CONTINUE WITH line. If -IC is specified, implicit concatenation will not occur. For example, SLIST PROGRAMS-IC will list the file PROGRAM and will print $CONTINUE WITH lines instead of interpreting them as implicit concatenation commands. The use of the IC modifier in I/O subroutine calls or as applied to FNames will override the setting of the implicit concatenation global switch (SET IC=ON or SET IC=OFF) for the I/O operations for which it is specified.

Bit 1
ERRRTN

Value: 1073741824 (dec) 40000000 (hex)

Default: -ERRRTN

Exceptions: None

If the ERRRTN modifier is specified (bit 1 in the modifier word is 1) when an I/O call is made, and if an I/O error occurs when no SETIOERR/SIOERR interception has been established, the error return code is passed back to the calling program instead of printing an error comment. This modifier may be used only with an I/O subroutine call. It may not be used as an attribute on an FName.

Bit 0
NOTIFY

Value: -2147483648 (dec) 80000000 (hex)

Default: -NOTIFY

Exceptions: None

If the NOTIFY modifier is specified (bit 0 in the modifier word is 1) when an I/O subroutine call is made, on return GRC is set to a value indicating what has happened:

0 = no unusual occurrence
1 = new FDUB opened and no I/O done
2 and above, reserved for future expansion

A new FDUB is opened if implicit concatenation occurred, if a change to the next member of an explicit concatenation is effected, or if a replacement FName is requested. This modifier may be used only with an I/O subroutine call. It may not be used as an attribute on a FName.

I/O Modifiers 373
CARRIAGE CONTROL

INTRODUCTION

The term carriage control refers to the user's ability to control the vertical spacing of his output. Carriage control is used mainly for output to a terminal or a printer. It may also be used to specify control operations for magnetic and paper tapes. See the appropriate Users' Guides in Volume I for details of usage. If the user has specified carriage control, the first character of every record (if output to a printer or a terminal) is interpreted as a carriage control character. For a description of the carriage control modifiers, see "I/O Modifiers" in this volume. The carriage control character determines the vertical positioning of the output page and is not part of the printed text. The control character is stripped from the output record and printing begins with the second character, rather than replacing the first character with a blank and starting the printing with the blank. If the first character is not one of the legal codes for the particular device being used, a default option of single space is assumed, and the first character is printed as part of the output text. The character codes are independent of the source language used by the programmer.

MTS supports two types of carriage control—logical and machine. Both are used in the manner described above, differing only in the legal carriage control characters and their effects. Logical carriage control is the more common, and, in general, the user need not be concerned with machine carriage control. MTS supports the machine type because several programs (notably the assembler and TEXT/360) produce it. In most cases in which carriage control is desired (such as output to printers and terminals), the default for logical carriage control is on. To select either machine carriage control or no carriage control, the appropriate modifier must be specified.

LOGICAL CARRIAGE CONTROL

The following table describes the logical carriage control characters and their effects.
### MTS 3: ROUTINE AND MACRO DESCRIPTIONS

**June 1970**

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect Before Printing</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Printer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBM Controller</td>
</tr>
<tr>
<td>blank</td>
<td>single space</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>double space</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>triple space</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>overprint previous line print without spacing first</td>
<td>ss&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>8</td>
<td>suppress carriage return after printing</td>
<td>undef</td>
</tr>
<tr>
<td>9</td>
<td>single space and suppress overflow&lt;sup&gt;3&lt;/sup&gt;</td>
<td>ss&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>skip to top of next page&lt;sup&gt;4&lt;/sup&gt;</td>
<td>skip 6&lt;sup&gt;5&lt;/sup&gt;</td>
</tr>
<tr>
<td>2</td>
<td>skip to next 1/2 page&lt;sup&gt;6&lt;/sup&gt;</td>
<td>skip 6</td>
</tr>
<tr>
<td>4</td>
<td>skip to next 1/4 page&lt;sup&gt;6&lt;/sup&gt;</td>
<td>skip 6</td>
</tr>
<tr>
<td>6</td>
<td>skip to next 1/6 page&lt;sup&gt;6&lt;/sup&gt;</td>
<td>skip 6</td>
</tr>
<tr>
<td>8</td>
<td>same as 6</td>
<td>skip 6</td>
</tr>
<tr>
<td>;</td>
<td>skip to top of next physical page (at perforation)</td>
<td>undef</td>
</tr>
<tr>
<td>&lt;</td>
<td>skip to bottom of physical page (at perforation)</td>
<td>undef</td>
</tr>
</tbody>
</table>

<sup>1</sup>ss = single space.
<sup>2</sup>undef = undefined, in which case spacing defaults to single space and the undefined character is printed as text.
<sup>3</sup>Normally, the printer automatically skips the first and last three lines of a page. A logical carriage control character of "3" suppresses this skip, causing these top and bottom margins to be ignored.
<sup>4</sup>"Top" is physically three lines down from the perforation because of the automatic margin mentioned above.
<sup>5</sup>Skip 6 lines.
<sup>6</sup>The logical page is divided into two halves, four quarters, and six sixths. A logical carriage control character of 6 will, for example, position the page at the next quarter block even if this may in fact be the top or the middle of a page.
READ

SUBROUTINE DESCRIPTION

Purpose: To read an input record from a specified logical I/O unit.

Location: Resident System

Alt. Entry: READ#

Calling Sequences:

Assembly: CALL READ, (reg, len, mod, lnum, unit)

FORTRAN: CALL READ (reg, len, mod, lnum, unit, &rc4,...)

PL/I: See the IFHEREAD subroutine description.

Parameters:

reg is the location of the core region to which data is to be transmitted.

len is the location of a halfword (INTEGER*2) integer giving the number of bytes read.

mod is the location of a fullword of modifier bits used to control the action of the subroutine. If mod is zero, no modifier bits are specified. See the "I/O Modifiers" description in this volume.

lnum is the location of a fullword integer giving the internal representation of the line number that is to be read or has been read by the subroutine. The internal form of the line number is the external form times 1000, e.g., the internal form of line 1 is 1000, and the internal form of line .001 is 1.

unit is the location of either a fullword integer giving the logical I/O unit number (0,...,19), a left-justified 8-character logical I/O unit name (e.g., SCARDS), or a fullword FDUB pointer (as returned by GETFD).

rc4 is the statement label to transfer to if the corresponding non-zero return code is encountered.

Return Codes:

0 Successful return.
4 End-of-file.
>4 See the "I/O Routines' Return Codes" description in this volume.
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

Page Revised March 1972

June 1970

Description: All five of the above parameters in the calling sequence are required. The subroutine reads a record from the I/O unit specified by unit into the region specified by reg and puts the length of the record (in bytes) into the location specified by len. If the MOD parameter (or the PDE name modifier) specifies the INDEXED bit, the LNUM parameter must specify the line number to be read. If the MOD parameter specifies the SEQUENTIAL bit, the subroutine will put the line number of the record read into the location specified by LNUM.

There are no default PDE names for READ.

There is a macro READ in the system macro library for generating the calling sequence to this subroutine. See the macro description for READ in this volume.

Examples: This example, given in assembly language and FORTRAN, calls READ specifying an input region of 20 fullwords. The logical I/O unit specified is 5 and there is no modifier specification made in the subroutine call.

Assembly:

CALL READ,(REG,LEN,MOD,LNUM,UNIT)
.
REG  DS  20F
LEN  DS  H
MOD  DC  F'0'
LNUM DS  F
UNIT DC  F'5'

or

READ 5,REG,LEN  Subr. call using macro.

FORTRAN:

INTEGER*2 LEN
.
.
CALL READ(REG,LEN,0,LNUM,5,630)
.
.
30
.

This example sets up a call to READ specifying that the input will be read from the file FYLE.

Assembly:

LA 1,'C'FYLE '
CALL GETFD
ST 0,UNIT
.
CALL READ,(REG,LEN,MOD,LNUM,UNIT)
.
REG DS  20F
LEN    DS    E
HCL    DC    F'I'
LNUM    DS    F
UNIT    DS    F

FORTRAN:  EXTERNAL GETFD
          INTEGER*4 ADRCP,UNIT
          CALL RCALL(GETFD,2,0,ADRCP('FYLE'),1,UNIT)
          :
          CALL READ(REG,LEN,0,LNUM,UNIT,330)
          :
          30    :

READ  195
SCARDS

SUBROUTINE DESCRIPTION

Purpose: To read an input record from the logical I/O unit SCARDS.

Location: Resident System

Alt. Entry: SCARDS#

Calling Sequences:

Assembly: CALL SCARDS,(reg,len,mod,linum)

FORTRAN: CALL SCARDS(reg,len,mod,linum,src4,...)

Parameters:

- **reg** is the location of the core region to which data is to be transmitted.
- **len** is the location of a halfword (INTEGER%2) integer giving the number of bytes to be transmitted.
- **mod** is the location of a fullword of modifier bits used to control the action of the subroutine. If **mod** is zero, no modifier bits are specified. See the "I/O Modifiers" description in this volume.
- **linum** is the location of a fullword integer giving the internal representation of the line number that is to be read or has been read by the subroutine. The internal form of the line number is the external form times 1000, e.g., the internal form of line 1 is 1000, and the internal form of line .001 is 1.
- **src4*** is the statement label to transfer to if the corresponding non-zero return code is encountered.

Return Codes:

- 0 Successful return.
- 4 End-of-file.
- >4 See the "I/O Routines' Return Codes" description in this volume.

Description: All four of the above parameters in the calling sequence are required. The subroutine reads a record into the region specified by **reg** and puts the length of record (in bytes) into the location specified by **len**. If the **mod** parameter or the FDNAME modifier specifies the INDEXED bit, the **linum** parameter must specify the line number to be read. If the **mod** parameter specifies the SEQUENTIAL bit, the subroutine will put the line number of the record read into the location specified by **linum**.
The default Fdname for SCARDS is *SOURCE*.

There is a macro SCARDS in the system macro library for generating the calling sequence to this subroutine. See the macro description for SCARDS in this volume.

Examples:

This example, given in assembly language and FORTRAN, calls SCARDS specifying an input region of 20 fullwords. There is no modifier specification made on the subroutine call.

Assembly:  

```
CALL SCARDS,(REG,LEN,MOD,LNUM)
```

```
REG DS 20F  
LEN DS H    
MOD DC '0'  
LNUM DS F 
```

or

```
SCARDS REG,LEN  Subr. call using macro.  
```

FORTRAN:

```
INTEGER*2 LEN
```

```
CALL SCARDS(REG,LEN,0,LNUM,830)
```

```
30  
```
**SDUMP**

**SUBROUTINE DESCRIPTION**

**Purpose:**
To produce a dump of any or all of the following:

1. general registers
2. floating point registers
3. a specified region of core storage

**Location:**
Resident System

**Calling Sequences:**

Assembly: CALL SDUMP, (switch, outsub, wkarea, first, last)

**Parameters:**

- **switch** is the location of a fullword containing switches that govern the content and format of the dump produced. The switches are assigned as follows:
  
  bit 31: on if hexadecimal conversion of the core region is desired.
  bit 30: on if mnemonic conversion of the core region is desired.
  bit 29: on if EBCDIC conversion of the core region is desired.
  bit 28: on if double spacing is desired; off if single spacing is desired.
  bit 27: on if long output records (130 characters) are to be formed; off if short output records (70 characters) are to be formed.
  bit 26: on if general registers are to be dumped.
  bit 25: on if floating point registers are to be dumped.
  bit 24: on if a core region is to be dumped.
  bit 23: on if no column headers are to be produced.

- **outsub** is the location of a subroutine (e.g., SPRINT) that causes the printing, punching, etc. of the output line images formed by SDUMP.

- **wkarea** is the location of a doubleword aligned area of 400 bytes that may be used by SDUMP as a work area.

- **first** is the location of the first byte of a core
region to be dumped. There are no boundary requirements for this address.

last is the location of the last byte of a core region to be dumped. There are no boundary requirements for this address; however, an address in last which is less than the address in first will cause an error return.

Note: The default case for switch (all switches off) produces a dump as though bits 24, 25, 26, and 31 were on. Furthermore, if bit 30 (memonics) is on, bit 31 (hexadecimal) is implied. Note that bits 24, 25, and 26 specify what is to be dumped, bits 27 and 28 specify the page format, and bits 29, 30, and 31 specify the interpretation(s) to be placed on the region of core specified. Bits 29 through 31 have significance only if bit 24 is on.

Return Codes:

0 Successful return.
4 Illegal parameters.

Description: Output Formats

Registers:

General and floating point registers, if requested, are always given in labeled hexadecimal format. The length of the output record is governed by the setting of bit 27 of the switch.

Core Storage:

Although any combination of switches is acceptable, the appearance of the dumps output for a region of core storage is determined as follows:

1. If, and only if, the mnemonic switch is on, the unit of core storage presented in each print item is a halfword aligned halfword.

2. If, and only if, the mnemonic switch is off and the hexadecimal switch is on (through intent or default), the unit of core storage presented in each print item is a fullword aligned fullword.

3. If, and only if, the mnemonic and hexadecimal switches are off but the EBCDIC switch is on, the unit of core storage presented in each print item is a doubleword aligned doubleword.
In all cases, the output includes

(1) the entire core storage unit (halfword, fullword, or doubleword) in which the first specified core location (parameter first) is found,

(2) the entire core storage unit in which the last location (parameter last) is found, and

(3) all intervening storage.

Thus, the first and last printed items of a core storage dump may include up to a maximum of seven core bytes more than actually requested in the parameter list.

If mnemonics are requested and SDUMP discovers a byte that cannot be interpreted as an operation code, then instead of a legal mnemonic, the characters "XXXX" appear directly below the hexadecimal presentation of the halfword in core that should have contained an operation code. When this occurs, the mnemonic scanner jumps ahead as though the illegal operation code specified an RR type instruction (two bytes) and tries to interpret the byte at the new location as an operation code, etc. Any mnemonic print line that contains the "XXXX" for at least one of its entries is also marked with a single "X" directly below the line address that prefixes the hexadecimal presentation of that same region of core. (The mnemonic conversion routine includes the Universal Instruction Set and those instructions exclusively used by the Model 67.) To facilitate the location of particular items in the output, line addresses always have a zero in the least significant hexadecimal position. Column headers are provided which give the value of the least significant hexadecimal digit of the address of the first byte in each print item.

A line of dots is printed to indicate that a region of core storage contains identical items. The core storage unit used for comparisons is halfword, fullword, or doubleword depending upon the type(s) of conversion specified. In all cases, the core storage unit corresponding to the last item printed before the line of dots and the core storage unit for the first item after the line and all intervening core storage units have identical contents. The last line is always printed (even if all of its entries exactly match the previously printed line).
Example: Assembly:

```
EXTRN SPRINT
CALL SDUMP,(SW,PRINT,WK,FIRST,FIRST+3)

WK    DS    50D
SW    DC    '0'
FIRST DC    X'F1F2F3F4'
```

The above example will cause SDUMP to print the hex string 'F1F2F3F4'.

MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

Page revised July 1971

June 1970
SPRINT
SUBROUTINE DESCRIPTION

Purpose: To write an output record on the logical I/O unit SPRINT.

Location: Resident System

Alt. Entry: SPRINT#

Calling Sequences:

Assembly: CALL SPRINT,(reg,len,mod,lnum)

FORTRAN: CALL SPRINT(reg,len,mod,lnum,&rc4,...)

Parameters:

reg is the location of the core region from which data is to be transmitted.

len is the location of a halfword (INTEGER*2) integer giving the number of bytes to be transmitted.

mod is the location of a fullword of modifier bits used to control the action of the subroutine. If mod is zero, no modifier bits are specified. See the "I/O Modifiers" description in this volume.

lnum (optional) is the location of a fullword integer giving the internal representation of the line number that is to be written or has been written by the subroutine. The internal form of the line number is the external form times 1000, e.g., the internal form of line 1 is 1000, and the internal form of line .001 is 1.

rc4,... is the statement label to transfer to if the corresponding non-zero return code is encountered.

Return Codes:

0 Successful return.
4 Output device is full.
>4 See the "I/O Routines' Return Codes" description in this volume.

Description: The subroutine writes a record of length len (in bytes) from the region specified by reg on the logical I/O unit SPRINT. The parameter lnum is needed only if the mod parameter specifies either INDEXED or PEEL (RETURNLINE#). If INDEXED is specified, the line number to be written is specified in lnum. If PEEL is specified, the line number of the record written is returned in lnum.
The default Fdname for SPRINT is *SINK*.

There is a macro SPRINT in the system macro library for generating the calling sequence to this subroutine. See the macro description for SPRINT in this volume.

Examples:

This example given in assembly language and FORTRAN calls SPRINT specifying an output region of 80 bytes. No modifier specification is made in the subroutine call.

**Assembly:**
```
CALL SPRINT,(REG,LEN,MOD)
```

```
REG  DS  20F
MOD  DC  F'0'
LEN  DC  H'80'
```

or
```
SPRINT REG,LEN  Subr. call using macro.
```

**FORTRAN:**
```
DATA LEN*2/80/
CALL SPRINT(REG,LEN,0)
```
SDDMP

SUBROUTINE DESCRIPTION

Purpose: To dump a region of the user's virtual memory in the MTS
standard format. For dumping registers, dumping with ene-
mcnics, and other options, see the SDUMP subroutine descrip-
tion in this volume.

Location: Resident System

Calling Sequences:

Assembly: CALL SDDMP,(switch,outsub,wkarea,first,last)

Parameters:

switch is the location of a fullword of information.
The first halfword of switch is taken as the
storage index number that will be printed out in
the heading line. The remainder of switch is
taken as a group of switches as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>(Integer value = 2048) NOLIB</td>
</tr>
<tr>
<td></td>
<td>If set, the call will be ignored if</td>
</tr>
<tr>
<td></td>
<td>LLOADINFO declares that the region of</td>
</tr>
<tr>
<td></td>
<td>storage is part of a library</td>
</tr>
<tr>
<td></td>
<td>subroutine.</td>
</tr>
<tr>
<td>28</td>
<td>(Integer value = 8) DOUBLE SPACE</td>
</tr>
<tr>
<td></td>
<td>If this bit is set, the lines of the</td>
</tr>
<tr>
<td></td>
<td>dump will be double spaced. Otherwise the</td>
</tr>
<tr>
<td></td>
<td>normal single spacing will occur.</td>
</tr>
</tbody>
</table>

outsub is the location of a subroutine that will be
called by SDDMP to "print" a line. This
subroutine is assumed to have the same calling
sequence as the SPRINT subroutine.

wkarea is the location of a 100-word (fullword aligned)
region which SDDMP will use as a work area.

first is the location of the first byte of a core
region to be dumped. There are no boundary
requirements for this address.

last is the location of the last byte of a core
region to be dumped. There are no boundary
requirements for this address; however, an
address in last which is less than the address
in first will cause an error return.
Return Codes:

0 Successful return.
4 Illegal parameters.

Description: This subroutine uses the same calling sequence as the subroutine SDUMP, but only looks at the bits and parameters as specified above in the calling sequence.

For each call, this subroutine "prints" (calls the output subroutine specified in outsub) the following:

1. Blank line.
2. Heading giving information about the region of storage. The subroutine LOADINFO is called to obtain the information.
3. Blank line.
4. Dump of the region, with 20 (hex) bytes printed per line. To the left of the hexadecimal dump is the actual hex location and the relative (to the first byte of the region) hex location of the first byte of the line; to the right of the dump is the same information printed as characters. Non-printing characters (bit combinations that do not match the standard 60 character set of printing graphics) are replaced by periods, and an asterisk (*) is placed at each end of the character string to delimit it. The lines "printed" are 133 characters long.

Example:

<table>
<thead>
<tr>
<th>Assembly:</th>
<th>EXITW SPRINT CALL STDDMP,(SW,SPRINT,WK,FIRST,FIRST+3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WK DS 50D</td>
</tr>
<tr>
<td></td>
<td>SW DC P'0'</td>
</tr>
<tr>
<td></td>
<td>FIRST DC X'F1F2F3F4'</td>
</tr>
</tbody>
</table>

The above example will cause STDDMP to print the hex string 'F1F2F3F4'.
WRITE

SUBROUTINE DESCRIPTION

Purpose: To write an output record on a specified logical I/O unit.
Location: Resident System
Alt. Entry: WRITE#
Calling Sequences:

Assembly: CALL WRITE,(reg,len,mod,lnum,unit )
FORTRAN: CALL WRITE(reg,len,mod,lnum,unit,src,...)
PL/I: See the INHERITE subroutine description.

Parameters:

reg is the location of the core region from which data is to be transmitted.
len is the location of a halfword (INTEGER*2) integer giving the number of bytes to be transmitted.
mod is the location of a fullword of modifier bits used to control the action of the subroutine. If $\text{mod}$ is zero, no modifier bits are specified. See the "I/O Modifiers" description in this volume.
lnum is the location of a fullword integer giving the internal representation of the line number that is to be written or has been written by the subroutine. The internal form of the line number is the external form times 1000, e.g., the internal form of line 1 is 1000, and the internal form of line .001 is 1.
unit is the location of either a fullword integer giving the logical I/O unit number (0,...,19), a left-justified 6-character logical I/O unit name (e.g., SPRINT), or a fullword FDB5B pointer (as returned by GGETFD).
src,... is the statement label to transfer to if the corresponding non-zero return code is encountered.

Return Codes:

0 Successful return.
4 Output device is full.
>4 See the "I/O Routines' Return Codes" description in this volume.
The subroutine writes a record on the logical I/O unit specified by `unit` of length `len` (in bytes) from the region specified by `reg`. The parameter `lnum` is used only if the `mod` parameter specifies either `INDEXED` or `PEEL` (RETURN line). If `INDEXED` is specified, the line number to be written is specified in `lnum`. If `PEEL` is specified, the line number of the record written is returned in `lnum`.

There are no default FDnames for WRITE.

There is a macro WRITE in the system macro library for generating the calling sequence to this subroutine. See the macro description for WRITE in this volume.

This example given in assembly language and FORTRAN calls WRITE specifying an output region of 60 bytes. The logical I/O unit specified is 6 and no modifier specification is made in the subroutine call.

**Assembly:**

```assembly
CALL WRITE,(REG,LEN,MOD,NUM,UNIT)

REG DS 20F
MOD DC F'0'
NUM DS F
LEN DC H'80'
UNIT DC F'6'
```

or

```
WRITE 6,REG,LEN Subr. call using macro.
```

**FORTRAN:**

```fortran
DATA LEN*2/80/

CALL WRITE(REG,LEN,0,NUM,6)
```

This example given in assembly language sets up a call to WRITE specifying that the output will be written into the file FYLE.

**Assembly:**

```assembly
LA 1,'FYLE '
CALL GETFD
ST 0,UNIT

CALL WRITE,(REG,LEN,MOD,NUM,UNIT)

REG DS 20F
LEN DS H
MOD DC F'0'
```
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

Page Revised March 1972

LNUM  DS  F
UNIT   DS  F

FORTRAN:

EXTERNAL GETFD
INTEGER*4 ADROP,UNIT
CALL RCALL(GETFD,2,0,ADROP('FILE '),1,UNIT)

CALL WRITE(REG,LEN,0,LNUM,UNIT,630)

30
The Computing Center maintains a number of macro libraries in public files. In addition, the user can construct and use his own macro libraries.

Any macro library to be used when assembling a program must be explicitly mentioned when running the assembler. Up to five macro libraries may be used for one assembly. A macro library is specified by attaching it to one of logical units 2, 3, 4, 5, or 0 when running the assembler. For example,

\texttt{S\textsc{run} \*ASMG SCARDS=SOURCEPGM SPUNCH=OBJ 0=\*SYSMAC}

will use \*SYSMAC as a macro library. The macro libraries are searched for a definition in the order of

1. Macro library attached to logical unit 2
2. Macro library attached to logical unit 3
3. Macro library attached to logical unit 4
4. Macro library attached to logical unit 5
5. Macro library attached to logical unit 0

so that

\texttt{S\textsc{run} \*ASMG SCARDS=IN SPUNCH=OBJ 2=MYMACLIBR 0=\*SYSMAC}

will cause a macro to be expanded from MYMACLIBR if it is there, or, otherwise, from \*SYSMAC. Note that any macro definitions supplied with the assembler input will take precedence over definitions in a macro library.

The following public files contain macro libraries:

\*SYSMAC

\*SYSMAC is the system macro library. These macros are described below.

\*OSMAC

\*OSMAC contains the macro library from IBM's Operating System. It is designed to enable the assembling of OS programs under MTS. The programs so assembled must not be run under MTS. Descriptions of these macros are found in "IBM System/360 Operating System Supervisor and Data Management Macro-Instructions", form C28-6647.
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

*1

*1 contains a set of macros to implement the *1 list processing language. *1 is a low-level list language similar to L6. For a complete description, see the "*1 User's Guide" in Volume 7 of the MTS Manual.
CONSTRUCTING A MACRO LIBRARY

As described below, a macro library has a rather simple structure. Small macro libraries can be easily constructed by hand. For constructing larger macro libraries, the program *MACGEN is available.

Structure of a Macro Library

A macro library is a line-file containing both a directory of the macros and the macro definitions themselves.

A. The directory:

1. Each entry of the directory contains the name of a macro in columns 1-8 and the line-number of the macro definition header of the corresponding macro in columns 10-16. Both the name and the line-number must be left justified with trailing blanks.

2. The line-number of the first entry in the directory must be 1.

3. The terminating entry in the directory is a string of eight zeros in columns 1-8.

B. The macros:

1. The line-number of the macro-definition header of each macro must be a positive integral number.

2. The first macro follows the last entry in the directory.

Example:

```
$NUMBER  1,1
BASR    10
BAS     20
00000000
$NUMBER  10, 1
MACRO
&LABEL  BASR  &REG1,&REG2
&LABEL  BALR  &REG1,&REG2
MEND
$NUMBER  20, 1
MACRO
&LABEL  BAS  &REG1,&LOC
&LABEL  BAL  &REG1,&LOC
MEND
```
MACGEN

The public file *MACGEN contains a program to construct a macro library. Before the program is run, the macro definitions should be put in the line file starting at some relatively high positive line number. The MACRO line of each definition must occur on an integral line number. Then *MACGEN is run to construct the directory, which must start at line 1 of the file. *MACGEN reads the definition via SCARDS and writes the directory via SPUNCH. An example might be

$RUN *MACGEN SCARDS=FILE(1000) SPUNCH=FILE(1)

if the definitions begin in line 1000 of FILE.

The following would produce a macro library from the same macros as the previous section:

$CREATE MAC
$NUMBER 1000
MACRO
&LABEL BASR &REG1,&REG2
&LABEL BALR &REG2,&REG2
MEND
MACRO
&LABEL BAS &REG1,&LOC
&LABEL BAL &REG1,&LOC
MEND
$RUNNUMER
$RUN *MACGEN SCARDS=MAC(1000) SPUNCH=MAC
#MACGEN

Contents: The object module of the macro library generator program.

Purpose: To generate a directory for a set of macro definitions.

Usage: The program is invoked by an appropriate $RUN command.

Logical I/O Units Referenced:
- SCARDS - the file containing a set of macro definitions.
- SPUNCH - the file which will contain the directory.

Example:

$RUN *MACGEN SCARDS=WATMAC(1000) SPUNCH=WATMAC

Description: An MTS-formatted line directory is produced on SPUNCH for the set of macro definitions read through SCARDS. Entries in the line directory occupy integral line numbers beginning with line number one (1). The directory terminator is inserted after the last macro definition has been read.
ENTER

MACRO DESCRIPTION

Purpose: To generate prolog code for the entrance to a subroutine.

Location: *SYMSAC

Prototype: [label] ENTER reg[,SA=savarea][,LENGTH=len][,TREG=tempreg]

Parameters:

reg is the register to be established as a base register. It should not be 15.

savarea (optional) is a keyword parameter specifying the location of a save area to use. If savarea is omitted, a call to the GETSPACE subroutine is made to get a save area of length specified by len.

len (optional) is a keyword parameter specifying the length of the save area to be obtained if savarea is omitted. If len is omitted, 72 is used.

tempreg (optional) is a keyword parameter specifying the temporary register to be used in the prolog code. If omitted, GPR5 is used. tempreg should not be the same as reg.

Description: ENTER causes code for the following to be produced:

1. Generates USING *,reg
2. Establishes reg as the base register.
3. If savarea is omitted, a call to GETSPACE is made to get a save area of length len.
4. Establishes forward and backward links between save areas.

Examples: SUBR ENTER 12
           F ENTER 9,SA=SAVAREA
           G ENTER 11,TREG=12
EXIT

MACRO DESCRIPTION

Purpose: To reestablish the calling program's save area and to return with a return code in GR15 and an optional returned value in GR0.

Location: *SYSMAC

Prototype: [label] EXIT [rc][,rval][,MF=fs]

Parameters:

rc (optional) is a self defining term or the location of a fullword return value to be loaded into GR15. If rc is omitted, the return code is zero. rc may be expressed as a register number in parentheses.

rval (optional) is a self defining term or the location of a fullword return value to be loaded into GR0. It may be expressed as a register number in parentheses.

fs (optional) specifies that the save area pointed to by GR13 is to be released by calling FREEPAC.

Description: This macro requires that the save area be properly linked or entry to the subroutine as is done by the ENTR macro. If the ENTR macro is used and it obtains space via the G2ISPACE subroutine, this space can be released by specifying MF=fs.

Examples:

EXIT 4
OUT EXIT 0,(0) Return value is in GR0.
EXIT 0,4 Return value is 4.
LABEL EXIT 0,RVAL Return value is in RVAL.
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

IOH/360 MACROS

MACRO DESCRIPTION

Purpose: To generate calls to IOH/360 to perform formatted input and output.

Location: *LIBRARY

Prototype:

(label) RDFMT fmt[,(par,...)]
(label) PRFMT fmt[,(par,...)]
(label) RCFMT fmt[,(par,...)]
(label) WRFMT fmt[,(par,...)]
(label) SSRFMT fmt[,(par,...)]
(label) GUSFMT fmt[,(par,...)]

Parameters:

fmt specifies the location of the IOH/360 format. This
must be given as a symbolic expression. See the
"Introduction to IOH/360" in this volume for a
description of the format language.

par specifies one simple or block parameter giving the
location to be read or written. If a simple
parameter is desired, this must be specified as a
symbolic expression. If a block parameter is
desired, this must be specified as two symbolic
expressions separated by ",", for example,

A,...,A+20

Description: The above macros are used to call IOH/360
from assembly language programs. This description covers
only the most elementary usage omitting many additional parameters
which may be specified, and several other related macros. For a
complete description of IOH/360, see the "IOH/360 User's

When one of these macros is executed, IOH/360 will be called
to perform input or output according to the format given by
fmt into or from the locations specified by par. Any number
of simple or block parameters may be specified, and input or
output will continue until a parameter specified as "0" is
encountered. For this reason, the last par should be given
as "0" to terminate input or output.

By using some of the more advanced features of these macros,
it is possible to compute dynamically the parameters to be
used, specify parameters relative to base registers, etc.
Those users who need the advanced features should see the
Examples:

```
RDLBL RDFMT INFMT,(CNT,A,...,A+10*4,0)
  .
  INFMT DC C'I,11WF*'
CNT   DS F
A     DS 11E

This example will read one fullword integer and 11 fullword floating-point numbers in free format.

PRLBL PRFMT OFMT,(NUM,RESULT,C,...,C+5*4,0)
  .
  NUM   DS F
RESULT DS E
C     DS 6E
OFMT DC C""-CASE ",I5,"RESULTS ",7WF6.2*'

This example will print one fullword integer and 7 fullword floating-point numbers plus the two comments in the format specification.
```
RETURN

MACRO DESCRIPTION

Purpose: To return control to the calling program and to signal normal termination of the returning program.

Location: *SYSMAC

Prototype: [label] RETURN [(E1 [, E2]) ][, T][, RC=code]

Parameters:

E1, E2 (optional) is the range of registers to be restored from the save area to which the address in GR13 points. The registers should be specified to cause the loading of registers 14, 15, 0 through 12 when used in a LM instruction. If E2 is not specified, only the register specified by the E1 operand is loaded. If the operand is omitted, the contents of the registers are not altered.

T (optional) causes the control program to flag the save area used by the returning program. A byte containing all 1's is placed in the high-order byte of word 4 of the save area after the registers have been loaded.

code (optional) is the return code to be passed to the calling program. The return code should have a maximum value of 4095; it will be placed right-adjusted in GR15 before the return is made. If RC=(15) is coded, it indicates that the return code has been previously loaded into GR15; in this case the contents of GR15 are not altered or restored from the save area. (If this operand is omitted, the contents of GR15 are determined by the E1, E2 operands.)

Description: The return of control by the RETURN macro instruction is always made by executing a branch instruction using the address in GR14. This macro can be written to restore a specified range of registers, provide the proper return code in GR15, and flag the save area by the returning program. See the "Calling Conventions" description in this volume for a further explanation of save areas and their formats.

---

1"IEM System/360 Operating System Supervisor Data Management and Macro Instructions", Form C28-6647.
MTS 3: SUBROUTINE AND MACRO DESCRIPTIONS

June 1970

Examples:

LAB1 RETURN (14, 12), RC=4
LAB2 RETURN (5, 10), T
LAB3 RETURN (5, 10), T, RC=(15)