```
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************

********************************************************************************
********************************************************************************
********************************************************************************
```

```
*************************************************************************************
** VM/370 VERSION 06 LEVEL es PLC Pack **************** CLASS A *** DEV 00E **********


           USERID    ORIGIN      MAINT       MAINT                                 VV
                                                                                   VV
           DISTRIBUTION CODE     MAINT                                      3333333333
                                                                           33333333333
           SPOOL FILE NAME TYPE  SOSREF      MEMO                          33      VV3
                                                                                    V3
           CREATION DATE         04/06/19 14:46:47                                   3
                                                                                  3333
           SPOOL FILE ID         0859                                             3333
                                                                                     3
           RECORD COUNT          5070                                                3
                                                                           33        3
                                                                           33333333333
                                                                            3333333333

                                                                              VM/370 R



                                    MM        MM   AAAAAAAAAA    IIIIIIIIII   NN        NN
                                    MMM      MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN       NN
                                    MMMM    MMMM  AA        AA       II       NNNN      NN
                                    MM MM  MM MM  AA        AA       II       NN NN     NN
                                    MM  MMMM  MM  AA        AA       II       NN  NN    NN
                                    MM   MM   MM  AAAAAAAAAAAA       II       NN   NN   NN
                                    MM        MM  AAAAAAAAAAAA       II       NN    NN  NN
                                    MM        MM  AA        AA       II       NN     NN NN
                                    MM        MM  AA        AA       II       NN      NNNN
                                    MM        MM  AA        AA       II       NN       NNN
                                    MM        MM  AA        AA   IIIIIIIIII   NN        NN
                                    MM        MM  AA        AA   IIIIIIIIII   NN         N



                                    MM        MM   AAAAAAAAAA    IIIIIIIIII   NN        NN
                                    MMM      MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN       NN
                                    MMMM    MMMM  AA        AA       II       NNNN      NN
                                    MM MM  MM MM  AA        AA       II       NN NN     NN
                                    MM  MMMM  MM  AA        AA       II       NN  NN    NN
                                    MM   MM   MM  AAAAAAAAAAAA       II       NN   NN   NN
                                    MM        MM  AAAAAAAAAAAA       II       NN    NN  NN
                                    MM        MM  AA        AA       II       NN     NN NN
                                    MM        MM  AA        AA       II       NN      NNNN
                                    MM        MM  AA        AA       II       NN       NNN
                                    MM        MM  AA        AA   IIIIIIIIII   NN        NN
                                    MM        MM  AA        AA   IIIIIIIIII   NN         N



*************************************************************************************
*************************************************************************************
*************************************************************************************
*************************************************************************************
*************************************************************************************

*************************************************************************************
*************************************************************************************
*************************************************************************************
```

```
*************************************************************************************
** VM/370 VERSION 06 LEVEL es PLC Pack **************** CLASS A *** DEV 00E **********


           USERID    ORIGIN      MAINT       MAINT                                  VV
                                                                                    VV
           DISTRIBUTION CODE     MAINT                                       3333333333
                                                                            33333333333
           SPOOL FILE NAME TYPE  SOSREF      MEMO                           33      VV3
                                                                                     V3
           CREATION DATE         04/06/19 14:46:47                                    3
                                                                                   3333
           SPOOL FILE ID         0859                                              3333
                                                                                      3
           RECORD COUNT          5070                                                 3
                                                                            33        3
                                                                            33333333333
                                                                             3333333333

                                                                               VM/370 R



                                    MM         MM   AAAAAAAAAA    IIIIIIIIII   NN         NN
                                    MMM       MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN        NN
                                    MMMM     MMMM  AA        AA       II       NNNN       NN
                                    MM MM   MM MM  AA        AA       II       NN NN      NN
                                    MM  MMMM  MM   AA        AA       II       NN  NN     NN
                                    MM   MM   MM   AAAAAAAAAAAA       II       NN   NN    NN
                                    MM         MM  AAAAAAAAAAAA       II       NN    NN   NN
                                    MM         MM  AA        AA       II       NN     NN NN
                                    MM         MM  AA        AA       II       NN      NNNN
                                    MM         MM  AA        AA       II       NN       NNN
                                    MM         MM  AA        AA   IIIIIIIIII   NN        NN
                                    MM         MM  AA        AA   IIIIIIIIII   NN         N



                                    MM         MM   AAAAAAAAAA    IIIIIIIIII   NN         NN
                                    MMM       MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN        NN
                                    MMMM     MMMM  AA        AA       II       NNNN       NN
                                    MM MM   MM MM  AA        AA       II       NN NN      NN
                                    MM  MMMM  MM   AA        AA       II       NN  NN     NN
                                    MM   MM   MM   AAAAAAAAAAAA       II       NN   NN    NN
                                    MM         MM  AAAAAAAAAAAA       II       NN    NN   NN
                                    MM         MM  AA        AA       II       NN     NN NN
                                    MM         MM  AA        AA       II       NN      NNNN
                                    MM         MM  AA        AA       II       NN       NNN
                                    MM         MM  AA        AA   IIIIIIIIII   NN        NN
                                    MM         MM  AA        AA   IIIIIIIIII   NN         N
```

Brown University
Student Operating System
Reference Manual

by

Sidney H. Gudes   Robert F. Gurwitz

Peter J. Relson

Program in Computer Science

Brown University

Providence, Rhode Island 02912

January 30, 1977

Revised: May 11, 1978

Printed: May 20, 1978

This is the first revision of the Brown University Student Operating System Reference Manual. It obsoletes the previous edition.

The revisions include typographical corrections and minor technical changes necessitated by changes in the SOS assembler and supervisor. These changes include /SOS card parameters, C' ' character strings, literal pool allocation, macro generator enhancements, and the addition of Appendix F.

1 PREFACE

This is a reference guide to the Brown University Student Operating System (SOS). SOS provides facilities for machine language or assembly language programming, in a batch processing environment.

This manual does not teach programming since it presupposes knowledge of a high level programming language, but it does attempt to provide all material needed for the reader inexperienced with assembly language. As a reference manual, it is intended to be used in conjunction with class discussions, handouts, and demos. A glossary is included so that quick references to unfamiliar terms can be made.

Section two describes the structure and instruction set of the (simulated) SOS machine. Following this are discussions of the various sections of SOS, each controlled by the operating system (called the Supervisor): the Input/Output facilities, the machine code processor, the assembler, and the job control processor. The macro generator is then described.

The original SOS specifications[1] have been modified by J. Dill, D. Dixon, C. Gallagher, S. Gudes, R. Gurwitz, D. Notkin, P. Relson, and P. Wisoff to include more powerful assembler features and to improve some of the Input/Output capabilities. SOS was rewritten to incorporate these features by C. Gallagher, S. Gudes, R. Gurwitz, P. Relson, and P. Wisoff.

The authors would like to thank the following people for the time that they spent criticizing and proofreading this manual: D. Dixon, R. Fleming, D. Notkin, R. Sedgewick and A. van Dam.

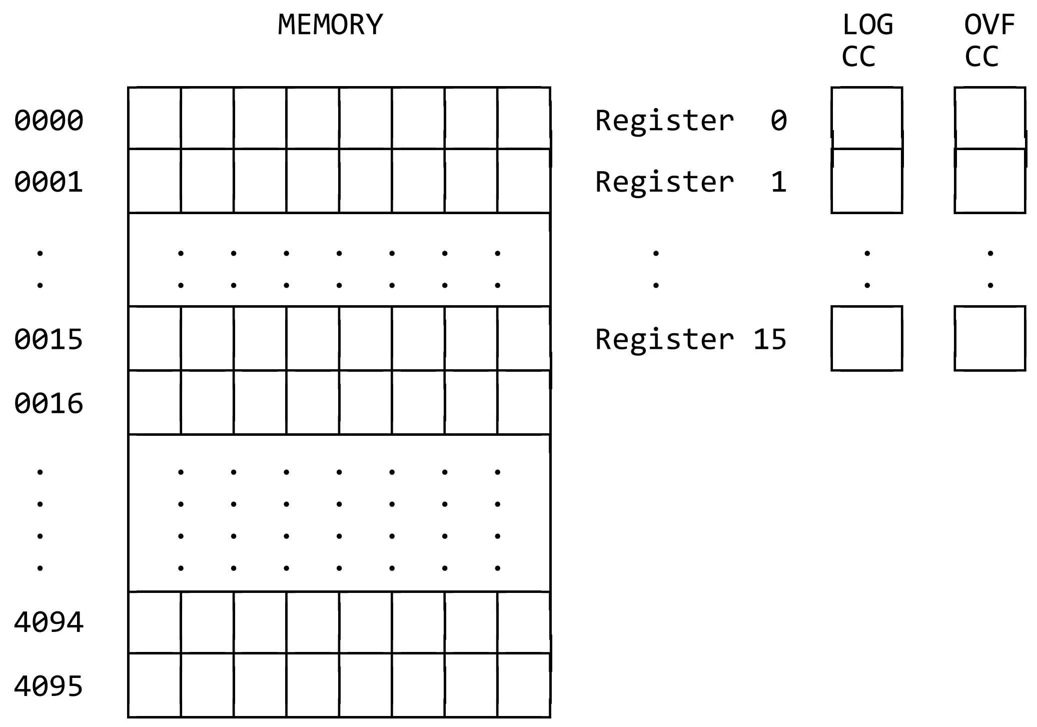
Thanks are also given to D. Meyer for making possible the diagrams contained herein.

In addition, we would like to thank the authors of the previous (1968) SOS manual: R. Batts, L. Kaufman, P. Knueven, R. Kogut, D. Krecker, M. Michel, J. Michener, R. Miller, and K. Prager.

[1]Wile, Munck, and van Dam, "Brown University Student Operating System," Proceedings of the ACM National Meeting, 1967.

2 SOS MACHINE

2.1 Structure of the SOS Machine



Structure of SOS Memory

2.1.1 Memory

The basic unit of information stored and processed by the SOS machine is a bit, one binary digit. Eight bits taken as a group are a "byte," and a word (also known as a fullword) consists of four bytes. Each word is generally written as eight hexadecimal (base sixteen) digits, a convenient shorthand for the 32 binary digits. Memory consists of 4096 words -- addresses 0000 through 4095 in decimal, 000 through FFF in hexadecimal.

2.1.2 Registers

The first sixteen locations of memory participate in arithmetic and logical operations and are called registers.

A logical condition code and an overflow condition code are associated with each register. They may be thought of as "hardware" extensions to the registers and may be set, reset and tested by individual instructions.

Certain arithmetic instructions, and no others, affect the overflow condition code; certain logical instructions, and no others, affect the logical condition code.

If overflow occurs during the execution of an arithmetic instruction, then the overflow condition code for the register involved is set, otherwise it is reset. Since instructions for which overflow cannot occur do not affect the overflow condition code, it remains set until it is reset.

The logical condition code is set by one of the logical instructions. This condition code can have one of three possible states -- all zeroes, all ones, or mixed. The logical condition code retains its status until it is changed by another logical instruction.

2.1.3 Data Representation

Three basic data types are handled by the SOS machine: arithmetic (positive and negative integers), logical (patterns of bits), and character (codes representing printable characters).

A positive integer is stored in a full word with leading zeroes, e.g., decimal 106 would be stored as the bit pattern

```
0000 0000 0000 0000 0000 0000 0110 1010
```

and is written as 0000006A in hexadecimal. The leftmost digit (bit, in binary) is referred to as digit (bit) zero. Thus, the binary representation has bits zero through 31; the hexadecimal representation has digits zero through seven.

Negative integers are stored in 32-bit two's complement notation (see Appendix A); for example, minus five is stored as

```
1111 1111 1111 1111 1111 1111 1111 1011
```

and is written as FFFFFFFB in hexadecimal.

Hexadecimal numbers will be indicated by "X' '" notation; e.g., X'10' is the hexadecimal number 10 (decimal sixteen).[1]

Logical data are also stored as fullwords, but are treated as strings of 32 independent bits, each bit being considered either "on" or "off" (one or zero), or alternatively "true" or "false".

Characters are stored in a coded notation known as EBCDIC, one character per byte. Characters are packed four bytes to the fullword, so if the number of characters to be stored is not an exact multiple of four, the assembler will pad the string on the

[1]Unless otherwise stated or indicated, all numbers in the text will be in base ten.

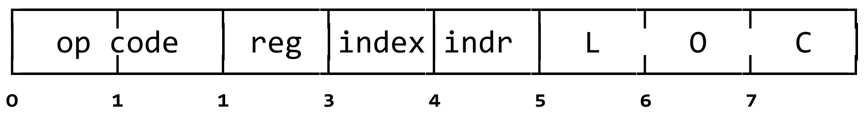
right with blanks to fill up the space in the last fullword of the string. Thus, "BIT" is stored as X'C2C9E340'.[3] See Appendix D for a table of characters and their EBCDIC equivalents.

2.1.4 Instruction Formats

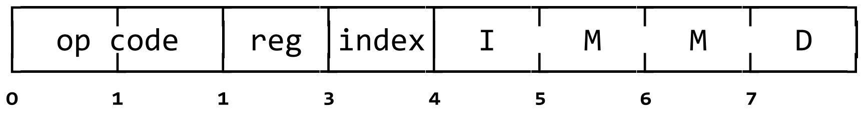
Most SOS instructions perform binary operations on the contents of two memory locations. One of the operands must be a register, and its contents are replaced by the result of the operation.

In the descriptions below, references to the SOS general purpose registers will be made using symbolic (mnemonic) equivalents. Thus, registers zero through fifteen are referred to as R0 through R15. It is suggested that all programming done in SOS assembler use these mnemonic names for the general purpose registers. A symbolic name surrounded by parentheses will be used to denote the contents of the specified memory location; e.g., (R1) refers to the contents of register 1; (VARX) to the contents of location VARX.

The basic SOS instruction consists of one fullword: a two (hexadecimal) digit operation code, a one digit register specification, a one digit index register specification, a one digit indirect addressing field, and a three digit location field. The last five digits are collectively called the address field.



The two "immediate" instructions (AXAI and SXAI) use a slightly different format, in which hex digits four to seven (sixteen bits) contain "immediate" data to be used in executing the instruction.



2.2 Run-Time Effective Address Calculation

Most of the time, when an instruction is executed, the SOS machine must compute the "effective address" of the operand. This calculation makes use of the address field of the instruction.

[3]Note that the character "B" is stored internally as the bit string 1100 0010 (X'C2'). Thus the distinction between the characters "A" through "F" and the hexadecimal digits A through F should always be remembered.

*	*	*	index	indr	L	O	C
0	1	1	3	4	5	6	7

Address calculation proceeds as follows:

(1) Start with the LOC field.
(2) If the indexing field is <u>non-zero</u>, add the contents of the specified register to the LOC value.
(3) Check the rightmost bit of the indirecting field. If it is off (zero), the effective address calculation is complete. If it is on (one), however, use the address that has been computed so far as the address of a word in SOS memory. Repeat steps (1), (2), and (3) with the address field of this word until the indirecting field is off or the limit of five levels of indirecting has been exceeded. If the latter occurs, or at any time the calculated address does not lie within the range zero to 4095 (X'FFF'), the program will abend. Situations requiring more than one level of indirecting are exceedingly rare.

Examples:

X'10300123'	The effective address is X'123'. Since no indexing or indirecting is specified, only the LOC field is used.
X'20470123'	Index register 7 is specified, so the effective address is X'123'+(R7). If R7 contains five, the effective address will be X'128'.
X'50301100'	Suppose that location X'100' contains the word X'0000034B'. An address of X'100' would be computed initially (no indexing). Since the indirecting flag is on, the contents at that address are used. Thus the effective address is computed from X'0000034B', with a result of X'34B' (no indexing or indirecting).

As a rather complicated example, suppose (R1) = 1, (R2) = 3, (R3) = 7, location X'100' contains X'00021101' and location X'104' contains X'50930300'. The effective address calculation for the instruction X'607110FF' proceeds as follows.

(1) The initial address found is LOC + index register = X'0FF' + (R1) = X'100'.
(2) Indirecting is on, so examine the contents of X'100'.
(3) The contents are X'00021101', so the initial address is X'101' + (R2) = X'104'.
(4) Indirecting is still on, so examine the contents of X'104', X'50930300'.
(5) The initial address is X'300' + (R3) = X'307'. Note that the first three digits of the word are totally ignored during effective address calculation.

(6) Indirecting is off, so the address calculated is the effective address, and the instruction means Store the contents of R7 at location X'307'. Of course, if the values in registers 1, 2, or 3 had been different, a different effective address would most likely have been found.

Remember that if the effective address is in the range zero to fifteen, it references one of the sixteen SOS general purpose registers.

Effective address calculation is done for all instructions except H, BR, AXAI, and SXAI.

2.3 SOS Instruction Set

The following sections describe each instruction, presenting its machine code format and assembly language mnemonic and format, as well as a description of its operation and an example of its use. For some instructions, programming hints (possible uses) are also given.

There are seven types of instructions: arithmetic, for doing fixed point arithmetic; logical, for performing logical operations and tests; data transfer, for moving information between registers and memory; shift, for manipulating register contents; branch, for changing the flow of control; halt, for terminating program execution; and Input/Output initiation.

The machine language format is given in the form introduced in Instruction Formats (section 2.1.4). The abbreviations used are:

R -- Register specification
X -- Index register specification
I -- Indirecting flag
LOC -- SOS memory location
* -- unused (ignored) field

The assembly language format (summarized in Appendix C) is given using assembler mnemonics and the following notation:

```
OP R,>LOC(X)
   or
OP R,IMMD(X)
```

Where:

OP = Opcode mnemonic
R = Register specification
> = Indirecting flag (optional)
X = Index register specification (optional)
LOC = SOS memory location
IMMD = Immediate data

2.3.1 Arithmetic Instructions

These instructions perform fixed point arithmetic as well as some other data manipulations. Many of these instructions set the overflow condition code.

Add
```
A     R,>LOC(X)
```

10	R	X	I	LOC
0	1	3	4	5

Add the contents of the location specified by the effective address to the contents of the specified register and place the sum in the specified register. The contents of the location at the effective address remain unchanged. If arithmetic overflow is detected in the sum, the overflow condition code is set; otherwise it is cleared.

Example:

Suppose that register 5 contains X'00000019' (25 decimal) and location X'200' contains X'FFFFFFFE' (-2 decimal). After executing

```
      A  R5,X'200'                (X'10500200')
```

register 5 contains X'00000017' (23 decimal) and the overflow condition code for R5 is cleared.

Subtract
```
S     R,>LOC(X)
```

20	R	X	I	LOC
0	1	3	4	5

Subtract the contents of the location specified by the effective address from the contents of the specified register. The difference replaces the contents of the specified register, and the contents of the location at the effective address remain unchanged. If arithmetic overflow is detected in the result, the overflow condition code is set; otherwise it is cleared.

This instruction can be used to zero a register, by subtracting the register from itself.

Example:

Suppose that register 2 contains X'0000000B' (11 decimal) and location register 4 contains X'0000000D' (13 decimal). After executing

```
        S   R2,R4              (X'20200004')
```

register 2 contains X'FFFFFFFE' (-2 decimal) and the overflow condition code for R2 is reset (the overflow condition code for R4 is unaffected).

Multiply
M   R,>LOC(X)

30	R	X	I	LOC
0	1	3	4	5

Multiply the contents of the specified register plus one (R+1) by the contents of the location specified by the effective address. The result is a 64-bit two's complement product placed in the register pair R, R+1. The contents of the location specified by the effective address remain unchanged. Register 15 may not be used as the specified register. The overflow condition code is not affected.

Example:

Suppose that register 7 contains X'FFFFFFFF' (-1 decimal) and location X'2AC' contains X'00000003' (3 decimal). After executing

```
        M   R6,X'2AC'           (X'306002AC')
```

registers 6 and 7 contain X'FFFFFFFFFFFFFFFD' (-3 decimal).

Divide
D   R,>LOC(X)

40	R	X	I	LOC
0	1	3	4	5

Divide the contents of the register pair specified by R, R+1 by the contents of the location specified by the effective address. The contents of the register pair are taken as a 64-bit two's complement dividend. After execution, the resulting remainder and quotient occupy registers R and R+1 (respectively) as two 32-bit two's complement numbers. The remainder has the same sign as the dividend, and the sign of the quotient is determined by the rules of algebra.

If arithmetic overflow occurs (the quotient is too large to fit in one register), a divide exception results, the overflow condition code is set for R; R and R+1 remain unchanged. Otherwise the overflow condition code is cleared. If division by zero is attempted, a zero-divide exception results and the program abends. Register 15 may not be used as the specified register.

Example:

Suppose that register 3 contains X'00000000', register 4 contains X'0000000A' (10 decimal), and location X'05A' contains X'00000003'. After executing

```
D   R3,X'05A'           (X'4030005A')
```

register 3 contains X'00000001' (the remainder), register 4 contains X'00000003' (the quotient), and the overflow condition code for R3 is cleared.

Note that if a 32-bit (or smaller) dividend is desired, the preceding register must contain zero if the dividend is positive, or minus one if negative. The SRDA instruction can be used to prepare for such a division.

Add Index Add Immediate
AXAI   R,IMMD(X)

A2	R	X	I M M D
0	1	3	4 7

Subtract Index Add Immediate
SXAI   R,IMMD(X)

A1	R	X	I M M D
0	1	3	4 7

The contents of the index register are added to or subtracted from the specified register, the immediate data is added to the result, and this sum replaces the specified register. The condition codes remain unchanged. These instructions can be summarized as

```
AXAI:     (R) + (X) + IMMD -> R
SXAI:     (R) - (X) + IMMD -> R
```

The immediate field is treated as a sixteen-bit two's complement number, which is expanded to a 32-bit number by extending the sign bit (bit sixteen of the instruction) on the left. Thus, an immediate field of X'FFFF' would be expanded to X'FFFFFFFF' (decimal minus one), and an immediate field of X'7FFF' would be expanded to X'00007FFF' (decimal 32767) before being added to the specified register.

These instructions are very useful for incrementing or decrementing registers by sixteen-bit two's complement constants, or loading registers with such constants. The advantage of using these instructions lies in the fact that no extra storage locations are used for data.

To add a constant to a register:

    Suppose that register 5 contains X'00001000'.
    After executing

```
        AXAI   R5,-1              (X'A250FFFF')
```

    register 5 contains X'00000FFF'.

To load a constant into a register:

    Suppose that register 3 contains X'0000000C'.
    After executing

```
        SXAI   R3,4(R3)            (X'A1330004')
```

    register 3 contains X'00000004'.

Note that SXAI R0,2(R0) will just add two to the contents of R0, rather than set the contents of R0 to two, since an index field of zero indicates that no indexing is to be done.

2.3.2 Logical Instructions

These instructions perform logical operations and tests. Each instruction affects the logical condition code associated with the specified register.

AND
N   R,>LOC(X)

B0	R	X	I	LOC
0	1	3	4	5

The contents of the specified register are logically ANDed with the contents of the location specified by the effective address. The operation is done separately for each of the 32 bits. The result replaces the contents of the specified register and the contents of the location specified by the effective address remain unchanged. The logical condition code is set to reflect the 32 bits of the specified register after execution -- all ones, all zeroes, or mixed.

This instruction can be used to clear selected bits (set them to zero) in a register.

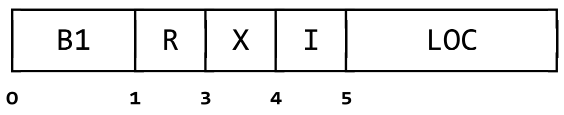
Example:

    Suppose that register 4 contains X'ABCDEF12'
    and location X'037' contains X'0000000F'.
    After executing

```
         N   R4,X'037'           (X'B0400037')
```

```
       register 4 contains X'00000002'.
```

OR
O   R,>LOC(X)



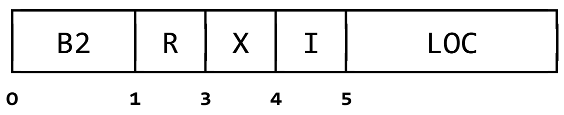
All specifications are identical to the AND instruction, except that the operation ORs the specified bits.

This instruction can be used to set selected bits in a register to one.

Example:

```
       Suppose that register 4 contains X'03AC2019'
       and location X'02B' contains X'0000000F'.
       After executing

               O   R4,X'02B'           (X'B140002B')

       register 4 contains X'03AC201F'.
```

Exclusive OR (XOR)
X   R,>LOC(X)



All specifications are identical to the AND instruction, except that the operation logically XORs the bits.

This instruction can be used to take the negative of a number, by XORing the number with minus one and adding one to the result.

Example:

```
       Suppose that register 0 contains X'FFFFFFFE'
       and location X'DEF' contains X'FFFFFFFF'.
       After executing

               X   R0,X'DEF'           (X'B2000DEF')

       register 0 contains X'00000001'.
```

A bit XORed with one will have its value changed regardless of whether it was zero or one. The instruction can also be used to exchange the contents of two registers without using an additional storage area:

```
       X   R1,R2
       X   R2,R1
       X   R1,R2
```

Test Under Mask
```
TM     R,>LOC(X)
```

B3	R	X	I	LOC
0	1	3	4	5

The contents of the specified register are tested according to the 32-bit mask contained in the location specified by the effective address. If a bit in the mask is zero, the corresponding bit in the register is masked (ignored); otherwise, it is tested to determine whether it is one or zero. The logical condition code is set to reflect whether the selected bits are all zeroes, all ones, or mixed.

Example:

```
       Suppose that register 5 contains X'FF00FF00'
       (1111 1111 0000 0000 1111 1111 0000 0000 in binary)

       and location X'0A0' contains X'00003000'
       (0000 0000 0000 0000 0011 0000 0000 0000 in binary).
```

Thus, bits 18 and 19 of the register are to be tested. The corresponding bits in register 5 are on, so after executing

```
              TM   R5,X'0A0'           (X'B35000A0')
```

the logical condition code is set to "all ones."

This instruction can be used to test flags (each word can contain up to 32 individual flags), or status bits of CCWs set by I/O operations, as described in section 3.3.

2.3.3 Data Transfer Instructions

Load
```
L     R,>LOC(X)
```

50	R	X	I	LOC
0	1	3	4	5

The contents of the location specified by the effective address are placed in the specified register. The contents of the location at the effective address remain unchanged.

Example:

```
       Suppose that register 6 contains X'00000104'.
```

After executing

```
        L   R5,R6               (X'50500006')
```

register 5 contains X'00000104'.

Store
ST   R,>LOC(X)

60	R	X	I	LOC
0	1	3	4	5

The contents of the specified register are placed into the location specified by the effective address. The contents of the specified register remain unchanged.

Example:

Suppose that register 8 contains X'00123456'.
After executing

```
       ST   R8,X'13A'            (X'6080013A')
```

location X'13A' contains X'00123456'.

2.3.4 Shift Instructions

These instructions move the contents of the specified register, bit by bit, to the left or right. The effective address of the shift instruction is calculated in the usual manner, but the result is used to indicate the number of bit positions to be shifted rather than as a reference to a storage location. Only the low order six bits of the specified effective address are used. Thus, shifts of up to 63 bits may be specified.

Shift Right Logical
SRL   R,>LOC(X)

70	R	X	I	LOC
0	1	3	4	5

Shift Right Double Logical
SRDL   R,>LOC(X)

72	R	X	I	LOC
0	1	3	4	5

Shift Left Logical
SLL   R,>LOC(X)

80	R	X	I	LOC
0	1	3	4	5

Shift Left Double Logical
SLDL   R,>LOC(X)

82	R	X	I	LOC
0	1	3	4	5

The contents of the register(s) are shifted to the left or right the number of bit positions specified by the effective address. Zeroes are moved in and bits shifted out are lost. Double shifts treat the register pair R, R+1 as 64-bit data, so register 15 may not be the specified register. The logical condition code of the specified register is set to reflect the resulting contents of that register.

Example:

```
       Suppose that registers 14 and 15 contain X'4040404040404040'
       and register 2 contains X'00000004'.
       After executing

               SRDL   R14,0(R2)       (X'72E20000')

       registers 14 and 15 contain X'0404040404040404'.
```

Shift Right Algebraic
SRA   R,>LOC(X)

71	R	X	I	LOC
0	1	3	4	5

Shift Right Double Algebraic
SRDA   R,>LOC(X)

73	R	X	I	LOC
0	1	3	4	5

Shift Left Algebraic
SLA   R,>LOC(X)

81	R	X	I	LOC
0	1	3	4	5

Shift Left Double Algebraic
SLDA   R,>LOC(X)

83	R	X	I	LOC
0	1	3	4	5

The contents of the specified register(s) are shifted to the left or right the number of bit positions specified by the effective address. For left shifts, zeroes are moved in on the right and bit zero (the sign bit) remains unchanged. The overflow condition code is set if a bit different from the sign bit is shifted past (out of) bit one; otherwise it is reset. For right shifts, the sign bit is propagated from the left, bits shifted out are lost, and the overflow condition code remains unchanged. Double shifts treat the contents of the register pair R, R+1 as 64-bit data, so register 15 may not be the specified register.

Shifting a register left algebraically by n bits is equivalent to multiplying the register by 2**n (e.g., shifting a register left by four is equivalent to multiplying it by sixteen). Conversely, shifting a register right algebraically n bits is equivalent to dividing by 2**n. Note that if a negative odd number is divided in this manner, the result will be rounded away from zero rather than toward zero, as expected. The advantages to multiplying and dividing by shifting (if it can be done) are that it is much faster than arithmetic multiplication or division and that it uses only one register.

Example:

Suppose that register 5 contains X'80000000'.
After executing

```
         SRA   R5,24           (X'71500018')
```

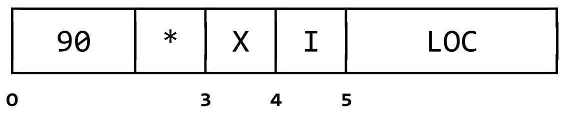
register 5 contains X'FFFFFF80'.

One way to set up for a division is to load the 32 bit dividend into register Rx and do a SRDA Rx,32.

2.3.5 Branch Instructions

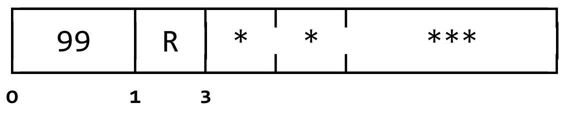
The branch instructions cause execution to continue at a location other than the next sequential one, provided that the condition tested by the branch (if any) is true. Some branches are used to test the logical and overflow condition codes, although none affect the condition codes.

Branch
B   >LOC(X)



Continue execution (unconditionally) at the location specified by the effective address.

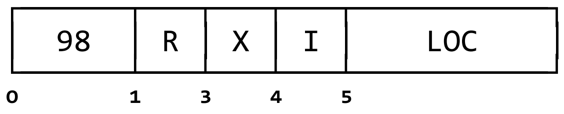
Branch Register
BR   R



Continue execution at the location specified by the contents of the specified register. The contents of this register are treated as an SOS memory address, and thus must lie in the range zero to X'FFF'.

Branch Register can be used to return from a subroutine to the calling program (see below).

Branch and Link
BAL   R,>LOC(X)



The current contents of the Program Counter are loaded into the specified register and execution continues at the location specified by the effective address. After execution of this instruction, the specified register contains the address of the location just after the BAL instruction, since the Program Counter is incremented prior to instruction execution (see Appendix B).

This instruction is used for subroutine calls, since the return address for the subroutine (the address of the next location) is inserted in a register and can be accessed via a Branch Register instruction upon completion of the subroutine. For example,

```
ROUTINE A:                        ROUTINE B:

    •                             STARTB   •
    •                                      •
    BAL  R14,STARTB    Call B              •
    •                             ENDB    BR   R14  Return to caller
    •
    •
```

Execution of the BAL causes execution to continue at STARTB and places the return address into register 14. When the BR is executed at ENDB, control returns to the calling routine at the instruction after the BAL.

Branch On Count
BCT   R,>LOC(X)

A0	R	X	I	LOC
0	1	3	4	5

The contents of the specified register are decremented by one and returned to the register. If the result is zero, the next sequential instruction is executed; otherwise, execution continues at the location specified by the effective address.

Example:

Suppose that register 5 contains X'00000003'. Then

```
LOOP  •
      •
      •
      •
      BCT  R5,LOOP
```

causes the code contained in the body of the loop to be executed three times, implementing the loop construct "DO R5 = 3 TO 1 BY -1". However, if register 5 contained zero initially, the loop would be executed 2**32 times.

Branch on Register High
BH   R,>LOC(X)

91	R	X	I	LOC
0	1	3	4	5

Branch on Register Equal
BE   R,>LOC(X)

92	R	X	I	LOC
0	1	3	4	5

Branch on Register Low
BL   R,>LOC(X)

93	R	X	I	LOC
0	1	3	4	5

Execution continues at the location specified by the effective address if the contents of the specified register are postive, zero, or negative, respectively. Otherwise, execution continues at the next sequential location.

Branch Ones
BO   R,>LOC(X)

95	R	X	I	LOC
0	1	3	4	5

Branch Mixed
BM   R,>LOC(X)

96	R	X	I	LOC
0	1	3	4	5

Branch Zeroes
BZ   R,>LOC(X)

97	R	X	I	LOC
0	1	3	4	5

Execution continues at the location specified by the effective address if the logical condition code for the specified register indicates all ones, mixed, or all zeroes, respectively. Otherwise, the next sequential instruction is executed.

Branch on Overflow
BOF   R,>LOC(X)

94	R	X	I	LOC
0	1	3	4	5

Execution continues at the location specified by the effective address if the overflow condition code for the specified register is set. Otherwise, the next sequential instruction is executed.

2.3.6 Halt Instruction

Halt
H   >LOC(X)

00	*	X	I	LOC
0	1	3	4	5

Execution of the halt instruction terminates program execution normally. The specified address (with normal indexing and indirecting) is assembled for use as an address constant (see section 5.1.3), but effective address calculation is not performed during execution of the instruction.

Example:

The instruction

```
        H    X'2E'                  (X'0000002E')
```

causes program execution to be terminated.

2.3.7 Input/Output Initiation

SOS does I/O through a separate simulated Data Channel (see section 3 for a full description of SOS I/O). The Channel executes a channel program consisting of CCW's (channel command words) written by the user or by the system I/O macros. The Supervisor is requested to activate the channel so that it may perform the indicated I/O operations. The SVC instruction has been provided for this purpose.

Supervisor Call
SVC   0,>LOC(X)

C0	0	X	I	LOC
0	1	3	4	5

The Supervisor is called to activate the Data Channel to perform the I/O operation specified by the channel program starting at the location specified by the effective address. Execution resumes at the next sequential instruction when the I/O operation has been completed.

Note that a zero in the R field of the instruction indicates a call for I/O initiation; any other value in the R field is invalid.

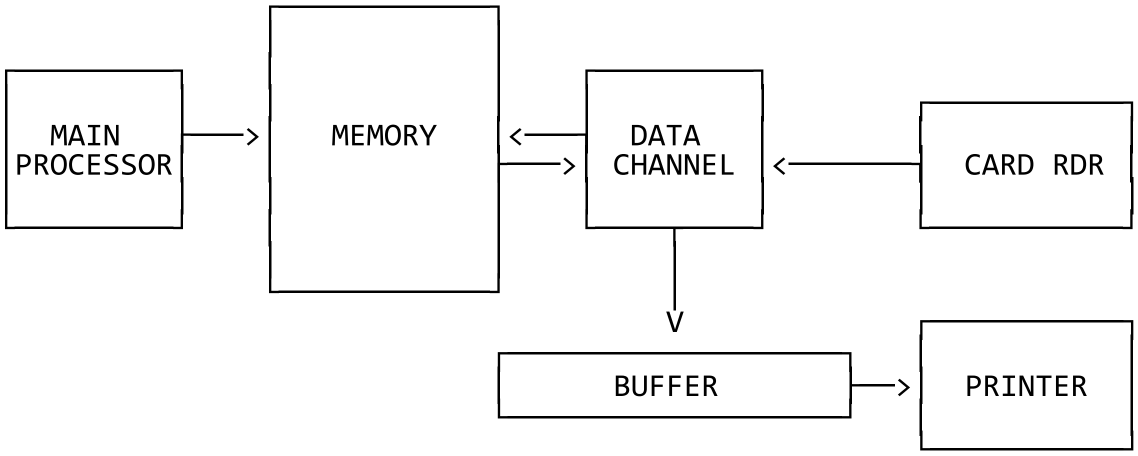
3 INPUT/OUTPUT

3.1 Input/Output Principles

3.1.1 Introduction and Channel Structure

SOS Input/Output operations allow the system to communicate with the user via a card reader and a printer. Data to be processed by the program are read by the card reader and results are output to the printer.

The structure of the SOS machine relevant to I/O operations is



The key concept is the use of a Data Channel to perform the I/O operations. The Data Channel (Channel for short) can be thought of as a special purpose computer. Like the Main Processor, it can access any word in memory.

The Channel executes numerically coded I/O instructions called channel command words (CCW'S), in a manner analogous to the way in which the Main Processor executes machine instructions; i.e., it has an Instruction Register and a Program Counter and operates in a fetch-execute cycle (see Appendix B). The Channel is connected to the printer via an output buffer.

In order to perform an I/O operation, the Channel must be activated by the Supervisor. This process is initiated by a particular instruction known as the Supervisor Call (SVC). When the processor encounters an SVC, it stops executing user program instructions and notifies the Supervisor. The Supervisor then determines the type of SVC from the register field of the instruction. A zero (the only legal value) in this field indicates a request for an I/O operation. The effective address of the SVC points to the beginning of a channel program consisting of one or more channel command words. The Supervisor activates the Channel and passes it the address of the channel program.

The Channel executes the channel program. On completion, it notifies the Supervisor, which signals the Main Processor to resume execution of the machine program, starting with the instruction immediately following the SVC. Note that the channel program is not in the line of execution of the program, for the Main Processor does not execute CCW's. In a sense, an I/O SVC is a call to a subroutine to be executed by the Channel rather than by the Main Processor.

3.1.2 Why Channels Are Used

Most real computers use Data Channels which, once activated, can operate simultaneously with the Main Processor. Since I/O operations are usually extremely slow compared to instruction execution speeds, this overlapping of I/O with processing avoids a great deal of waiting and allows much more efficient programming. However, such systems also raise problems. For example, the main program may try to use data before the Channel has finished reading it in. The SOS machine does not overlap its I/O and processing, but the concept of the Data Channel with its separate I/O CCW's has been retained for pedagogical purposes.

3.1.3 General Facilities

I/O operations are performed with three data formats: two's complement hexadecimal numbers (X format), signed decimal numbers (D format), and character strings (C format). The Channel handles conversion of data from internal machine representation to external form and vice-versa. In addition, the Channel outputs the trace and partial dumps during execution.

SOS provides two facilities for I/O programming: CCW chains (user-created channel programs), and a set of system macros that generate a predefined set of channel commands to perform simple, commonly used operations. While the macros suffice for elementary operations, more extensive I/O operations should be done with user-created channel programs. For each type of CCW there is a system macro (section 3.3.1) that generates it along with an inline SVC to activate the Channel. Hence, the macro names will be used in the following discussion to refer to both the CCW and the system macro that generates it. For descriptions of the individual CCW formats and operations, and the corresponding system macros, see section 3.3.

3.1.3.1 Input

Each data type has a corresponding GET command which causes information to be read from cards, converted to internal representation, if necessary, and stored in memory. Each GET command causes one card to be read. GETX and GETD allow a variable number of hexadecimal or decimal numbers, respectively, to be read from a single card and placed in sequential memory locations. GETC reads a variable number of characters from a card and stores them contiguously.

3.1.3.2 Output

SOS output operations are modelled after typewriter functions. Thus, there are commands to write information in each of the three formats (PUTX, PUTD, PUTC); to space and backspace the "carriage" (SKIP, BKSP); to set, clear, and use tab stops (STAB, CTAB, TAB); and to set up a user heading buffer, which consists of two output lines (each of 120 characters), for page headings (HEAD).

Output does not go directly to the printer. Rather, each line is constructed in an output buffer area (which is separate from the heading buffer) one output line in length. The special "carriage return" command (RET) prints the contents of the buffer, clears it (fills it with blanks), and repositions the "carriage" to column one in preparation for the next line of output.

3.1.3.3 Diagnostic Aids

The commands TON and TOFF turn the trace on and off. They can be set to change the trace status after a specified number of executions of the given instructions. PDUMP causes the contents of memory between specified core locations to be dumped in hexadecimal and character formats.

3.2 Channel Programs and Chaining

A channel program consists of a series of CCW's which are executed by the Channel. While the Main Processor automatically fetches and executes instructions until stopped by a Halt instruction or an error, the Channel must be specifically instructed to fetch the next CCW on completion of the previous one. This is done by setting a "chain bit" (see below). As long as the chain bits in the CCW's comprising a channel program are set, CCW's are fetched and executed sequentially. When a chain bit of zero is encountered, control is returned to the Main Processor after execution of that CCW.

Compared to issuing a separate SVC for each CCW, CCW chaining is advantageous in that less memory is used, a lower instruction count results since a chain of CCW's counts as only one instruction, and better efficiency is achieved for complex I/O operations.

3.3 CCW's and System I/O Macros

This section describes the format of the channel command word and the assembler CCW instruction. The second part of the section describes the system I/O macros, along with descriptions of the individual CCW's.

3.3.1 CCW Formats

A CCW occupies two sequential machine words in core. Some commands do not utilize the second word, although CCWs are always assembled and fetched as two words and unused fields are ignored by the Channel.

Channel Command Word

```
CCW     CC,SB,CT,>LOC(X)
```

CC	SB	RC	CT
0	1	4	6

***	X	I	LOC
0	3	4	5

The specified fields of the CCW are assembled into two full words for execution by the Channel. Interpretation of the various CCW's by the Channel is described below.

The fields have the following meanings:

CC -- Command code (word 1, bits 0-7). This is analogous to the opcode in a machine instruction. It identifies the operation to be performed by the Channel. Any expression between zero and 255 is valid in the assembler specification.

SB -- Status Byte (word 1, bits 8-15). The eight bits of this byte are used to indicate various exceptional conditions encountered during execution of the command. An exception to this is the first bit, which is the chain bit, set by the programmer. All other bits are to be examined by the programmer after execution. Any expression between zero and 255 is valid in the assembler specification. The bit meanings when on are:

0 (high order bit) -- Chain Bit: Execute this CCW and chain it to the next one. This is the only bit that should be set by the programmer.

1 -- Backspace Exception: An attempt was made to backspace past the left margin.

2 -- Command Code Exception: Invalid command code, the command is ignored.

3 -- Get Hexadecimal or Decimal Exception: Fewer numbers were found on the current card than specified in the count field.

4 -- Data Exception: Invalid characters were found in a decimal or hexadecimal number, or the number was too long to fit in one full word.

5 -- Overflow Exception: An attempt was made to move the carriage beyond the right margin in a PUT, SKIP, or TAB command.

6 -- Overprint Exception: New characters were moved into the buffer over non-blank ones, which are lost.

7 (low order bit) -- Card Reader End of File.

RC -- Return Count (word 1, bits 16-23). This count is set by the Channel when an exception occurs during execution of a GET command. It indicates data missing or not read. See the individual command descriptions for the exact meaning.

CT -- Count (word 1, bits 24-31), specified in the individual command descriptions.

* -- Unused (ignored) field.

The X, I, and LOC fields (word 2, bits 12-31) have the same meaning as an address in a regular machine instruction (index register specification, indirecting flag, and location field). They are specified the same way to the assembler. This address points to data used in some operations, and is unused in others. See the individual command descriptions.

For example,

```
       CCW   9,0,1,>X'107'(R3)
```

assembles into the consecutive words X'09000001' and X'00031107'.

Test Under Mask (TM) can be used to test the status byte following execution of a channel program. A mask of X'007E0000' applied to the first word of a CCW could be used with a Branch Mixed (BM) test to transfer control to an error routine if any status bit other than the end of file or chain bit is set.

3.3.2 CCW Operations and System I/O Macro Formats

This section describes the individual CCWs and the system I/O macros which can also be used to generate them. A CCW with a command code not mentioned below will be ignored. For each command, the name and format of the corresponding system macro is given, along with the format of the actual CCW and an operational description. For the macro operands, the following notation is used:

&CT appearing alone means that the user is to specify an expression whose value is to appear in the count field of the CCW.

&CT=n means that if the user leaves this parameter out, the value n will be assumed in the count field. Otherwise, if CT=expression is specified, the value of the expression is to appear in the count field.

&ADD means that the user is to specify an expression, whose value is to appear in the address field of the CCW. This expression may be in >LOC(X) form.

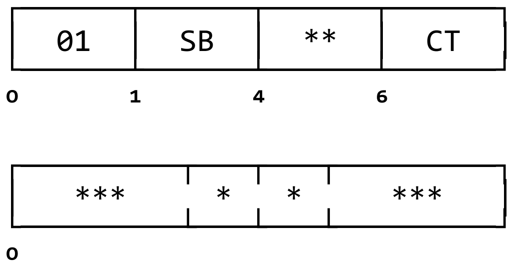
&EOF means that the user is to specify an expression whose value is an address to be branched to in the event of end of file condition. The expression may be in >LOC(X) form.

&ADDFR, &ADDTO mean that the user is to specify an expression as for &ADD.

The following notation is used in the CCW format descriptions:

SB -- Status byte
RC -- Return count
CT -- Count
X -- Index register specification
I -- Indirecting flag
LOC -- Specified location
* -- Unused (ignored) field

Backspace
BKSP &CT



The carriage is backspaced the number of spaces specified by the count field. The contents of the buffer remain unchanged. Attempting to backspace over the left margin results in setting of status bit 1 and positioning the carriage at the left margin. No indication of the number of excess backspaces is returned.

Tab
TAB

02	SB	**	**
0	1	4	6

***	*	*	***
0			

The carriage is moved to the next tab that has been set by the programmer (see Set Tab, below). The contents of the buffer remain unchanged. A tab over the right margin results in setting status bit 5, and performing a "RET 0".

Set Tab
STAB &CT

03	SB	**	CT
0	1	4	6

***	*	*	***
0			

A tab is set at the column number which appears in the count field. The buffer contents and carriage position remain unchanged. If the count field is greater than 120, the command is ignored. All tabs are cleared at the beginning of each job.

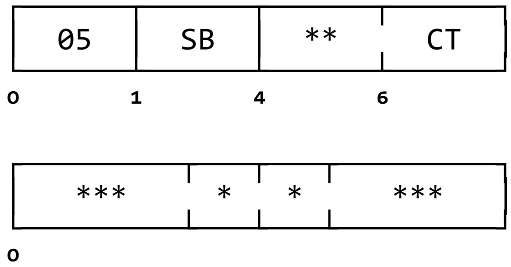
Clear Tab
CTAB &CT

04	SB	**	CT
0	1	4	6

***	*	*	***
0			

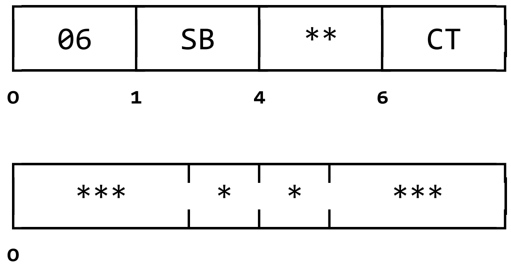
The tab stop in the column specified by the count field is cleared. A count field of zero clears all tabs. The buffer contents and carriage postion remain unchanged. If the count field is greater than 120, the command is ignored.

Print, Return Carriage, Clear Buffer, and Space
RET   &CT



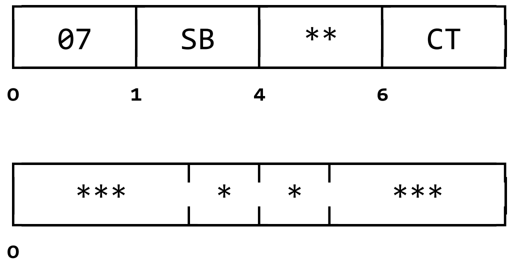
The contents of the buffer are printed. The carriage is positioned at the left margin of the next line on the page and the buffer is cleared (filled with blanks). The number of lines specified in the count field are then skipped on the page. If &CT is zero, it may be omitted from the macro invocation.

Space Carriage
SKIP   &CT



The carriage is moved to the right by the number of columns specified in the count field. The contents of the buffer remain unchanged. Spacing over the right margin causes a "RET 0" to be done. Status bit 5 is set in this case.

Heading Control
HEAD   &CT



The count field is tested. If it is zero, a new page will be started when the next RET is done -- a new page is not started immediately upon execution of this CCW; the contents of the buffer remain unchanged. If the count is one, the first line of the heading buffer is set up with the contents of the output buffer. If the count is two, the second line of the heading buffer is set up with the contents of the output buffer. In the latter two cases, the output buffer is cleared and the carriage is positioned at the left margin. Any other value in the count field causes a value of zero to be assumed. If &CT is zero, it may be omitted from the macro invocation.

Put Hexadecimal
PUTX &ADD,&CT=8

08	SB	**	CT
0	1	4	6

***	X	I	LOC
0	3	4	5

The contents of the location specified by the effective address are written into the buffer in hexadecimal format starting at the current carriage position. Only the number of digits specified by the count field (with a maximum of eight) are written; high order digits are ignored if fewer than eight are specified. Negative numbers are generated in hex complement form. Overflow off the right end of the carriage causes a "RET 0" to be done (printing the contents of the buffer at the time of the overflow), the digits which did not fit in the buffer are lost and status bit 5 is set. Overprinting, i.e., writing over non-blank characters in the buffer, causes status bit 6 to be set, and the old characters are lost. Note that a count of less than eight can be used to suppress leading zeroes.

Get Hexadecimal
GETX &ADD,&EOF,&CT=1

09	SB	RC	CT
0	1	4	6

***	X	I	LOC
0	3	4	5

Hexadecimal numbers are read from the next input data card and stored in successive words of core starting at the location specified by the effective address. The number of numbers read is specified by the count field. (If the count field is zero, no card is read.) The numbers may appear anywhere on the card and must be separated by one or more blanks or commas. If a number has less than eight digits, leading zeroes are inserted. Negative numbers must be in hex complement form and must be eight digits in length.

Bad data, i.e., a number that contains an invalid hexadecimal digit, is too large to fit in a single word in core, or is more than eight digits long, causes status bit 4 to be set. The contents of the word where the number was to have been stored remain unchanged, the GETX terminates, and the return count is set to the count of numbers remaining to be read. If there are fewer numbers on the card than specified in the count field, status bit 3 is set and the return count field is set to the number of missing numbers. If there are more numbers on a card than specified in the count field, the extra numbers are ignored. On the first attempt to read beyond the end of file (no data cards remain), status bit 7 is set; on the second, the program abends. With the system macro, transfer is made to the location specified by &EOF when end of file is detected.

Put Decimal

```
PUTD     &ADD,&CT=11
```

0A	SB	**	CT
0	1	4	6

***	X	I	LOC
0	3	4	5

The contents of the location specified by the effective address are converted into an eleven character decimal number with floating sign (if negative) and leading blanks (i.e., the sign always precedes the most significant digit). The number of characters specified in the count field are taken from the low order positions and written into the buffer starting at the current carriage position.

Overflow off the right end of the carriage causes a "RET 0" to be done; the digits which did not fit in the buffer are lost, and status bit 5 is set. Overprinting causes status bit 6 to be set.

Get Decimal

```
GETD     &ADD,&EOF,&CT=1
```

0B	SB	RC	CT
0	1	4	6

***	X	I	LOC
0	3	4	5

Optionally signed decimal numbers are read from the next input data card, converted into two's complement representation, and stored in successive storage locations starting at the effective address specified. The number of numbers to be read is specified by the count field. (If the count field is zero, no card is read.) The numbers may appear anywhere on the card separated by one or more blanks or commas.

All error conditions are as specified for GETX, except that only decimal numbers are valid.

Put Characters
PUTC &ADD,&CT=4

0C	SB	**	CT
0	1	4	6

***	X	I	LOC
0	3	4	5

The string of characters starting at the location specified by the effective address, and containing the number of characters specified by the count field, is written into the buffer starting at the current carriage position. The string may extend across as many consecutive word boundaries as necessary and need not end on a whole word boundary. Error conditions are as in PUTX.

Get Characters
GETC &ADD,&EOF,&CT=4

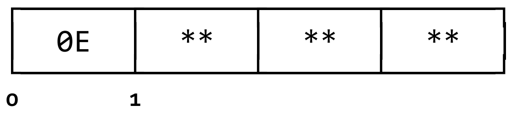
0D	SB	**	CT
0	1	4	6

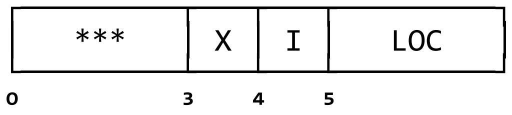
***	X	I	LOC
0	3	4	5

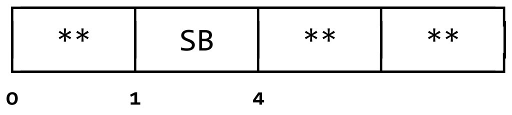
A string containing the number of characters specified by the count field is read from the next input data card and stored contiguously starting at the location specified by the effective address. Characters are read starting in column one of the card. If the number of characters is not a multiple of four, the remainder of the last word is padded with blanks. If the count field is greater than 80, only 80 characters will be read.

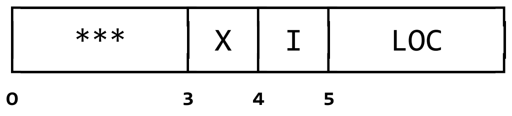
End of file conditions are as specified for GETX.

Partial Dump
PDUMP   &ADDFR,&ADDTO





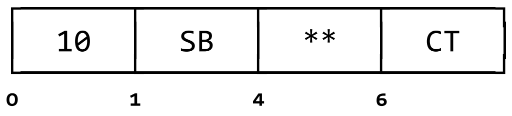


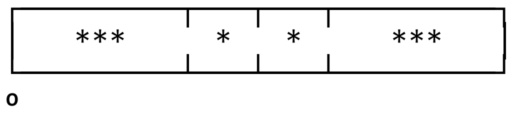


Two consecutive CCW's are needed. The contents of memory between and including the locations specified by the first and second effective addresses are printed in hexadecimal and character format. The address of the SVC which initiated the operation is printed as an identifier. No RET is needed and the contents of the buffer remain unchanged. The chain bit of the first CCW is ignored; the chain bit of the second CCW determines whether chaining is desired.

If the second effective address is less than the first, the program abends. If they are equal, only one word is dumped.

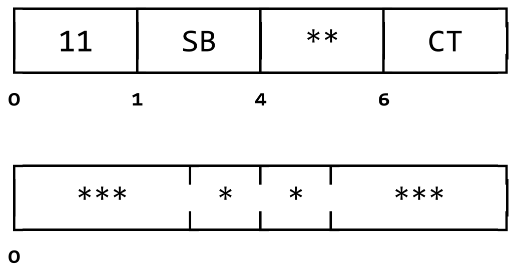
Trace Off
TOFF   &CT





The count field is tested. If it is zero, the trace is turned off. (It remains off if already off.) If the count field is greater than zero, it is decremented by one and returned to the CCW. No change is made to the status of the trace in this case. The effect of this is to turn the trace off on the n+1st time it is executed, where n is the original count. This facility can be used, for example, to trace selected iterations of a loop. If &CT is zero, it may be omitted from the macro invocation.

Trace On
TON   &CT



The count field is tested. If it is zero, the trace is turned on. (If it was already on, it remains on.) If the count is greater than zero, it is decremented by one and returned to the CCW. No change is made to the status of the trace in this case. The effect of this is to turn the trace on the n+1st time it is executed. If &CT is zero, it may be omitted from the macro invocation.

4 MACHINE CODE PROCESSOR

The SOS machine code processor is a facility for directly loading a machine language program into memory and controlling its execution. Each word to be loaded is encoded as an eight digit hexadecimal number starting in column one. Column nine of each instruction must be blank, and columns ten through 72 may be used for comments. An entire card (through column 72) is treated as a comment if an asterisk (*) appears in column one.

In order to properly load and execute a program, it is necessary to know where to load the program, and what instruction is to be executed first. Two special operation codes, called pseudo-ops, are used to pass this information to the machine code processor.

FF0 n

Sequentially load the following instructions starting at location n, until the next pseudo-op. "n" must be a valid hexadecimal number of from one to three digits (in order to provide a valid SOS address). It must be separated from the "FF0" by one or more blanks. Columns one through four for this pseudo-op should thus contain "FF0 ". Comments may be used, separated from the operand by one or more blanks. If an initial FF0 card is omitted, loading will begin at location X'10'.

FFF n

Stop loading, and execute the program that has been loaded (provided that the error limit has not been exceeded), with execution starting at location n. The format is the same as described above for the FF0 pseudo-op (with, of course, "FFF " in columns one through four). Every machine code program must end with an FFF pseudo-op.

For example, the following code might be used to load a program and calculate 2 + 2:

```
FF0 39
* CALCULATE 2 + 2
5010003C  GET THE NUMBER
1010003C  ADD IT TO ITSELF
00000000  STOP
00000002  THE CONSTANT 2
FFF  39   START EXECUTION AT LOCATION 39
```

Examples of invalid machine code processor pseudo-ops:

```
FFO  3A            letter "O" typed instead of number "0"
FFF  1234          operand too long
FF0    12Q         non-hexadecimal character in operand
FFF                missing operand
```

5 ASSEMBLER

The assembler can be characterized as a program whose input is an assembly language _source_ program consisting of symbolic instructions. The assembler translates these statements into an _object_ program consisting of machine language instructions, which are then executed by the SOS machine. Five of the assembler's major functions are:

1) Converting mnemonic opcodes to machine language opcodes;
2) Converting labels to storage addresses;
3) Translating symbolic expressions in the operand field to their numeric equivalents;
4) Defining constants and literals, and reserving storage areas;
5) Creating a cross-reference table of all symbols used by the program.

In addition to allowing symbolic specification of machine instructions, the assembler provides a number of assembly time directives to control its operations, called pseudo operations (pseudo-ops) since they don't cause machine instructions to be generated. Among these are the storage pseudo-ops (DC - Define Constant, DS - Define Storage space), the logical equate (EQU), the equivalents to the machine processor's FF0 and FFF pseudo-ops (ORG and END), and listing control pseudo-ops (PRINT, SPACE, TITLE, EJECT). These pseudo-ops are described in section 5.2.

5.1 Assembler Specifications

5.1.1 Fields

SOS assembly language statements are in free format, in the first 72 columns of each card. The assembler processes information in four fields:

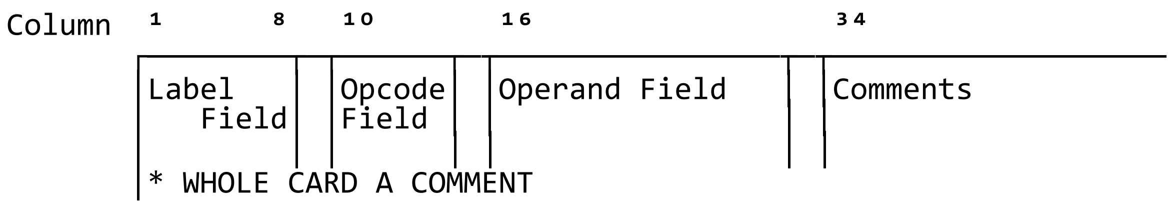
1) Label field: This field contains an optional symbol (called a label, of course), which must begin in column one. The field itself extends up to the first blank on the card, for a maximum of eight characters. Thus a statement with no label has a null label field.

2) Opcode field: The field beginning at least one blank column after the label field, starting with, and extending through the opcode. This field may contain a mnemonic opcode, assembler pseudo-op, or macro name.

3) Operand field: The field, beginning at least one blank column past the opcode field, which encompasses the operand(s).

4) Comments field: The field beginning one blank column past the end of the operand field, containing the remainder of the card. This field is used for comments pertinent to the instruction on

the card. An entire card (up to and including column 72) can be used for a comment by placing an asterisk (*) in column 1.



Recommended Format of Typical Assembly Language Statement

Since a valid label field can consist of a maximum of eight characters, the opcode field can start in column ten. In addition, since all SOS assembly language instruction opcodes and pseudo-ops are no greater than five characters in length (though macro names may be up to eight), if the opcode begins in column ten, the operand field can begin in column sixteen. Of course, for a macro name of greater than five characters, the operand field must begin past column sixteen. Comments normally begin in, or after, column 34.

In general the opcode and operand fields must be present on every card not containing an asterisk in column one. All other fields are optional. See sections 2.3 and 5.2 for specific descriptions of the various instructions and pseudo-ops.

Example:

```
LAB      ST    R3,LOCAT(R7)       SAVE XPOS
```

Label Field:	"LAB", the label
Opcode Field:	"ST", store
Operand Field:	"R3,LOCAT(R7)"
Comments Field:	" ...SAVE XPOS..."

5.1.2 Numeric and Character Constants

Constants may be used in a variety of ways in the operand field. There are four ways of representing them to the assembler.

1) A decimal constant is written as a sequence of one to ten decimal digits optionally preceded by a plus or minus sign. A decimal constant may range between -2147483648 and 2147483647, in order to fit into 32 bits.

Examples of valid decimal constants:

```
+3
-2
1000000000
```

Examples of invalid decimal constants:

```
32F5           illegal digit
```

2147483648	too large
99999999999	too long (and too large)

2) A <u>hexadecimal constant</u> is written as "X'" followed by one to eight hexadecimal digits (0,1,...,9,A,B,C,D,E,F) followed by another "'", optionally preceded by a plus or minus sign. If fewer than eight digits are specified, the assembler assumes that the number is to be padded on the left with zeroes (leading zeroes are to be inserted). The hexadecimal constant must be in eight-digit hex-complement notation (see Appendix A). Thus, to express a positive number in hex, leading zeroes may be omitted, but to express a negative number (without using a leading minus sign), any leading F's must be written (up to the full eight digits).

Examples of valid hexadecimal constants:	(decimal equivalent):
X'5'	5
X'3B'	59
X'FFFFFFFB'	-5
X'100'	256
-X'3F'	-63
X'1000'	4096
X'10000'	65536

Examples of invalid hexadecimal constants:

X'3BG'	invalid hex digit
X'-5'	invalid hex digit
X'123456789'	too many digits
X''	no hex digits

Note: X'F123456' represents a positive number (one zero digit of padding is inserted) whereas X'F1234567' represents a negative number.

Logical data, i.e., masks and specific bit patterns, are most frequently written as hexadecimal constants.

3) A <u>C-type character constant</u> is written "C'" followed by 1 to 65 characters followed by another "'", and is typically used to define messages to be printed. The restriction to 65 characters is imposed by the limit of 72 columns that may be used on any one card. Clearly, if the recommended instruction format is used, C' will be located in columns 16 and 17, and the limit will be 57 characters on one line. A C-type constant may only be used in the operand field of a DC pseudo-op or in a literal. It <u>may not</u> be used as a term in an expression. The characters enclosed in the apostrophes are converted to their EBCDIC equivalents (see Appendix D) and are stored four to a word. If the number of characters enclosed by the apostrophes is not a multiple of four, the constant is padded on the right with enough blanks to make it a multiple of four.

Example:

```
         DC    C'ABCDE'
```

assembles as two words, whose hexadecimal representations are X'C1C2C3C4' and X'C5404040'. If no characters are specified between the apostrophes, however, a full word is assembled anyway. Thus "DC C''" assembles to a word whose hexadecimal representation is X'40404040'.

To represent the apostrophe character in a C-type character constant, two consecutive apostrophes must be used. The first unpaired apostrophe (following the initial "C'") marks the end of the C-type character constant.

Examples:	Hexadecimal Equivalent (X'7D' is an apostrophe)
C'IT''S'	X'C9E37DE2'
C''''	X'7D404040'
C'''A''B'	X'7DC17DC2'
C'A'' '''	X'C17D407D'
C''''''	X'7D7D4040'
C'ERROR	Error: no second apostrophe

Note however, that for

```
         DC    C'THE BOY'S DOG'
```

only two words are assembled: X'E3C8C540' and X'C2D6E840' because the constant ends at the second apostrophe. The remainder of the line is treated as the comments field and is thus ignored.

The C-type character constant is used to define messages to be printed via the Put Character I/O operation. The hexadecimal equivalent of only the first word of a multiword character constant is printed on the assembly listing.

4) The A-type character constant is written "A'" followed by zero to four characters followed by another "'", and is generally used to compare an unknown character string to a known one. The characters are translated to their EBCDIC equivalents. If fewer than four characters are specified, hexadecimal (not character) zeroes are padded on the left to fill out a whole word. (If no characters are specified, the value is zero.)

Examples:	Hexadecimal Equivalent
A''	X'00000000'
A'A'	X'000000C1'
A'AB'	X'0000C1C2'
A'1234'	X'F1F2F3F4'
A'12345'	Error: too many characters

The restriction to four characters is imposed by the fact that the value of an A-type constant must alway fit into a fullword. Representation of apostrophes within A-type constants follows the rules given above for C-type character constants.

The A-type character constant, unlike the C-type, may be used in expressions. For example, suppose R1 contains one (unknown) character, i.e., is of the form 000000XX. Then the instructions

```
         AXAI  R1,-A'!'
         BE    R1,FOUNDIT
```

will transfer execution to location FOUNDIT if R1 contained the character representation of an exclamation point (since the addition will result in zero in such a case). In this way, A-type constants can be used for character comparisons.

5.1.3 Symbols and Address Constants

A symbol may be used in the operand field of an instruction or pseudo-op, thus permitting references to constants and locations. In order to be used in the operand field, a symbol must be given a value by being placed in the label field of (a) an EQU pseudo-op, in which case its value is that of the EQU's operand; or (b) some other assembly language statement, in which case its value is that of the location counter when that instruction is assembled. For the most part, a symbol may be given its value anywhere in the program. For an ORG, DS, or EQU pseudo-op, however, any symbol used in the operand field must have been given a value _earlier_ in the program. If R2 = 2 [4] and LAB1 = X'123', the statement

```
         L     R2,LAB1
```

will result in evaluating the symbol R2 to yield two, the symbol LAB1 to yield X'123', and thus the machine language instruction X'50200123' will be generated. In a less clear example,

```
         DC    LAB1
```

will generate the word X'00000123'. The operand field of the DC pseudo-op is evaluated, which entails evaluating the symbol LAB1, and thus results in X'00000123' as the value of the DC. A word generated in this way is called an _address_ _constant_ -- its value is the numeric address of LAB1. Then, if it were encountered through indirecting during effective address calculation, it would reference the word at location LAB1. Note, however, that if an address constant were _executed_ during the sequential flow of control, the program would terminate, since the opcode of zero is that of a Halt instruction.

Address constants may also be generated by the Halt instruction ("H"). The statement

```
         H     LAB1
```

[4]This notation (in general, symbol = number) is used to indicate that the symbol has been given the value "number." In this case, it means that the symbol R2 has the value 2, _not_ that the contents of register 2 are 2.

will generate a word with LOC field X'123', opcode of X'00', and index, indirecting and register fields of zero, a word identical to that generated by "DC LAB1". The use of "H" to create address constants is slightly more powerful than the use of "DC" -- since the operand of a Halt instruction is an address, indexing and indirecting may be specified. Thus

```
         H     >LAB1(R11)
```

would produce the word X'000B1123' and, if encountered through indirecting during effective address calculation, would cause indexing using register 11 to be applied, along with another level of indirecting. Address constants are used in parameter passing, as in the demo on subroutine conventions.

Since the use of "H" depends on the fact that its opcode is zero, the use of the DC pseudo-op is recommended, and will produce the desired result in most cases.

5.1.4 Expressions and Relative Addressing

Any subfield of the operand field of an assembly language statement may be more complex than just a single constant or symbol. It may contain an expression consisting of sums ("+") and/or differences ("-") of decimal, hexadecimal and A-type constants, symbols, and the value of the location counter (represented by "*"), with the whole expression optionally preceded by a plus or minus sign.

At assembly time the assembler converts such expressions into their numeric equivalents and constructs a machine language instruction out of them. For example,

```
         L     X'4'-2+3,4+A'0'(45-43)
```

assembles to the machine code instruction X'505200F4'. There may not be two or more consecutive operators in an expression. Thus an expression such as

```
               3-+X'A'
```

is not valid, even though "+X'A'" is a valid hexadecimal constant.

In general, the only restriction on expressions is that the resulting numeric equivalent must be able to fit in the specified field; e.g., an expression used for a register or index field must evaluate to a number from 0 to X'F' and an expression in the LOC subfield must fit into three hexadecimal digits.

The use of expressions and "*" in the LOC subfield permits "Relative Addressing." Supposing that LAB1 = X'1AC', the instruction

```
         L     5,LAB1+2
```

will be assembled as X'505001AE' which references a core location that, relative to LAB1, is two words away. Then, if LAB2 = X'056',

```
         L     10-5,LAB1+LAB2
```

will be assembled as X'50500202', referencing the location LAB2 words after location LAB1.

A final facility under relative addressing is the use of "*", which represents the value of the location counter for the instruction currently being assembled. For example, if the instruction

```
         B     *+5
```

is to be loaded into location X'055', "*" will equal X'055' when this instruction is being assembled and the machine code generated will be X'9000005A' (*+5 = X'055' + 5 = X'05A'). When executed, this instruction will cause a branch to be taken to the location five words past the current location.

If indexing is used, note that the character immediately following the ")" begins the comments field. Thus

```
         A     R1,LAB1(R3)+2
```

will be assembled to 501301AC, as will

```
         A     R1,LAB1(R3)
```

If the user intended the former instruction to reference two words past location LAB1, the proper format would be

```
         A     R1,LAB1+2(R3)
```

5.1.5 Literals

A "literal" is used to specify data directly (i.e., literally) in the address subfield of the operand field of an assembly language instruction. The assembler sets up a "literal pool" to hold all literals, each of which is written as an arbitrary expression or a C' ' character string, preceded by an "=" sign. The assembler evaluates the expression or string, places the value into memory, and places the address of the memory location into the LOC field of the machine word. If several statements use the same literal, that value is defined only once and its location in the literal pool is used in the LOC field of each of those statements.

Examples:

```
         L     R3,=15            Set register 3 to 15
         L     R4,=X'7E0000'     Set register 4 to X'007E0000'
         A     R5,=3+X'10'       Add decimal 19 to register 5
```

Any instruction having an address subfield of the operand field may use a literal instead. If this is done, however, indexing and indirecting <u>may</u> <u>not</u> be used.

The starting location of the literal pool is the largest value taken on by the location counter during assembly. In general, this will be its value when the END pseudo-op is encountered; however, if the ORG pseudo-op has been used, the literal pool could start at a higher address. Note that if, for whatever reason, the highest value of the location counter is near the end of core, there may not be enough room for the literal pool, in which case it will overwrite itself and print a warning message.

<u>5.2 Pseudo-ops</u>

```
          ORG   Expression
```

The ORG pseudo-op resets the location counter to the value of the expression and the assembler continues loading sequentially from this location until another ORG or an END pseudo-op is encountered. All symbols in the expression must have been defined earlier in the program and the expression must produce a valid SOS address. An ORG pseudo-op telling the assembler where to begin loading the program is optional, and may be omitted. If it is omitted, loading will begin at the location following the highest numbered register (X'10'). (See also the END statement below.)

```
          END   Expression
```

The END pseudo-op indicates to the assembler that assembling and loading are to be stopped, and execution is to begin at the specified address. (Note that this is different from the /END used for Job Control.) <u>Every</u> SOS assembly language program must end with an END pseudo-op. If the END card appears without an operand, the default operand of X'10', which is the default loading address, is used.

The expression specified on both ORG and END pseudo-ops may be either absolute addresses (ORG X'60', END 6), or arbitrary assembler expressions (END START). Also, as is the case with machine language pseudo-ops, ORG and END are not machine instructions, and therefore are not, and cannot be, loaded into memory.

```
Label     DC    Expression
          or
Label     DC    C-type Character Constant
```

The first of the above forms tells the assembler to reserve one storage location and to set this location to the value of the expression specified in the operand field. Any legal assembler expression may be used.

```
Examples:              Hexadecimal equivalent generated

         DC    16                 00000010
         DC    -1                 FFFFFFFF
         DC    X'6A4'+1           000006A5
```

The second form of the DC pseudo-op tells the assembler to reserve one or more words of storage for the C-type character constant specified.

The DC instruction is used to define in-line data (constants used by the program which are programmer-supplied, and not read in from the card reader by the program). The label is optional.

```
Label    DS    Expression
```

The DS instruction tells the assembler to reserve a number of words in core equal to the value of the expression in the operand field. The DS operand must not cause the address limit of X'FFF' to be exceeded. All symbols in the expression must have been previously defined.

Examples:

```
         DS    1        Reserve one word of storage.
         DS    2        Reserve two words of storage.
         DS    N        Reserve  N  words of  storage, provided that
                        the symbol N has been given a value earlier.
```

DS is usually used to reserve areas for intermediate results or areas into which data will be read from the card reader. The label is optional. Note that no preset values are assigned to DS'd locations, so no assumptions about their contents should be made.

```
Label    EQU   Expression
```

The EQU instruction sets the value of the label equal to the value of the operand expression. All symbols in the expression must have been previously defined. An example of equating a register to a convenient mnemonic is

```
R2       EQU   2
```

```
         PRINT GEN
```

This causes the code generated by all following macros (section 7) in the program to be listed until the next PRINT NOGEN is encountered. PRINT OFF, however, overrides this option.

PRINT NOGEN

This causes the code generated by macro calls not to be listed until the next PRINT GEN is encountered. (PRINT NOGEN is the assumed option if neither is specified.) If a macro-generated statement contains an error, however, it will be printed regardless of print status.

PRINT ON

This causes all following lines of the source program to be listed, until the next PRINT OFF.

PRINT OFF

This suppresses the printing of all following lines of the source program, until the next PRINT ON. However, any statement which produces an error will be printed regardless. PRINT ON and PRINT OFF are used to reduce the amount of output produced by omitting sections of code that need not be looked at. (PRINT ON is the assumed option.)

SPACE n

Insert n blank lines in the listing. If n is omitted or zero, the default of 1 is used. Otherwise, n must be a one or two digit unsigned decimal number. If n exceeds the number of lines left on the current page, a new page is started and no additional blank lines are inserted. The pseudo-op itself is not printed.

TITLE 'character string'

During assembly time a heading is printed at the top of each page; the first line of this heading contains the title. This pseudo-op causes the old title to be replaced by the character string specified between the apostrophes, and a new page to be started. Representation of apostrophes within the string follows the rules defined for C-type and A-type character constants. The pseudo-op itself is not printed. Note that this pseudo-op has no effect on the heading printed during execution, which is controlled by the HEAD command (section 3.3.2).

EJECT

Start a new page and print the current heading. The pseudo-op itself is not printed.

6 EXECUTION CONTROL: /SOS, /DATA, /END

In order to run programs, the user has to identify a program, to separate one program from another, and to specify what is to be done with the program, its data, and its output, before and after it is executed. This information is supplied to the SOS system by using Job Control Language (JCL).

JCL statements are characterized by a "/" in card column one. There are three such statements: /SOS, /DATA, /END. No other cards with a "/" in column one are treated as JCL.

(1) A /SOS card must be placed at the beginning of the program;
(2) a /DATA card must be placed before any data to be read by the program (but may be omitted if no data is included);
(3) a /END card must be placed at the end of the data (if included) or the program (if not).

The /SOS card is used to specify what is to be done with the program by means of a set of bookkeeping parameters and options, which starts in column six or beyond and continues to column 72. If more space is needed, another /SOS can be coded on the following card, and the options continued in column six or beyond of that card. Individual parameters may be separated by any number of commas and/or blanks, and may appear in any order. If a parameter is specified more than once, the <u>last</u> specified value is used.

In the descriptions below, those parameters which have a numeric argument are indicated by nnn following the parameter name. All numbers must consist of one to three decimal digits, optionally followed by a "K" which indicates that the number is to be multiplied by 1000 (e.g., TRCNT6, TRCNT5K, and TRCNT411 set the trace count to 6, 5000, and 411, respectively).

Two system variables, SOSDFLT and SOSMAX, each a four element vector, are used in assigning counts to the PRINT, ERROR, INSCT, and TRCNT parameters. SOSMAX contains the maximum values allowable for these parameters; if a specified option exceeds the value in SOSMAX, the value in SOSMAX is used instead. SOSDFLT contains the default values; if these are not explicitly stated on the /SOS card, the values in SOSDFLT are used.

Underlined options are the defaults.

<u>TRACE</u>, NOTRACE
The trace is initially set on (TRACE) or off (NOTRACE). The trace status may be changed within the program by execution of a TON or TOFF CCW.

TRCNTnnn
The maximum number of traceable instructions for the program is set to nnn. If nnn is exceeded during execution, tracing will be forced off, but execution will continue. nnn cannot be set greater than SOSMAX(1) and defaults to SOSDFLT(1).

LINCTnnn
The number of lines to be printed before a new page is started. The default is 55 lines per page. nnn cannot be zero.

INSCTnnn
The maximum number of instructions executed by the program is indicated in nnn. If it is exceeded, execution is halted and a dump is produced. nnn cannot be greater than SOSMAX(3), and defaults to SOSDFLT(3).

PRINTnnn
The maximum number of lines to be printed by an SOS program <u>during</u> <u>execution</u> is specified by nnn. If the maximum is exceeded, execution halts and a dump is produced. nnn cannot exceed SOSMAX(2) and defaults to SOSDFLT(2).

<u>DUMP</u>, NODUMP
SOS provides a dump of user memory at the end of a run in order to aid in the debugging of programs. If NODUMP is specified <u>and</u> execution terminates via a Halt instruction, the final dump will be suppressed. All other situations will produce a dump.

<u>WARN</u>, NOWARN
WARN indicates that execution messages of warning severity (e.g., trace count exceeded, arithmetic overflow) are to be printed in the listing. NOWARN suppresses the printing of warning messages.

<u>LIST</u>, NOLIST
LIST indicates that a program listing, consisting of the card images read, the machine code which is assembled, the location counter values for each statement, a literal pool (if ASM) and, if XREF was specified, a cross reference table, is to be generated for this program. NOLIST suppresses this listing.

<u>XREF</u>, NOXREF
XREF indicates that a cross-reference table, consisting of all labels defined in the program, their values, the statement at which they were defined, and a list of all statements in which those labels are used is to be printed. NOXREF suppresses this table. NOXREF is forced if MACHINE and/or NOLIST are specified.

<u>DATA</u>, NODATA
DATA indicates that the card-image data following the /DATA card are to be printed with the listing. NODATA suppresses this.

MACHINE, <u>ASM</u>
MACHINE indicates that the program is in SOS machine code. ASM indicates that the program is in SOS assembler language. MACHINE implies and forces NOMACRO and NOXREF.

<u>MACRO</u>, NOMACRO
MACRO indicates that the SOS system macro library is to be accessed for macro calls (the system library contains the I/O macros described under CCW's as well as several debugging macros). NOMACRO indicates that the system macro library will

not be accessed. NOMACRO is forced if MACHINE is specified. NOMACRO does <u>not</u> prevent the use of user-defined macros, nor does it prevent the use of the SOSREG macro.

ERRORnnn
If nnn or more errors occur during assembly or machine code processing, the interpreter will not be invoked. nnn cannot be set greater than SOSMAX(4), and defaults to SOSDFLT(4).

The following example illustrates the basic setup of an SOS job:

```
/SOS LIST    NOXREF, PRINT3K
/SOS   NOTRACE  ,, ,, TRCNT722, INSCT12K MACHINE
     •
     (SOS machine language program)
     •
/DATA
     •
     (Data cards)
     •
/END
```

SOS is a batch processing system - more than one job may be included and run consecutively within one input stream. This results in a time saving to the user, since some of the system's initialization need not be repeated for the jobs following the first. Each job within the input stream must have the JCL described above, and must be placed immediately following the end of the preceding job (the /END card). Thus, note the following batch processing example:

```
/SOS NOTRACE NODUMP
     •
     (SOS assembly language program)
     •
/END
/SOS TRACE INSCT30K ASM
     •
     (SOS assembly language program)
     •
/DATA
     •
     (Data cards)
     •
/END
```

The first job above contained no data cards to be read and so the /DATA card was not needed (or used). Also, the default of ASM was applied to that job, since neither ASM nor MACHINE was specified.

7 MACRO GENERATOR

7.1 Introduction

The macro generator is a powerful facility used in conjunction with the assembler, allowing a programmer to generate similar, though not necessarily duplicate, sections of code from a previously defined template. Macros free the programmer from retyping code which is repeated numerous times by allowing the code to be defined once, and providing facilities to recall and alter certain aspects of the definition. In one sense, a macro definition extends the repertoire of the assembler's instruction set, by allowing the definition of new "operations" in terms of known ones.

In the macro definition, symbolic (dummy) parameters may be specified. When the macro is called (invoked), the calling statement provides values which are to be substituted for these dummy parameters.[5] This is similar to defining a subroutine in terms of dummy variables, and then invoking the subroutine with specific values which replace the dummy values and allow the subroutine to be executed.

For example,

```
         MACRO
&LABEL   MOVE  &FROM,&TO,&REG
&LABEL   L     &REG,&FROM
         ST    &REG,&TO
         MEND
```

defines the macro MOVE, with parameters &LABEL, &FROM, &TO, and &REG, that generates a Load statement and a Store statement. If this definition were invoked with the macro call

```
SHIFTIT  MOVE  VALUE,RESULT,R7
```

the following code would be generated:

```
SHIFTIT  L     R7,VALUE
         ST    R7,RESULT
```

The macro generator, when processing a macro call, is only concerned with <u>text substitution</u>. It is <u>not</u> concerned with whether or not the statements it generates are syntactically correct. It merely does the substitutions and passes the statements back to the assembler, which processes them.

SOS provides several pre-defined system macros. Aside from the previously-mentioned I/O macros, the macro SOSREG has been set up to

[5]Those familiar with text editing systems (specifically, the CMS editor) can think of macro processing as a global change, in which all occurrences of each symbolic parameter in the definition are replaced by the value given in the macro call.

generate the EQU's for all sixteen register mnemonics. It is invoked by the statement

```
         SOSREG
```

7.2 Format of a Macro Definition

7.2.1 MACRO and MEND

The first statement of a macro definition (the header) is the MACRO statement, consisting of the string "MACRO" in the opcode field. The last statement (the trailer) is the MEND statement, which has the string "MEND" in the opcode field. Neither the header nor the trailer may contain any other non-blank characters. The MACRO and MEND statements can be thought of as parentheses or delimiters for a macro definition.

7.2.2 Macro Prototype Statement

This statement must _immediately_ follow the MACRO statement. It provides the name of the macro being defined, as well as the names of any parameters which may be passed to the definition when it is called.

7.2.2.1 Macro Name

The name of the macro is the opcode field of the prototype. A macro name consists of up to eight alphanumeric characters, the first of which must be a letter. It may be neither an assembler opcode nor a pseudo-op name.

7.2.2.2 Symbolic Parameters

The operand field of the prototype statement may contain zero or more symbolic parameters, separated by commas. The name of a symbolic parameter is written as an "&" followed by from one to seven alphanumeric characters. There are two types of symbolic parameters -- Positional and Keyword. A positional parameter is signified by writing its name. A keyword parameter is specified by its name followed by an equal sign, followed by a string of zero or more characters. This string is the default value of the parameter. See section 7.3.1 for examples of the use of symbolic parameters.

7.2.2.3 Label Parameter

There may be a positional symbolic parameter in the label field of the prototye, or the label field may be left blank.

7.2.3 Model Statements

The body of the macro definition consists of all the statements, called "model statements," between the prototype statement and the MEND statement. The only restriction placed on a model statement is that the length of the statement after expansion must be less than or equal to 72 characters. If it is greater, the expanded statement is truncated and then processed normally by the assembler.

7.3 Invoking a Macro Definition

A macro is invoked by a statement with the macro name in the opcode field, and the values of any symbolic parameters in the label and/or operand fields, in accordance with the format of the prototype.

7.3.1 Assigning Values to Symbolic Parameters

If the prototype has a label parameter, the label field of the macro call is assigned to this symbolic parameter. Thus, if there is no label on the macro call, the null string is assigned to the label parameter. If there is no label parameter in the prototype, the label field of the call is ignored.

Positional parameters are given their values by writing the values in the order in which the parameters appear in the prototype statement. For example, the prototype

```
&LAB     TEST  &A,&B,&C,&D
```

and the macro call

```
LOOP     TEST  711,LABWL,,'HOW SWELL'
```

will cause the following relations to be established:

&LAB is assigned the character string "LOOP";
&A is assigned the character string "711";
&B is assigned the character string "LABWL";
&C is assigned the null string; and
&D is assigned the string "'HOW SWELL'".

The double commas are necessary to assign the null string to &C because the macro generator takes the value in the third position of the operand of the call and assigns it to the third positional parameter in the prototype. If only one comma had been used, &C would have been assigned the value "'HOW SWELL'" and &D would have been assigned the null string as its value, since any omitted positional values default to the null string. Indeed, the call

```
         TEST
```

assigns the null string to &LAB, &A, &B, &C, and &D. Although it is possible to assign the null string to _trailing_ positional parameters by omitting them altogether, it is not possible to do this to _leading_ positional parameters without including a comma to "hold the place" of the parameter.

Note that it would not have been possible to include a comment in the above macro call, since the comment field would have been taken as the operand field. In order to indicate a null operand field and have a comment as well, a single comma placed into the operand field will indicate that it is to be taken as null:

```
          TEST  ,                 THIS IS MY COMMENT
```

See section 7.5 for restrictions on this feature.

In general, the string to be assigned to a symbolic parameter is delimited on the right by a comma or a blank. Thus to pass a comma or a blank as a value, it must be within apostrophes ('HOW SWELL'). When an apostrophe is encountered, its closing apostrophe is searched for, regardless of what characters might be between the two apostrophes. Note that the apostrophes are _not_ removed by the macro generator; the whole string between delimiters is assigned as the value of the parameter. For example, the macro call

```
         TEST  'B,L,A'--'N K 'S,A
```

would assign the string "'B,L,A'--'N K 'S" to &A, "A" to &B, and the null string to &LAB, &C, and &D.

A keyword parameter is given a value in the invocation by using its name (without the "&") followed by an equal sign, followed by its value. Keyword parameters differ from positional parameters in two ways: first, because the order in which they appear is of no significance; and second, because they can be assigned a default value other than the null string. For example, given the prototype

```
         KEYMAC &F=DEF,&G=,&R=C'VA'' LUE'
```

and the macro call

```
         KEYMAC
```

the following assignments are made:

&F is assigned its default value, "DEF";
&G is assigned its default value, the null string; and
&R is assigned its default value, "C'VA'' LUE'".

For the call

```
         KEYMAC R=NEW,G=OLD
```

the following assignments are made:

&F is assigned its default value, "DEF";
&G is assigned the string "OLD"; and
&R is assigned the string "NEW".

Note that any number of keyword parameters may be omitted, in which case their default values are used.

Due to the dependence of positional parameters on the order in which they occur, all positional parameters must be placed before all keyword parameters, in both the macro prototype and the macro call. Thus given the prototype

```
&LAB     MIXED &A1,&B2,&C,&D3,&I=,&MARS=PLANET
```

and the macro call

```
         MIXED MOON,SOL,MARS=RED
```

the following assignments are performed:

&LAB is assigned the null string since no label was specified;
&A1 is assigned the string "MOON";
&B2 is assigned the string "SOL";
&C is assigned the null string because the trailing positional parameters were omitted from the call;
&D3 is assigned the null string for the same reason;
&I is assigned the null string, its default value; and
&MARS is assigned the string "RED".

An equal sign as the <u>first</u> character of a parameter is not regarded as signalling a keyword parameter, thus allowing literals to be passed as arguments to a macro:

```
          MIXED =X'4000',=C'DUSTY'
```

results in &A1 being set to "=X'4000'" and &B2 being set to "=C'DUSTY'".

An equal sign may not be passed within a positional parameter otherwise (unless it is within apostrophes, of course). In a keyword parameter, however, an equal sign following the first is considered to be part of the parameter and is processed as any other character.

7.3.2 Macro Body Expansion

After values are assigned to the symbolic parameters, the macro generator fetches the body of the macro and replaces all occurrences of each symbolic parameter within the model statements with the value of that parameter for the particular call. The substituted statements are then sent to the assembler, which checks them for correctness and assembles them.

Since an "&" denotes the beginning of a symbolic parameter, problems could occur in generating a literal ampersand. Thus, two ampersands must be coded in a model statement to indicate one literal ampersand. For example, "&&" in a model statement would generate "&"; "&&&PARM" would generate an ampersand followed by the value of the symbolic parameter &PARM; and "&&&&&VAL&&" would generate two ampersands, followed by the value of &VAL, followed by one more ampersand.

The macro generator does extremely little error checking, leaving this to the assembler. Specifically, only invalid or unknown parameter names, and statements which may not be generated by the macro generator (MACRO, MEND, END, or a Job Control Language statement) are checked for.

7.4 Placement of Definitions and Invocations

All macro definitions must be placed at the top of the assembly language program. The only statements allowed before and between macro definitions are comments, SPACE, TITLE, PRINT, and EJECT. Any other statement will cause an error when the succeeding definitions are encountered. Thus a sample deck may look like:

```
*
* THIS IS THE FIRST MACRO
*
         MACRO
         <prototype>
         <body>
         MEND
         SPACE 5
*
* THIS IS THE NEXT MACRO
*
         MACRO
         <prototype>
         <body>
         MEND
         EJECT

         etc.
```

A macro call may appear anywhere that an assembler opcode or pseudo-op may appear, since the macro is, in effect, a "new" opcode.

7.5 Continuing the Prototype and Invocation Statements

Since keyword parameter default values and parameters given in a macro call are sometimes quite long, the macro prototype statement and macro invocation statements may be continued onto a second card. These are the only two statements processed by the assembler or macro generator which may be continued.

The operand field of a macro prototype or call is continued by breaking the field off at a comma which separates one parameter from another, and then continuing the operands in or beyond column two of the following card. For example,

```
OTHERLAB MOVE  LABEL,
               OFF,R12
```

is a valid continuation of a macro call of MOVE (which was defined previously).

In order to indicate continuation, there must be at least one parameter on the first statement, since a single comma in the operand field indicates a null operand field. Only one continuation card is allowed for each macro prototype or call.

7.6 Concatenation

Any field of a model statement may contain a series of characters and/or symbolic parameters concatenated together. Symbolic parameters may be concatenated with character strings or with other symbolic parameters in one of two ways: either by juxtaposing them, or by separating them with a period. For example, if the value of &H is "R" and the value of &NUM is "1", then

```
     &H&NUM    produces R1
     &H.&NUM   produces R1
     &H.B      produces RB
     B&H       produces BR
     &NUM.1    produces 11
     1&NUM     produces 11
```

The periods are necessary in the third and fifth examples because merely writing &HB, for example, would cause the macro generator to look for a symbolic parameter called &HB. Thus the period denotes the end of the symbolic parameter name, and is consequently _not_ placed into the generated statement. The "&" acts as a delimiting character in the other cases.

All characters other than alphanumerics delimit symbolic parameters. Thus

```
                    &A+&B+&C&D
```

expands to the value of &A, concatenated to "+", concatenated to the value of &B, concatenated to "+", concatenated to the value of &C, concatenated to the value of &D. In order to concatenate a period to a parameter, say &H, it must be written as "&H..", since the first period delimits the parameter name.

7.7 Comments

Two types of comments may appear as model statements. If ".*" appears in the first two columns of a model statement, the statement will _not be generated_ when the macro is called. If "*" is in the first column of a model statement, the statement will be treated as any other model statement: symbolic parameter substitution will be performed, etc. Naturally, the assembler will treat this statement as a comment card.

7.8 Nested Macro Calls

One macro may call another macro; that is, it may contain a model statement which, when expanded, has the opcode of another macro. This is called "nesting" macro calls.

When a macro call is encountered by the macro generator while it is expanding a definition, it suspends expansion of the current definition and begins to process the new call. Once the new call has been expanded completely, the rest of the first call is generated. For example, if the following macros are defined:

```
         MACRO
         ONE   &A,&B
* ONE STATEMENTS
         MEND
*
         MACRO
         TWO   &A,&B
* TWO STATEMENTS
         ONE   &B,7
* MORE TWO'S
         MEND
```

then the macro call

```
         TWO   ALL,SET
```

will generate

```
* TWO STATEMENTS
         ONE   SET,7
* ONE STATEMENTS
* MORE TWO'S
```

The invocation generates the comment "* TWO STATEMENTS", and then the statement "ONE   SET,7". The macro generator recognizes ONE as a macro, and suspends expansion of TWO in order to expand ONE. Once ONE has been expanded (by generating "* ONE STATEMENTS"), the generation of the TWO macro resumes, and "* MORE TWO'S" is generated. Note that the ONE macro did not have to be defined before the TWO macro was defined.

Nested macro calls can occur several levels deep; that is, a macro can invoke a macro which invokes a macro and so forth. There is a preset maximum nesting level of six, and a macro nested too deeply will be ignored.

7.9 &SYSNDX

The special macro symbol &SYSNDX is a counter which is initially 0000, and is incremented by one each time a macro call is encountered. It is a four digit decimal number in character string form. &SYSNDX is usually used to make unique labels for statements within a given level of a macro. For instance, the following statements:

```
         MACRO
&LABEL   CALL  &ROUTINE,&ARG
         BAL   R1,LAB&SYSNDX
         DC    C&ARG
LAB&SYSNDX BAL  R14,&ROUTINE
         MEND
*
         CALL  DORK,'HELLO'
*
         CALL  SCAN,'WHAT? ME WORRY'
```

would generate

```
         BAL   R1,LAB0001
         DC    C'HELLO'
LAB0001  BAL   R14,DORK
*
         BAL   R1,LAB0002
         DC    C'WHAT? ME WORRY'
LAB0002  BAL   R14,SCAN
```

Had &SYSNDX not been concatenated to LAB, the label LAB would have been generated twice, prompting an error message from the assembler.

The value of &SYSNDX remains constant within a macro expansion, so that if there is a macro call nested within a macro, &SYSNDX is incremented when the inner macro is encountered, and returns to its value for the outer call after this inner macro has been fully expanded.

Since &SYSNDX is a reserved macro generator keyword, it may not be used as the name of a symbolic parameter.

8 APPENDICES

A. Computer Arithmetic and Number Representation

1. Binary Numbers:

The most basic unit of physical computer memory is the bit, a unit which can be in one of two states: "on" or "off." This can be treated as representing a one (on) or zero (off). Hence a computer is capable of storing strings (sequences) of zeroes and ones. The number system which corresponds to this system is the binary, or base two,[6] number system. Consequently, all computer arithmetic is performed in this number system, which can be summarized by the following rules:

```
0 + 0 =  0          0 * 0 = 0
0 + 1 =  1          0 * 1 = 0
1 + 0 =  1          1 * 0 = 0
1 + 1 = 10          1 * 1 = 1
```

Given N bits, each of which has two possible values, the number of different values which can be represented by this sequence of N bits is 2**N,[7] and the range is then 0 to 2**N-1 . For example, if N is 3, 2**N is 8 and the range of unsigned integers is 000 to 111 (zero to seven in decimal, eight different numbers).

2. Two's Complement:

Representation:

Since negative as well as positive integers must be represented in the computer, the above scheme for representing numbers is not frequently used.

One scheme which might be used to represent negative numbers in a computer is called "signed magnitude." In this method, the high order bit (bit 0) of a word is used to indicate the sign (e.g., off for positive, on for negative), with the rest of the bits indicating the magnitude of the number. The problem with this system is that addition and subtraction require a variety of sign and magnitude checking algorithms, leading to slow and expensive implementation.

An alternative representation, called "two's complement," requires much simpler hardware due to a bit flicking trick.

The K's complement of an N digit integer is defined as

K**N - (the integer)

[6]This is sometimes referred to as radix two.
[7]Where ** indicates exponentiation

Thus the tens complement of 43 is 57,[8] of 57 is 43, of 043 is 957, and the two's complement of 01 is 11.

The 32-bit, two's complement representation of a non-negative integer less than or equal to 2**31-1 is simply the integer itself in signed magnitude or "true" notation. However, the representation of a negative 32-bit integer which ranges between -1 and -2**31 is the 32-bit two's complement of the magnitude of that integer. Thus, this method covers numbers in the range -2**31 to 2**31-1, all integers which can be represented in 32 bits using two's complement notation.

NOTE 1: Two's complement may be obtained by using the definition, or may be more simply obtained by complementing the number (by XORing it with a word of all ones) and then adding one to the result of the complement. This is a very useful trick, which is one of the reasons for the inexpensiveness of a two's complement system in computers.

NOTE 2: The high order bit may be tested to determine the sign of the integer since the two's complement representations of positive integers have a zero high order bit, and the two's complement representations of negative integers have a high order bit of one.

Thus the smallest positive integer is 00...0 (decimal 0), and the largest is 011...1 (2**31-1); the negative integer with the smallest absolute value is 11...1 (decimal minus one), while the one with the largest is 100...0 (-2**31). Furthermore, the padding bits between the sign bit and the most significant bit of the integer "propagate" the sign bit: all zeroes for positive numbers, all ones for negative numbers.

Arithmetic:

The addition and subtraction algorithms for two signed (two's complement) integers are now identical. For addition, add the two numbers and ignore overflow; for subtraction, take the two's complement of the second integer, add, and ignore overflow.

This is illustrated by subtraction in base 10:

43851 - 31067 = 12784
or
43851 + (100000 - 31067) - 100000 = 12784

The expression in parentheses is the ten's complement, and if 100000 is not subtracted, 112784 results. Ignoring the high order digit (which is, in effect, the same as subtracting it, due to the properties of a place value number system), however, results in the proper answer.

[8]10**2 - 43 = 57

Since only a finite number of integers may be represented in N bits (assuming N is finite, which it is for a computer), the results of some arithmetic operations may not be representable in the finite field allowed. For example,[9] the subtraction

01010110 - 00010111 = 00111111

is done by first converting 00010111 to its two's complement (namely 11101001), then adding, getting 100111111 as the answer. This is nine bits, the original eight bits and an overflow bit. As in the decimal case above, the correct answer is gotten by ignoring overflow.

Remember that in two's complement the most negative number for an eight-bit word is 10000000 and the largest positive number 01111111. Therefore, add two large negative numbers:

10000110 + 10000110 = 100001100

and ignore the overflow bit, with a result of 00001100, a positive number in two's complement, and also the wrong answer. Similarly, consider the addition of two large positive numbers:

01000000 + 01001001 = 10001001

There is no carry into the overflow bit, but the resulting number is a large negative number, not a positive number as desired. The machine uses an algorithm to decide if a correct operation was performed based on carries into the overflow and leading (high order) bits:

1) If there are carries into the high order bit, and also into the overflow bit, the answer is correct.
2) If there is no carry into the high order bit or the overflow bit, then the answer is correct.
3) Any other combination of carries into the high order and overflow bits indicates an incorrect answer. This type of error condition is correctly interpreted as overflow and signalled by the computer in some manner (setting a flag and, in the case of SOS, printing a warning message as well).

Note that in two's complement form the high order bit functions as the sign bit, while the most significant bit of the number lies to its right.[10]

3. Number Representation:

Although a computer works with binary numbers, it is much more practical and economical to write these binary numbers in terms of

[9]For simplicity, all examples are done with eight-bit words, but the principle is the same for 32-bit words.
[10]It is the first bit from the left which is the inverse of the sign bit.

other bases. One of the most common of these is the hexadecimal or base sixteen number system. This is made up of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, which correspond to the base 10 integers 0 to 15.

Hexadecimal (hex) numbers correspond directly to binary numbers, and are much easier to read and remember. For example, the two numbers

01111010001101011001111101001011
and
7A359F4B

are exactly equivalent. Hex numbers are formed simply by treating the binary digits as four bit groups, and assigning one hex digit for each four bits. This is possible because sixteen is a power of two; one merely factors the polynomial expansion of base two numbers by grouping by fours and factoring out powers of sixteen. Other common bases are octal[11] and decimal. Octal is also directly equivalent to binary, each octal digit being formed from three bits, but decimal is not, and must be converted through a much more cumbersome process.

Notice that hex is much more compact than either octal or decimal (that is, fewer digits are needed to represent the same number). Also note that two's complement is directly equivalent to hex complement.

In all cases, it should always be remembered that no matter how a number is represented externally for convenience, its machine representation is always a string of zeroes and ones.

4. Number Conversions:

A number in one base can always be converted to an equivalent number in any other base, although in some cases the conversion may be laborious, and, in addition, some precision may be lost. As has already been mentioned, binary numbers can easily be converted to octal or hex simply by grouping the number into threes or fours. Conversely, octal and hex numbers may be converted to binary by simply writing each octal or hex digit in binary.

```
OCTAL         5  |  6  |  7  |  4
                 |     |     |
BINARY     1 0 1|1|1 0|1 1|1|1 0 0
                   |       |
HEX           B    |   B   |   C
```

To convert from binary, octal, or hex to decimal, write the number as a polynomial expansion in its base and work out the result.[11] On the other hand, to convert from decimal to some other

[11]Base eight; digits 0, 1, ... 7.
[11]This can be proven by using the definition of a place value

base, divide the decimal number by the decimal equivalent of the highest power of the desired base that can be divided into the decimal number. Do the same to the remainder, proceeding similarly until no remainder is left. For example, the conversion from 853 (decimal) to binary proceeds as follows:

$853/2^9 = 1 + 341$ (remainder) $2^9 = 512$
$341/2^8 = 1 + 85$ (remainder) $2^8 = 256$
$85/2^7 = 0 + 85$ (remainder) $2^7 = 128$
$85/2^6 = 1 + 21$ (remainder) $2^6 = 64$
$21/2^5 = 0 + 21$ (remainder) $2^5 = 32$
$21/2^4 = 1 + 5$ (remainder) $2^4 = 16$
$5/2^3 = 0 + 5$ (remainder) $2^3 = 8$
$5/2^1 = 1 + 1$ (remainder) $2^1 = 4$
$1/2^1 = 0 + 1$ (remainder) $2^1 = 2$
$1/2^0 = 1$

Hence the binary result is 1101010101. Certain alternate procedures exist for various bases depending on special characteristics of these bases, such as the conversion from binary to hex and vice versa. However, the above procedure, though cumbersome, will work for conversion from any base to any other. An alternative method of decimal to binary conversion is the following:

$853/2 = 426 + 1$
$426/2 = 213 + 0$
$213/2 = 106 + 1$
$106/2 = 53 + 0$
$53/2 = 26 + 1$
$26/2 = 13 + 0$
$13/2 = 6 + 1$
$6/2 = 3 + 0$
$3/2 = 1 + 1$
$1/2 = 0 + 1$

Divide the decimal number by two; save the remainder and divide the quotient by two, continuing until the quotient is zero. The resultant string of remainders (starting with the last remainder) is the binary equivalent of the decimal number.

5. Floating Point Notation:

A special representation is used to gain a greater numerical range within a machine word with a fixed number of bits, at the expense of some precision. This representation is known as "floating point notation." While fixed point representation (binary integers) has no provision for exponents, floating point encodes both sign and exponent information in a manner similar to scientific notation.

An eight hex digit machine word is visualized as having two exponent digits, an implied hexadecimal point (analogous to a

---

numeral as a polynomial over a base.

decimal point), and a six digit mantissa (i.e., XX.DDDDDD). The exponent is actually seven bits, as the high order bit is the sign bit of the mantissa (so that DDDDDD is a magnitude, not a two's complement number), and the value of the exponent is expressed in what is known as 'excess 40' notation. Rather than using a second bit to represent the sign of the exponent, X'40' (64 decimal) is added to all exponents. An exponent of 0 thus appears as 40, -1 as 3F, etc. This strategem allows exponents to range from -40 to +3F. Arithmetic on signed exponents is thus made easier -- to find the resulting exponent from a multiplication, for example, requires only that the exponents be added and 40 subtracted from the result.

Typical floating point representations would be:

A3F1 x 16^{1} = .A3F1 x 16^{6} = 46A3F100
A3F1 x 16^{11} = .A3F1 x 16^{15} = 4FA3F100
-A3F1 x 16^{1} = -.A3F1 x 16^{6} = C6A3F100
-A3F1 x 16^{-5} = -.A3F1 x 16^{-1} = BFA3F100
0 x 16 = .0000 x 16 = 00000000

To encode hex into floating point, set the hexadecimal point to the left of the six digit mantissa (fractional part), remembering to adjust the exponent by adding X'40' to it and setting the sign appropriately.

Similarly, to decode floating point into hex, use the sign bit as the sign of the result, subtract X'40' from the rest of the exponent, and adjust the exponent for moving the hexadecimal point to the right of the mantissa.

B. Instruction Cycle: Fetch - Increment - Execute

The execution of machine instructions takes place in cycles of three phases. In the Fetch phase, the instruction whose address is in the Program Counter is fetched from memory and copied into the Instruction Register (figure A). The Program Counter is then incremented by one (the Increment phase) so that it points to the instruction in memory directly after the one now in the Instruction Register (figure B). In the Execute phase, the instruction in the Instruction Register is decoded and executed.

For example, suppose that the instruction is a division. The contents of the register pair at the address given in the first operand are copied into the first two locations of the Arithmetic Logic Unit (ALU) (figure C). Next the word at the address of the second operand is selected from memory and, similarly, its contents are copied into the fourth location in the ALU (figure D). Then, the division operation is performed in the ALU, and the result is stored back into the designated register pair (figure E).

The cycle begins again by fetching the new instruction whose address is given in the Program Counter, which had been incremented in the second phase of the previous cycle.

Branches, both conditional (provided that the condition is true) and unconditional, cause the Program Counter to be reloaded during the Execute phase.

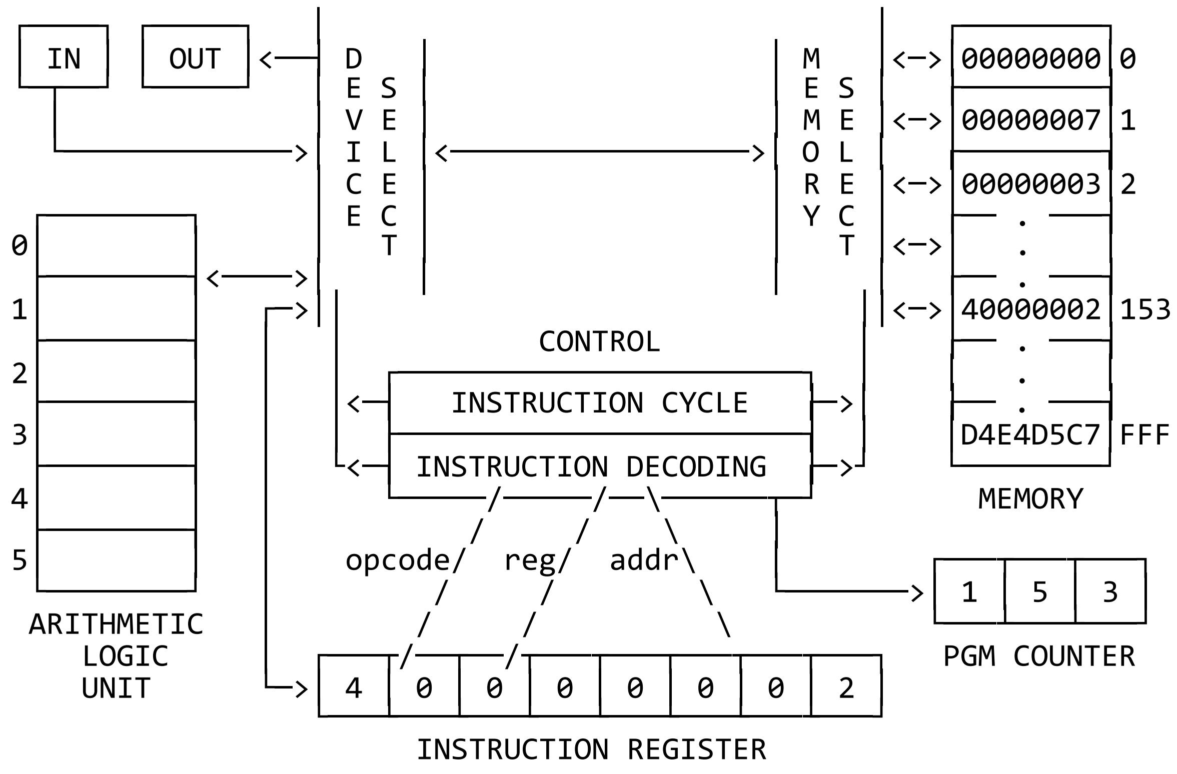


FIG A. FETCH

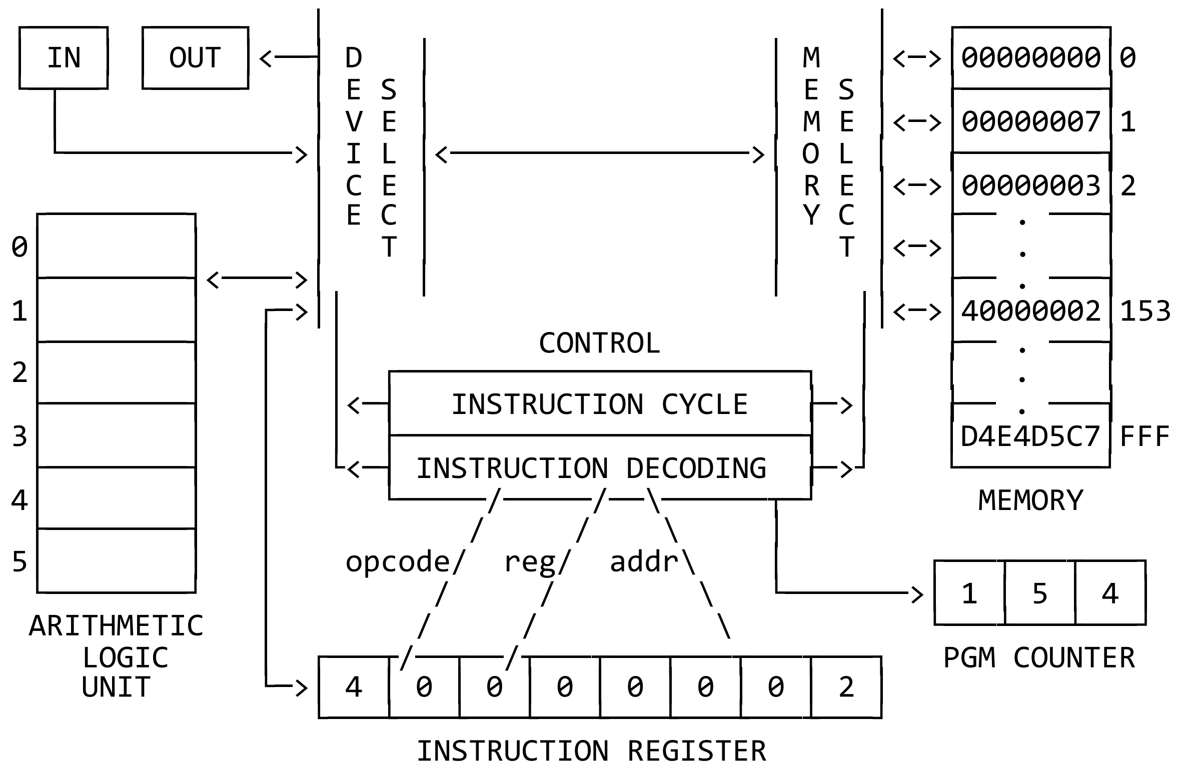


FIG B. INCREMENT

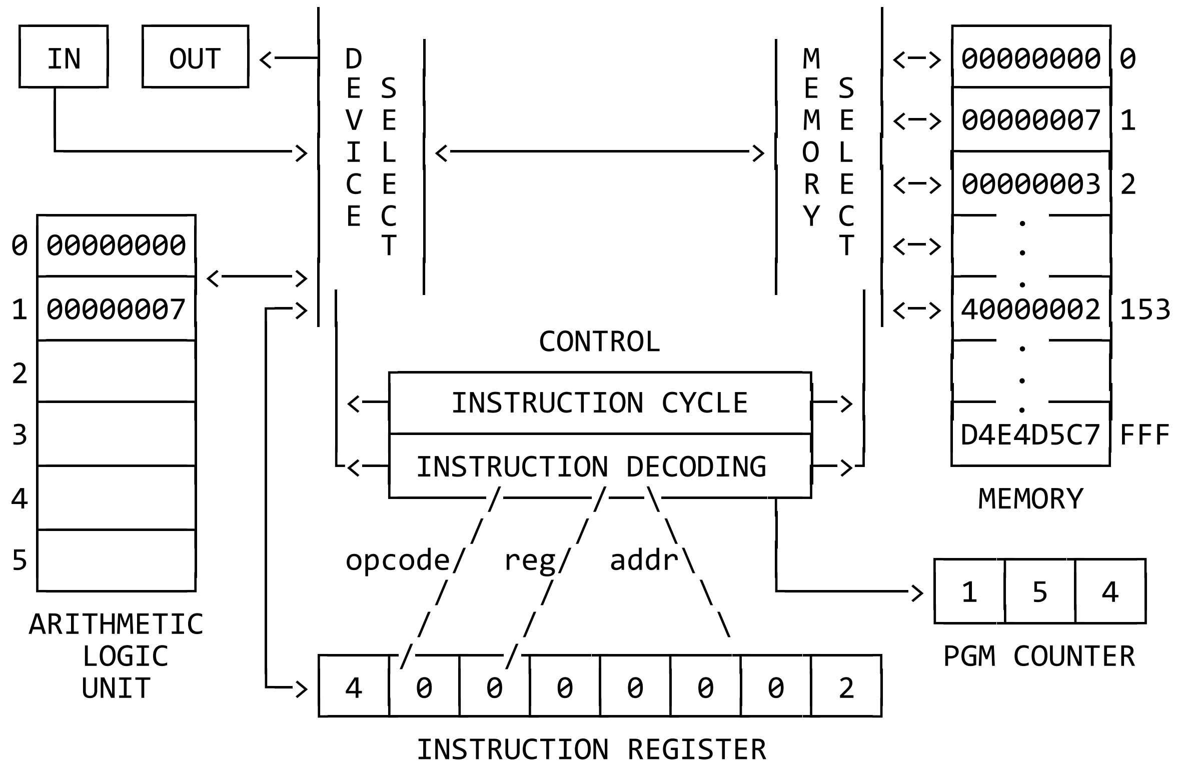


FIG C. EXECUTE (load first operand)

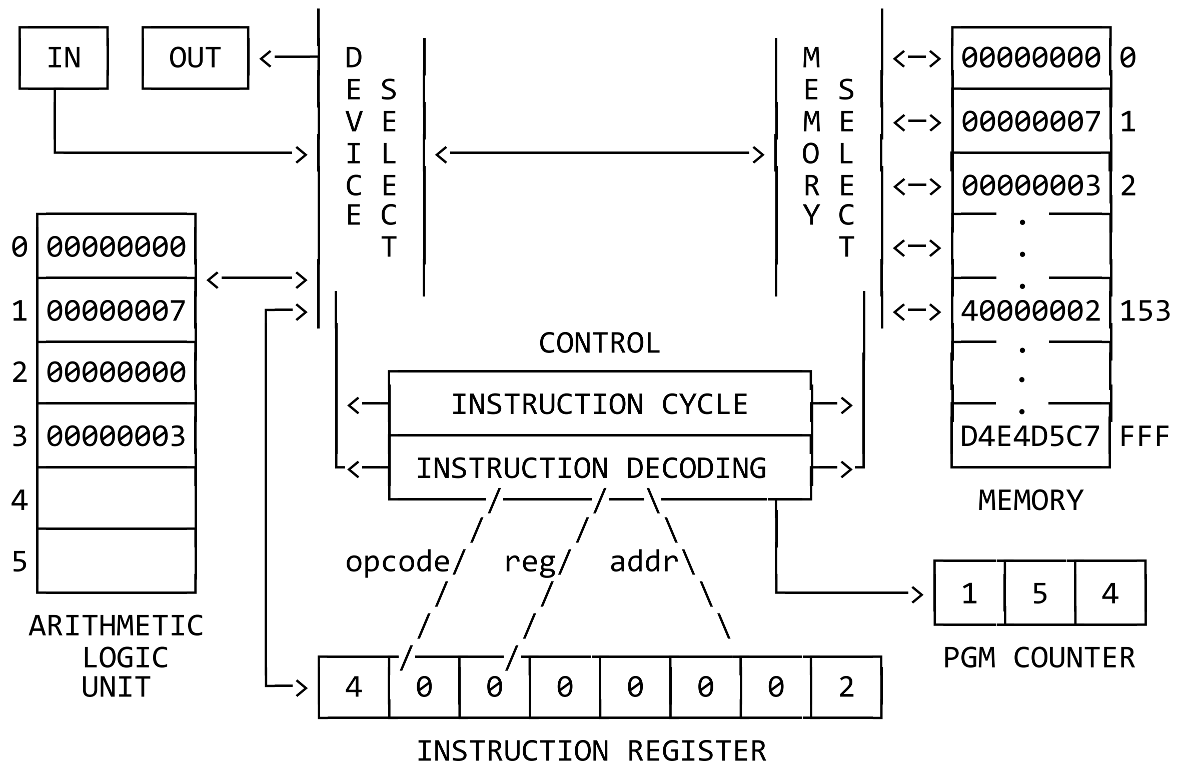


FIG D. EXECUTE (load second operand)

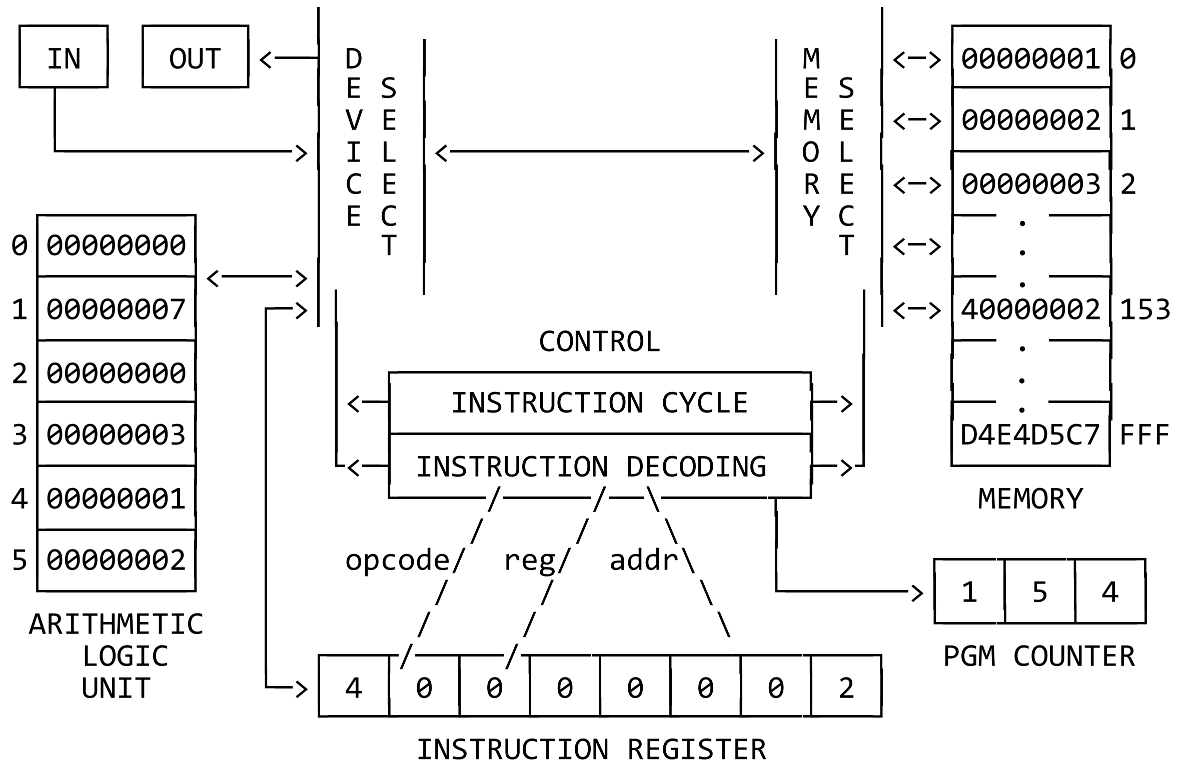


FIG E. EXECUTE (compute result and store)

C. Statement Formats

OPCODE	MNEMONIC	OPERAND	DESCRIPTION
10	A	R,>LOC(X)	Add word at EA to R
A2	AXAI	R,IMMD(X)	(R) + (X) + IMMD -> R
90	B	>LOC(X)	Branch to EA
98	BAL	R,>LOC(X)	Load PC into R and branch to EA
A0	BCT	R,>LOC(X)	Decrement R by 1, if not zero \\branch to EA
92	BE	R,>LOC(X)	If R is zero, branch to EA
91	BH	R,>LOC(X)	If R is positive, branch to EA
93	BL	R,>LOC(X)	If R is negative, branch to EA
96	BM	R,>LOC(X)	If logical CC for R mixed, branch to EA
95	BO	R,>LOC(X)	If logical CC for R ones, branch to EA
94	BOF	R,>LOC(X)	If overflow CC for R set, branch to EA
99	BR	R	Branch to address contained in R
97	BZ	R,>LOC(X)	If logical CC for R zeroes, branch to EA
40	D	R,>LOC(X)	Divide R, R+1 by word at EA
00	H	>LOC(X)	Halt execution
50	L	R,>LOC(X)	Load word at EA into R
30	M	R,>LOC(X)	Multiply R+1 by word at EA
B0	N	R,>LOC(X)	AND R with word at EA
B1	O	R,>LOC(X)	OR R with word at EA
20	S	R,>LOC(X)	Subtract word at EA from R
81	SLA	R,>LOC(X)	Shift R left by EA bits
83	SLDA	R,>LOC(X)	Shift R and R+1 left by EA bits
82	SLDL	R,>LOC(X)	Shift R and R+1 left by EA bits
80	SLL	R,>LOC(X)	Shift R left EA bits
71	SRA	R,>LOC(X)	Shift R right EA bits, propagating \\the sign bit
73	SRDA	R,>LOC(X)	Shift R, R+1 right by EA bits, \\propagating the sign bit
72	SRDL	R,>LOC(X)	Shift R, R+1 right by EA bits
70	SRL	R,>LOC(X)	Shift R right EA bits
60	ST	R,>LOC(X)	Store R at EA
C0	SVC	0,>LOC(X)	Execute CCW chain at EA
A1	SXAI	R,IMMD(X)	(R) - (X) + IMMD -> R
B3	TM	R,>LOC(X)	Set logical CC for R using mask at EA
B2	X	R,>LOC(X)	XOR R with word at EA

CC	=	condition code
EA	=	effective address
IMMD	=	immediate data
LOC	=	a memory location
PC	=	program counter
R	=	register specification
X	=	optional index reg (if omitted, parentheses also omitted)
>	=	optional, indicates indirecting is used

D. EBCDIC Character Codes

CHARACTER	EBCDIC	CHARACTER	EBCDIC
blank	40	F	C6
¢	4A	G	C7
.	4B	H	C8
<	4C	I	C9
(	4D	J	D1
+	4E	K	D2
|	4F	L	D3
&	50	M	D4
!	5A	N	D5
$	5B	O	D6
*	5C	P	D7
)	5D	Q	D8
;	5E	R	D9
¬	5F	S	E2
-	60	T	E3
/	61	U	E4
,	6B	V	E5
%	6C	W	E6
_	6D	X	E7
>	6E	Y	E8
?	6F	Z	E9
:	7A	0	F0
#	7B	1	F1
@	7C	2	F2
'	7D	3	F3
=	7E	4	F4
"	7F	5	F5
A	C1	6	F6
B	C2	7	F7
C	C3	8	F8
D	C4	9	F9
E	C5		

E. Error Messages

The SOS System provides a comprehensive set of error messages, designed to inform the user of any errors in a clear and concise manner.

Error messages are of the form SOSmnnn, where "m" indicates the source of the error message, and "nnn" is the error code. The processor codes are as follows:

0: Generated by the system Supervisor if an error which does not preclude the running of the job occurs, e.g. invalid /SOS card option; or generated by the machine code processor.
1: Generated by the assembler.
2: Generated by the machine language interpreter.
3: Generated by the macro generator.
4: Generated when an error severe enough to abend the current job, but not severe enough to preclude running the rest of the jobs in the batch, is detected.

0001 ERROR LIMIT EXCEEDED

More errors occurred during assembly or machine code loading than were specified in the ERROR parameter of the /SOS card (or SOSDFLT if not specified). Execution is not attempted, and the next job is fetched and processed.

0002 UNABLE TO OPEN SYSTEM MACRO LIBRARY

The SOS system macro library (SOSLIB) is not in correct macro library format (MACLIB for CMS, PDS for OS). This may also mean that the directory has been damaged. This error will not occur if the macro library cannot be found, in which case no error message would be generated. The SOS system macros will not be made available. Contact the appropriate systems personnel.

0003 WARNING: NO /END CARD, ONE HAS BEEN ASSUMED

Self explanatory.

0004 WARNING: TEXT FOUND BETWEEN END AND /DATA CARDS

The text is ignored.

0005 WARNING: INVALID TEXT BEFORE /SOS CARD

The statement following a /END card was not a /SOS card or the SYSIN input data set was empty.

0006 FATAL ERROR WHILE READING FROM SYSIN

This is a supervisor error which should never occur. Contact the appropriate systems personnel.

0007 INVALID PARAMETER; IGNORED

An illegal parameter was found in the preceding line of job control.

0008 UNEXPECTED END OF FILE ENCOUNTERED

End of file was detected during JCL processing. The run is terminated.

0009 JOB CONTROL PARAMETER OUT OF RANGE; SOSMAX USED

A decimal value for a job control parameter on the preceding line is greater than the maximum allowed. The maximum value is assumed.

0010 INVALID NUMERIC ARGUMENT; IGNORED

A job card parameter which takes a numeric argument has an illegal character in the number, or the number is longer than three digits. The parameter is ignored.

0011 MISSING FFF OR FF0 OPERAND

The FFF or FF0 pseudo-op lacks the required operand, which must be a valid SOS address. If FFF, execution begins at the default address of X'10'. If FF0, it is ignored.

0012 FFF OR FF0 OPERAND TOO LONG

The specified operand is longer than three hexadecimal digits. If FF0, it is ignored. If FFF, execution begins at the default address of X'10'.

0013 INVALID CHARACTER IN FFF OR FF0 OPERAND

The specified operand contains a non-hexadecimal digit. If FF0, it is ignored. If FFF, execution begins at the default address of X'10'.

0014 INSTRUCTION CONTAINS ILLEGAL CHARACTER

The instruction is not an eight digit hexadecimal number. An abend-forcing instruction is loaded.

0015 ADDRESS LIMIT EXCEEDED

The location counter exceeded the SOS address limit of X'FFF'. All subsequent instructions, until the next pseudo-op, are loaded at location X'FFF'.

0016 WARNING: MISSING FFF CARD

The machine language program does not end with an FFF pseudo-op. Execution begins at the default address of X'10'.

0017 COLUMN 9 CONTAINS A NON-BLANK CHARACTER

The eight digit instruction is not followed by a blank. An abend-forcing instruction is loaded.

0018 LINE COUNT MAY NOT BE ZERO; IGNORED

A value of zero was specified as the argument to the LINCT parameter on the /SOS card. This value is invalid. The parameter is ignored.

1001 DUPLICATE LABEL

The label used in the current EQU pseudo-op has been used in the label field of a previous instruction. The label and pseudo-op are ignored.

1002 INVALID OPCODE

The character string located in the opcode field is not a valid SOS assembly language instruction, pseudo-op, system macro, or programmer-defined macro. An abend-forcing statement is generated.

1003 INVALID DELIMITER

In general, a character that was expected was not found. For example, a blank or a "(" must appear after the address subfield of an instruction. An abend-forcing statement is generated.

1004 MACRO STATEMENT NOT ALLOWED

All macro definitions must appear at the top of the program. Only comments, SPACE, TITLE, EJECT, and PRINT pseudo-ops may appear before or between macro definitions. The statement is ignored.

1005 INVALID ADDRESS

The address in the operand field of the statement does not lie within the boundaries of SOS memory. If an ORG pseudo-op, it is ignored. If an END pseudo-op, execution begins at the default address of X'10'. Otherwise, an abend-forcing instruction is generated.

1006 MISSING OPERAND

An ORG or DS pseudo-op lacks the necessary operand. The statement is ignored.

1007 NEGATIVE OPERAND IN DS STATEMENT

The operand of a DS statement must be non-negative. The statement is ignored.

1008 ADDRESSING ERROR

The operand of a DS pseudo-op, when added to the current location counter, caused the boundaries of SOS memory to be exceeded. The pseudo-op is ignored.

1009 MISSING OR INVALID LABEL

The label field of an EQU pseudo-op did not contain a valid label. The pseudo-op is ignored.

1010 MISSING OPENING APOSTROPHE

The operand field of the TITLE pseudo-op did not begin with an apostrophe. It is ignored.

1011 MISSING CLOSING APOSTROPHE

The operand field of a TITLE pseudo-op or a DC pseudo-op with a C-type constant did not end with an apostrophe. If a TITLE pseudo-op, it is ignored. If a DC statement, an abend-forcing instruction is generated.

1012 MISSING OPCODE

There is no opcode field in this statement. An abend-forcing instruction is generated.

1013 PRINT OPTION MISSING OR INVALID

The operand field of a PRINT pseudo-op does not contain one of "ON ", "OFF ", "GEN ", or "NOGEN ". The statement is ignored.

1014 SPACE OPERAND INVALID OR TOO LONG

The operand field of a SPACE pseudo-op is not a one or two digit unsigned decimal number. The statement is ignored.

1015 INVALID REGISTER FIELD

The value of the register field of the instruction was not within the range zero to fifteen. An abend-forcing instruction is generated.

1016 MISSING REGISTER FIELD

The register field of the instruction was not specified. An abend-forcing instruction is generated.

1017 MISSING COMMA

A comma was not found where expected, e.g., after the register subfield of an instruction which requires one. An abend-forcing statement is generated.

1018 MISSING CLOSING PARENTHESIS

The close parenthesis after the index register specification is missing. An abend-forcing statement is generated.

1019 MISSING BYTE FIELD

One of the first three fields of a CCW is missing. A CCW which will do nothing is generated.

1020 INVALID BYTE FIELD

One of the first three fields of a CCW does not lie within the range zero to 255. A CCW which will do nothing is generated.

1021 INVALID IMMEDIATE DATA FIELD

The specified immediate data field cannot fit into four hexadecimal digits. An abend-forcing statement is generated.

1022 ADDRESS SPECIFICATION MISSING

The address field of an instruction is missing. An abend-forcing statement is generated.

1023 INVALID USE OF INDEXING

Indexing may not be specified in the operand field of an END pseudo-op. Execution begins at the default address of X'10'.

1024 UNDEFINED SYMBOL

An address constant specified in the operand field did not appear in the label field of an instruction or EQU pseudo-op. An abend-forcing statement is generated.

1025 MISSING SYMBOL

A symbol did not follow an operator ("+" or "-") in an expression in the operand field of a statement. An abend-forcing statement is generated.

1026 INVALID CHARACTER

A symbol in the operand field contained a non-alphanumeric character. An abend-forcing statement is generated.

1027 SYMBOL TOO LONG

A symbol in the operand field contained more than eight characters. An abend-forcing statement is generated.

1028 A-TYPE CONSTANT TOO LONG

An A-type constant contained more than four characters. An abend-forcing statement is generated.

1029 MISSING NUMBER

There was no digit between the apostrophes of a hexadecimal constant indicator (X''). An abend-forcing statement is generated.

1030 INVALID DIGIT

An invalid digit appeared in a decimal or hexadecimal constant. An abend-forcing statement is generated.

1031 NUMERIC SPECIFICATION TOO LONG OR VALUE TOO LARGE

A decimal constant was too large or greater than ten digits, or a hexadecimal constant was longer than eight digits. An abend-forcing statement is generated.

1032 UNABLE TO ALLOCATE LITERAL

The literal pool is full. There is no other location at the end of SOS memory in which to place another literal. An abend-forcing statement is generated.

1033 WARNING: MISSING END CARD

There was no END pseudo-op at the end of the program. Execution begins at the default address of X'10'.

1034 WARNING: LABEL IGNORED

A label appeared in the label field of a pseudo-op in which it is invalid (e.g., ORG).

1035 WARNING: EFFECTIVE ADDRESS EXCEEDS CORE BOUNDARY

The location counter exceeded the maximum SOS memory address of X'FFF'. All succeeding statements, until an ORG pseudo-op, will be loaded at location X'FFF'.

1036 WARNING: DUPLICATE LABEL

The label of this instruction has been used in the label field of a preceding instruction or EQU pseudo-op. The label on this instruction is ignored.

1037 WARNING: LABEL TOO LONG

The label is greater than eight characters in length and is ignored.

1038 WARNING: INVALID CHARACTER IN LABEL

The label contains a non-alphanumeric character or begins with a numeric. It is ignored.

2002 INVALID REGISTER SPECIFICATION

A double register operation was specified with register 15 as the indicated register. Execution halts and a dump is produced.

2003 ZERO DIVIDE EXCEPTION

The divisor in the divide operation is zero. Execution halts and a dump is produced.

2004 INSTRUCTION COUNT EXCEEDED

The number of instructions executed by the program has exceeded the number specified on the /SOS card (or the value of SOSDFLT if it was not so specified). Execution halts and a dump is produced.

2005 WARNING: INVALID SVC

An SVC has been encountered with a non-zero register specification.

2006 CCW OVERLAYS CORE BOUNDARY

Part or all of the CCW is outside the limits of SOS core. Execution halts and a dump is produced.

2008 ADDRESSING EXCEPTION

The calculated effective address lies outside the limits of SOS core. Execution halts and a dump is produced.

2009 INDIRECTING LIMIT EXCEEDED

The level of indirecting exceeded five. Execution halts and a dump is produced.

2012 ATTEMPT TO READ PAST END OF FILE

An attempt was made to do a "GET" after end of file had occured. Execution halts and a dump is produced.

2013 PRINT COUNT EXCEEDED

The number of lines printed during execution exceeded the maximum number of lines specified on the /SOS card (or SOSDFLT if not specified). Execution halts and a dump is produced.

2014 INVALID OPERATION CODE

An invalid operation code has been encountered during execution. Execution halts and a dump is produced.

2015 WARNING: ARITHMETIC OVERFLOW

The current instruction generated a number which could not fit into a 32-bit integer. The overflow condition code is set, and the contents of the register are undefined.

2016 WARNING: DIVIDE EXCEPTION

The quotient generated in the divide operation is too large to fit into a 32-bit integer. The register contents remain unchanged, and the overflow condition code is set.

2017 WARNING: TRACE COUNT EXCEEDED

More instructions have been traced than the number specified on the /SOS card (or SOSDFLT if not specified). The trace is forced off, but execution continues.

2018 WARNING: ABOVE LINE WENT PAST RIGHT MARGIN

A PUTX, PUTC, or PUTD attempted to write characters past column 120 of the above line. The extra characters are lost, and a RET of the line has been forced.

3001 LABEL INVALID OR TOO LONG

The label parameter on a macro prototype statement does not begin with an ampersand, consists of characters other than alphanumerics, and/or is longer than eight characters. The definition is ignored.

3002 INVALID CONTINUATION CARD

A continuation card has a label field or is all blank. The definition or invocation, as appropriate, is ignored.

3003 SYMBOLIC PARAMETER INVALID OR TOO LONG

A symbolic parameter in the operand field of a macro prototype statement contains characters other than alphanumerics or is longer than eight characters. The definition is ignored.

3004 MACRO NAME MISSING, INVALID, OR TOO LONG

The opcode field of a macro prototype statement is either missing, contains characters other than alphanumerics, begins with a numeric, and/or is longer than eight characters. The definition is ignored.

3005 ILLEGAL TO USE &SYSNDX AS SYMBOLIC PARAMETER NAME

&SYSNDX is a reserved system keyword and may not be specified in a macro prototype statement. The definition is ignored.

3006 GIVEN MACRO NAME DOES NOT MATCH DIRECTORY ENTRY

The MACLIB or PDS directory name for the macro is not the same as the macro name found on the prototype statement. This error can only occur for SOS library macros, and should be brought to the attention of appropriate systems personnel. The definition is ignored.

3007 MISSING CLOSING APOSTROPHE

The default value of a keyword parameter in the macro prototype or one of the parameter values in a macro invocation contained an odd number of apostrophes. The definition or invocation, as appropriate, is ignored.

3008 DUPLICATE PARAMETER NAME

A macro prototype contained two references to the same symbolic parameter. The definition is ignored.

3009 AMPERSAND NOT FOUND WHEN EXPECTED

In scanning a macro prototype, an ampersand was expected but not found (i.e., following a comma or as the first non-blank on a continuation card). The definition is ignored.

3010 BADLY FORMED MACRO STATEMENT

A macro header statement may not contain a label, operand, or comment field. The definition is ignored. This error occurs only for SOS library macros. The appropriate systems personnel should be contacted.

3011 MISPLACED POSITIONAL PARAMETER

All positional parameters must precede all keyword parameters. The definition or invocation, as appropriate, is ignored.

3012 "END", "MEND", "MACRO", OR JCL FOUND IN MACRO DEFINITION

These statements may not be generated from within a macro definition. The definition is ignored.

3013 INVALID DELIMITER

This error indicates an invalid internal TRT table. The definition or invocation, as appropriate, is ignored. Contact the appropriate systems personnel.

3014 TOO MANY CONTINUATION CARDS

Only one continuation card is allowed per prototype or invocation statement. The definition or invocation, as appropriate, is ignored.

3015 MISSING MEND STATEMENT

EOF, JCL, or "END" or "MACRO" statements were encountered before the MEND statement. The definition is ignored.

3016 PREVIOUSLY DEFINED MACRO NAME

An attempt was made to define a macro whose name is the same as that of an assembler opcode or pseudo-op, or whose name is that of a previous user-defined macro (attempting to use the name of an SOSLIB macro as a user defined macro causes the SOSLIB macro to be overridden and ignored). The definition is ignored, and the previous definition remains valid.

3017 MISSING PROTOTYPE STATEMENT

A macro header statement was immediately followed by a macro trailer statement. They are ignored.

3018 TOO MANY POSITIONAL PARAMETERS

More positional parameters were specified in a macro call than were on the prototype. The statement is ignored.

3019 INVALID OR UNDEFINED KEYWORD NAME

An attempt was made in a macro invocation to assign a value to a keyword parameter which does not appear in the prototype for that macro. Alternatively, a keyword parameter name on a macro invocation statement contains characters other than alphanumerics. The statement is ignored.

3020 KEYWORD VALUE ASSIGNED MORE THAN ONCE

A macro call attempted to assign a value to the same keyword parameter twice. The statement is ignored.

3021 UNEXPECTED END OF FILE ENCOUNTERED

While attempting to read the continuation of a macro prototype statement, EOF occurred. The definition is ignored.

3022 WARNING: BADLY FORMED MACRO STATEMENT

A macro header statement contained a label, operand, or comment field. The statement is processed normally.

3023 WARNING: BADLY FORMED MEND STATEMENT

A macro trailer statement contained a label, operand, or comment field. The statement is processed normally.

3024 WARNING: SUBSTITUTION OVERFLOW; TRUNCATED AT COL 72

After performing symbolic parameter substitution on a macro model statement, the expanded statement exceeded 72 characters in length. All characters following the 72nd are lost, but otherwise the statement is processed normally.

3025 INVALID SYMBOLIC PARAMETER

While attempting to do symbolic parameter substitution in a macro model statement, a symbolic parameter was encountered which was either of length zero (i.e., a lone ampersand) or was longer than eight characters. An abend-forcing statement is generated.

3026 UNDEFINED SYMBOLIC PARAMETER

In attempting to do symbolic parameter substitution in a macro model statement, a syntactically valid parameter was encountered for which there was no definition (i.e., the parameter did not appear in the prototype for this macro and was not &SYSNDX). An abend-forcing statement is generated.

3027 ERROR IN DEFINITION OF MACRO

An attempt was made to invoke a macro whose definition was in error. The invocation is ignored.

3028 MAXIMUM NEST LEVEL EXCEEDED

An attempt was made to invoke a macro which was nested deeper than the maximum nesting level (six). The invocation is ignored.

3029 STATEMENT MAY NOT BE MACRO GENERATED

An attempt was made to generate a MACRO, MEND, END, or JCL statement. These statements must be read from the input stream and may not be generated from a macro definition. An abend-forcing statement is generated.

4001 NO MORE FREE STORAGE

The free storage area used for macro processing, assembly, and the listing cross reference has been exhausted.

4002 PERMANENT I/O ERROR IN PASS2

This indicates a severe internal conflict. Contact the appropriate systems personnel.

4003 MISSING MEND STATEMENT FOR USER-DEFINED MACRO

A user macro definition did not have a MEND statement preceding JCL or the END statement. Probable user error. Correct and re-run.

4004 NOT ENOUGH FREE STORAGE FOR USER MACRO DEFINITION

In trying to save a user macro definition, free storage was exhausted.

F. Abend Messages and Codes

The SOS system must have a predefined environment, including certain files and free storage facilities, available in order to process its tasks. If it is determined that these conditions do not exist, SOS abends its task, returning to the system.

Certain error returns will also force the system to abend, that is, if it cannot recover from a system error it will terminate rather than continuing in what might be a fruitless endeavor.

Under CMS, the message text is printed at the terminal or in the CMSBATCH listing, with the message number being returned as an error code. Under OS, the message number is the user ABEND code, and the message text is not printed.

1 (OS: ERROR OPENING SYSIN) (CMS: FILE NOT FOUND)

Under OS, the DCBOFLGS bit indicating that a file is open was off after the OPEN macro was issued. Usually this means that the DD card for SYSIN was missing. Under CMS, the file of the specified name and of type SOS was not found on any logged in disk.

2 (OS: ERROR OPENING SYSPRINT)

The DCBOFLGS indicating that a file is open was off after the OPEN macro was issued. Usually this means that the DD card for SYSPRINT was missing.

3 (OS: NOT ENOUGH FREE STORAGE) (CMS: ERROR LINKING TO SEGMENT)

Under OS, a conditional FREE asking for at least 55K of storage returned indicating that the space was not available. Increasing the region size should eradicate this problem. Under CMS, an attempt was made to link a temporary segment at device 3 for use as scratch storage, but the attempt failed. This message is usually preceded by an explanatory message from the CMS SEGMENT command.

4 (OS: SYNAD ERROR EXIT TAKEN)

An unrecoverable I/O error has occured. This message is accompanied by a message on the system log which contains the SYNAD information from the file involved. See the appropriate systems personnel.

5 (OS: ERROR OPENING SYSUT2)

The DCBOFLGS for SYSUT2 indicating that the file is open was off after the OPEN macro was issued. This usually indicates a missing DD statement for SYSUT2.

6 FATAL I/O ERROR IN ERROR HANDLER

While attempting to print an error message, the error handler routine received a non-zero return code from the I/O routine. This indicates internal system inconsistencies and should be brought to the attention of the appropriate systems personnel.

7 (OS: ERROR OPENING SYSUT1)

The DCBOFLGS for SYSUT1 indicating that the file is open was off after the OPEN macro was issued. This usually is caused by a missing DD statement for SYSUT1.

8 (CMS: SPECIFY FILE NAME)

A file name was not specified with the SOS command.

9 UNEXPECTED EOF DURING JCL PROCESSING

The JCL analyzer, in attempting to read a card to see whether it is continuation, received an end-of-file indication from the I/O routine. Probable user error.

10 FATAL I/O ERROR DURING JCL PROCESSING

The JCL analyzer, in attempting to read a card in to see whether it is continuation, received a non-zero and non-EOF return code from the I/O routine. Contact the appropriate systems personnel.

11 (CMS: PRINTR ERROR n IN IOMATIC)

In attempting to write a line directly to the printer, the I/O routine received a non-zero return code (namely n). If you have not done something stupid like detaching your printer, see the appropriate systems personnel.

12 SYSIN LRECL MUST BE 80

The SYSIN data set (or SOS file) has logical records which are not of length 80.

13 SYSIN MUST HAVE FIXED RECORDS

The SYSIN data set (or SOS file) must have records which are of fixed length (variable or undefined are no-no's).

14 NOT ENOUGH CORE FOR SYSTEM MACRO DEFINITIONS

There was not enough free storage to read the SOS system macro definitions (GETC, PDUMP, etc.) into core. Under OS, increase the region size. Under CMS, see the appropriate systems personnel. In either case, if macros are not being used, specifying NOMACRO on the /SOS card will keep them from being read in.

15 (OS: SYSIN BLOCKSIZE MUST BE A MULTIPLE OF 80)

SYSIN must have a blocksize such that BLOCKSIZE = BLOCKING_FACTOR * 80, where BLOCKING_FACTOR is an integer.

16 NOT ENOUGH FREE CORE FOR BUFFER ALLOCATION

In attempting to allocate internal I/O buffers, free storage was exhausted. Under OS, increase the region size. Under CMS, see the appropriate systems personnel.

17 NOT ENOUGH ROOM FOR MACRO CONTROL AREA

There was not enough free storage to allocate the control blocks necessary for handling a macro definition. Under OS, use a larger region size.

19 EOF WHILE LOADING USER MACRO DEFINITION

While reading in a user defined macro, EOF was encountered before the MEND statement. Probable user error.

20 (CMS: INVALID FILE MODE)

This indicates a severe error in either SOS or internal CMS file management, and should be brought to the <u>immediate</u> attention of the appropriate systems personnel.

21 (CMS: NO R/W DISK LOGGED IN)

For a job requiring disk I/O (SYSPRINT to a listing file and/or an assembler task), no writeable disk was logged into the machine.

9 GLOSSARY

The following is a list of oft-used words in computer science. Most descriptions are fairly general, while those things in square brackets apply to SOS in particular.

ABEND: Abnormal end or termination of a job (due to an illegal instruction, exceeding a system limit, etc.).

ACCUMULATOR: see GENERAL PURPOSE REGISTER.

ADDRESS: The numerical location of a word in memory.

ADDRESS CONSTANT: A constant [8 hex digits] which can be interpreted as the address of, or a pointer to, a piece of information.

ADDRESSING EXCEPTION: The result of an attempt to reference (at execution time) a memory location which is greater than the size of memory, or which is negative [Execution abends if this occurs.].

ALGORITHM: An unambiguous, step by step, problem solving procedure which leads to an answer in a finite amount of time. Long division and alphabetizing a list of names are examples of algorithms. Contrast to HEURISTIC.

AND: To combine bits under the rule:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

ARITHMETIC LOGIC UNIT (ALU): That part of a computer's circuitry concerned with performing arithmetic and logical operations.

ASCII: American Standard Code for Information Interchange. A standard method among hardware manufacturers for representing characters with a six, seven, or eight bit code.

ASSEMBLER: A program which translates symbolic program statements written in an assembly language into machine language, generally with a one-to-one correspondence between symbolic statements and machine language statements.

ASSEMBLY LANGUAGE: A computer language in which program statements are specified symbolically in terms of operation codes, register specifications, and addresses.

BINARY NUMBERS: Numbers in base (radix) two.

BIT: A BInary digiT; the most basic unit of computer storage. A bit has two states, either "on" or "off". To a programmer this is usually interpreted as one (on) or zero (off), or alternatively true (on) and false (off).

BOMB: To abend in a reasonably spectacular or humorous manner.

BRANCH: To cause execution to continue at a location other than the next sequential one; the instruction that causes such a branch.

BUFFER: A work area in which data is accumulated prior to some use (e.g., building up a line to be printed).

BUG: A syntactic or logical error in a program.

BYTE: A grouping of n consecutive bits, occurring at a multiple of n bits from the beginning of memory [n is eight].

CALL: The invocation of a subroutine or other block of code, with provisions made for a return from said block of code to the caller [see the BAL instruction].

CALLING SEQUENCE: The sequence of instructions and data by which a subroutine is called. It usually includes providing the subroutine with a return address and a parameter list.

CCW: Channel Command Word, an instruction to be performed by the CHANNEL.

CHANNEL: An electronic device which aids the CPU in performing Input/Output operations.

CLOBBER: To place incorrect data into a core location or register, destroying the necessary information contained therein.

CODE: To write program statements from a design; the actual source statements.

COMPILER: A translation program which generally produces more than one statement of machine code for each statement of the compiler language.

CONCATENATION: Juxtaposition of character (or bit) strings.

CONDITION CODE: Flags internal to a computer which are set as the result of arithmetic or logical operations. These flags can be tested by a program to determine whether certain conditions have occurred during execution [a logical and overflow condition code are provided for each register.].

CONTROL COUNTER: see PROGRAM COUNTER.

CORE: The individual ferrite rings from which most computer memories today are constructed. These rings are capable of magnetically "remembering" data by representing zero and one as two different directions (clockwise or counterclockwise) or magnetization. Also a synonym for MEMORY.

CPU: Central Processing Unit. That part of the computer which contains the circuitry to decode and execute machine instructions, and to control operations performed by other parts of the computer.

DEBUG: To detect and remove bugs (errors) from a program, or malfunctions from a machine.

DEFAULT: An assumed value used whenever a parameter is not explicitly specified.

DISK, DISK PACK: A (group of) thin plastic or ceramic plate(s) coated with a magnetizable substance (ferric oxide) capable of magnetically "remembering" data.

DIVIDE EXCEPTION: A condition which occurs when a division operation results in a number which is too large to be contained within the finite number of bits the machine has allowed for the result [execution does not halt due to a divide exception; the overflow condition code is set; the instruction is ignored.].

DRUM: A cylindrical metallic device which is coated with a magnetizable substance and is capable of remembering data; it is similar to but generally faster than a disk (see above).

DUMP: A printout of a core area in a machine-dependent format. Also, the act of producing such a printout.

EBCDIC: Extended Binary-Coded-Decimal Interchange Code. An IBM standard method for representing characters using an eight bit code.

EFFECTIVE ADDRESS CALCULATION: The calculation (by the CPU and/or ALU) of the actual memory location referred to in a machine instruction.

EXCLUSIVE OR: To combine bits according to the rule:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

EXECUTION PHASE: The step of the instruction cycle during which the instruction to be performed is parsed, the register(s) and core location(s) involved are calculated, and the operation is performed.

FETCH PHASE: The first step of the instruction cycle, during which the next instruction to be executed is copied from core into the instruction register.

FIELD: A sequence of contiguous positions used to represent a logical unit of information; for instance, the first eight columns of an SOS statement can contain a label, or the last five hex digits of an SOS machine instruction form the address field.

FIXED FORMAT: The fields of a statement must occur at specific positions; data must be placed in certain columns for proper execution.

FIXED POINT: The manner in which integers are represented in digital computers, usually by a bit pattern of fixed length. See FLOATING POINT.

FLAG: A bit which is set and later tested so that a program can "remember" a condition which has occurred previously. Sometimes called a "switch."

FLOATING POINT: A computer based "scientific notation" capable of representing extremely large and small numbers in a finite sequence of bits. The floating point number contains both the fraction and the exponent (see FIXED POINT and Appendix A5). [SOS does not support floating point arithmetic].

FLOW CHART: A pictorial description of the logic of a program, consisting of standardized graphical symbols and connectives.

FREE FORMAT: The fields of a statement need not occur in specific positions, but are usually separated by a delimiter and occur in a specific order.

FULLWORD: See WORD.

GENERAL PURPOSE REGISTER: A unit in which arithmetic and logical operations may be performed. Ordinary memory locations can only be used to store data [the first sixteen words of memory are treated as general purpose registers]. When used to perform arithmetic operations, also called an accumulator.

<u>GLITCH</u>: A small bug which does not seriously affect the running of a program.

<u>HAND SIMULATION</u>: An important debugging technique in which the execution of a program is checked by keeping track, on paper, of the contents of relevant registers and storage locations, as the actions of various operations are simulated. Also called desk checking.

<u>HARDWARE</u>: The electronic circuitry and mechanical devices which make up a computer system (core, channels, CPU, ALU, I/O devices).

<u>HEURISTIC</u>: Pertaining to exploratory methods of problem solving in which solutions are discovered by evaluation of the progress made toward a final result. Contrast to ALGORITHM.

<u>HEXADECIMAL COMPLEMENT</u>: A method of representing negative numbers in base sixteen. Equivalent to TWO'S COMLEMENT.

<u>HEXADECIMAL NUMBER</u>: Numbers in base 16, represented by the characters 0, 1, ..., 9, A, B, C, D, E, F, which correspond to the decimal numbers zero through fifteen. A hexadecimal digit can be represented by four binary digits, hence one byte contains two hexadecimal digits. See X'number'.

<u>HIGH ORDER</u>: Most significant; the high order bit in a 32 bit number is bit zero (furthest to the left).

<u>IMMEDIATE DATA</u>: Data that is actually part of the instruction that uses it, as opposed to data which is located elsewhere in memory and is referenced via an effective address [immediate data is stored in 16-bit two's complement form].

<u>INCREMENT PHASE</u>: The step in the instruction cycle in which the program counter is incremented by one.

<u>INDEXING</u>: The use of the contents of a register (in addition to the location field) in calculating the effective address during the execution cycle. It is typically used for accessing successive memory locations in loops, or for array references.

<u>INDIRECT ADDRESSING</u>: The use, during effective address calculation, of the instruction effective address to address a word of memory which is then used to calculate the effective address [this can be continued for several levels, but there seems to be no practical use for going more than two levels deep].

<u>IN-LINE DATA</u>: Data which is placed within the program at the time it is written ("DC'ed" in), as opposed to being read from an I/O device.

<u>INPUT/OUTPUT</u>: The process of and the procedures for moving data into and out of the main memory of a computer (to and from a secondary storage device).

<u>INSTRUCTION</u>: A bit string which can be decoded by a computer into an executable command.

<u>INSTRUCTION COUNT</u>: The maximum number of instructions that a program is allowed to execute. Exceeding this count will cause the program to abend.

<u>INSTRUCTION COUNTER</u>: See PROGRAM COUNTER.

<u>INSTRUCTION CYCLE</u>: The three step "Fetch, Increment, Execute" process characteristic of computer operations.

<u>INSTRUCTION MODIFICATION</u>: A method of altering instructions during program execution. This method of controlling program execution, which was highly popular in the 1950's and 60's, is frowned upon today as making programs hard to understand.

<u>INSTRUCTION REGISTER</u>: The part of the CPU containing the instruction currently being decoded and executed.

<u>INTERFACE</u>: The parameter passing, calling sequences, and data sharing (sometimes referred to as "hand shaking") by which one section of a system interreacts with another, as in an assembler interfacing with a macro generator, or a printer with a channel. Also, the act of building such an interface.

<u>INTERPRETER</u>: A program that simulates, one at a time, the operations specified by given source program statements in order to calculate results. Assemblers or compilers, on the other hand, translate the source program, in its entirety, to machine code which is executed by the hardware at a later time. An interpreter can be viewed as a software implementation (simulation) of the "source language machine" (i.e., the execution of its instruction cycle) [the SOS machine interprets SOS machine language generated by the assembler].

<u>JCL</u>: Job Control Language (see section 6).

<u>JOB</u>: A sequence of tasks which perform a useful function, e.g. an assembly followed by a load of the assembled machine language followed by execution is a job consisting of three tasks.

<u>JOB STREAM</u>: The flow of jobs into a job processor through an input device (via cards, tape, remote job entries).

<u>KEYWORD OPERAND</u>: A parameter whose meaning is specified by an associated keyword and which is independent of positional relationships.

<u>KLUDGE</u>: The use of a feature of a program, processor, algorithm, or machine in an underhanded manner to produce a useful but hardly obvious result.

<u>LABEL</u>: A symbolic identifier for data, storage areas, or instruction statements within an assembler or higher level language. Other statements within a program can refer to these labels in their operand fields. At assembly time, these labels are translated into actual memory addresses.

<u>LEFT JUSTIFY</u>: To align a variable, parameter, or string in a field so that its left-most extent corresponds to the left-most boundary of the field.

<u>LITERAL</u>: An expression in the operand field of an assembler statement, preceded by an identifier which signals it as a literal ["="]. The assembler will assign the expression to a memory location, and will place the address of the memory location into the address field of the instruction containing the literal.

LOADER: A system program which places machine code programs into the proper areas of core, as specified by relocation statements [FF0 and FFF pseudo-ops].

LOCATION: The position within memory that denotes a word, register, or some other integral unit of storage.

LOCATION COUNTER: An assembly time counter used by the assembler to assign memory locations to source statements. It starts at some default memory location [X'10'], or is specified explicitly, and is incremented by an appropriate amount for each statement [one for instructions, more for DC'd character strings and CCW's]. ORG and DS pseudo-ops also change the value of the location counter. Its value may be used in assembler statements by using the location counter reference symbol [asterisk] in an expression [e.g., L R1,*+5].

LOGICAL DATA: Data considered as a string of bits, rather than as a signed number or a string of characters.

LOW ORDER: Least significant; the low order bit of a 32 bit word is bit 31 (furthest to the right).

MACHINE CODE: The bit patterns (instructions) that are actually decoded and executed by the computer.

MACHINE LANGUAGE: A computer language consisting of machine code instructions.

MACRO (DEFINITION): A section of code which is used as a "template" by the macro generator to produce other sections of code similar to it.

MACRO GENERATOR: A program which processes macro definitions and calls to produce assembly language statements which are similar to the definition, but which may have certain features changed by specifying them in the macro call. Also called a Macro Processor.

MASK: A pattern of bits used by a programmer to test the status of other bits (flags) set previously.

MEMORY: The main storage area of a computer. The terms memory, storage, core, memory locations, and combinations thereof are used to denote all or specific portions of a computer's memory.

MNEMONIC: Attempting to assist memory (people memory, not core memory) through a symbolic abbreviation or representation of a quantity [e.g. "A" is the mnemonic for the operation code X'10'; RESULT might be the programmer's mnemonic label on a memory location].

MODULE: A functionally defined program which can run independently of another program, but which may also be used in conjunction with other programs to produce a larger, more complex, more useful functional entity.

MUNG: Garbage. Basically, a register or a core location contains mung if it does not contain what the programmer expects it to contain, or if the programmer does not know what it contains.

NO-OP: NO OPeration. A statement which does nothing.

OBJECT DECK (CODE, LANGUAGE): A machine code program which has been generated by an assembler or compiler.

ONE ADDRESS COMPUTER: Each instruction in the machine language of this type of computer can reference only one main storage location. [SOS is actually a "one-and-a-half" address computer, since the register field can address part of core].

OPERATING SYSTEM: A fairly large and complex program which controls the allocation of a computer's resources to users of the system.

OPERATION (OP) CODE: The digits or mnemonics in an instruction which indicate the machine operation to be performed.

OR: To combine bits according to the rule:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

OVERFLOW: The generation, by an arithmetic operation, of a number which is too large to be represented by the number of bits the computer has allowed for number representation.

Fixed Point -- an arithmetic operation produced a number too large to fit into a single machine word. Technically an overflow is recognized by a carry into the sign bit without a carry out or by a carry out of the sign bit without a carry in.

Floating Point -- the exponent became too large to be represented.

PADDING: Adding blanks or zeroes (as appropriate), on the left or right, to an item of information in order fill up a field of specified length.

PARAMETER LIST: A grouping of integral units of storage, usually contiguous, containing information which is passed from one routine to another.

PASS: To communicate information, in the form of a parameter or parameter list, from a calling routine to a subroutine. This is done by placing the information into a location agreed upon by the caller and the subprogram (e.g., the IBM convention is to have R1 point to a list of addresses of actual items).

Also, pass denotes a phase of execution, as in a "two pass" assembler. Multiple passes implies going through all the source data several times, performing different functions each time. For example, the first pass of an assembler might assign storage locations and translate mnemonics, while the second pass might do the actual code generation.

POINTER: A piece of data which is the address of ("points to") another piece or list of data. Following successive pointers to access data is called POINTER CHASING. See ADDRESS CONSTANT.

POSITIONAL PARAMETER: A parameter whose meaning is indicated by its positional relation to other parameters.

PRINT COUNT: The maximum number of lines a program is allowed to print. Exceeding this count will abend the run.

PROCESSOR: A program which does language translation or interpretation; also a synonym for CPU.

PROGRAM COUNTER: A register internal to the CPU which contains the address of the next instruction to be executed.

PSEUDO-OPS (OPERATIONS): Opcodes which specify actions to be taken during assembly time, and are not generated as actual machine code. Pseudo-ops, for example, can indicate to the assembler how to format the listing [PRINT, TITLE, EJECT, SPACE], or to reserve storage [DC, DS].

RELATIVE ADDRESSING: The use of constants (or symbols equated to constants) to reference core locations in relation to (as an offset from) other core locations. [For example, if DATA refers to location X'031', then

```
         L   R3,DATA+8
```

would load into R3 the contents of memory location X'039'.] Relative addressing is distinguished from effective address calculation in that the former is done at assembly time, while the latter is done at execution time.

RETURN CODE: A value passed from a subroutine to its caller indicating the degree of successful completion.

RIGHT JUSTIFY: To align a variable, parameter, or string in a field so that its right-most extent coincides with the right-most boundary of the field.

ROUTINE: A well-defined section of code which performs a specific function.

SCHLEPP: One who carries out menial but essential tasks, such as getting lunch, watering plants, bursting output, FRESSing, and reproducing (that is, XEROXing).

SNAP: A selective dump performed at various times during execution [a la PDUMP].

SOFTWARE: In general, any program; sometimes refers to the set of programs which make a system usable by a programmer.

SOURCE DECK (CODE, LANGUAGE): A program written in symbolic statements, such as assembler.

STORAGE: That part of the computer concerned with holding machine instructions and data so that they may be accessed by the CPU during execution. See CORE, MEMORY.

STRUCTURED PROGRAMMING: A programming discipline in which program flow of control is restricted to one path in, one path out sequences of instructions formed from sequential code, conditionals, loops, and subroutine calls, in order to present the program clearly in an easy to understand manner.

SUBROUTINE: A section of code, often invoked from more than one place, which performs a predefined function. Subroutines are usually passed parameters and save all registers they use (i.e., they're nice to their caller).

SUPERVISOR: [The operating system for SOS, which controls job processing, and invokes the assembler or machine language processor as necessary. It initiates the interpreter and any I/O produced during execution.]

SYMBOL: A name which stands for a constant, a register, or a memory location. [A string of one to eight alphanumeric characters starting with an alphabetic character.]

TASK: The performance of a single, useful function. For example, assembling a source program.

TRACE: An instruction by instruction printout of data, core locations, and results involved in the execution of specific operations.

TRACE COUNT: [The maximum number of instructions an SOS program is allowed to trace. Exceeding the count will turn off the trace, but program execution will continue.]

TWO'S COMPLEMENT: A method of expressing positive and negative numbers in base two (see Appendix A.2).

UNDERFLOW: The generation of a floating point number with an exponent too small to be represented by the machine.

VIRTUAL: Appearing to be, but not really being. This term is used when a computer simulates a facility in such a way that the user believes that the facility really exists. Interpreters create "virtual computers," also called virtual machines.

WORD: An integral unit of computer storage or the contents of such a unit (in SOS and the IBM 360 and 370 series, one word is 32 bits, each bit typically referred to by numerical position, left to right, 0 to 31. Other popular word sizes are 36, 60, and 16 bits).

WRAPAROUND: The capability of a computer to treat core as if it were circular. For example, if core consisted of 512 words, a

```
         ST    R4,515
```

would store R4 into location 003 [wraparound is not supported; see ADDRESSING EXCEPTION].

XOR: See EXCLUSIVE OR

X'number': (IBM 360 and 370, and SOS notation). The number is treated as a hexadecimal (base sixteen) number. In common usage, when clear from context, the X' ' may be omitted. See HEXADECIMAL.

ZAP: To zero out, e.g., "zap" a register or memory location.

10 INDEX

TABLE OF CONTENTS

****
****
****
****
****

****
****
****

```
****************************************************************************************
** VM/370 VERSION 06 LEVEL es PLC Pack **************** CLASS A *** DEV 00E **********


           USERID    ORIGIN      MAINT       MAINT                               VV
                                                                                 VV
           DISTRIBUTION CODE     MAINT                                    3333333333
                                                                         33333333333
           SPOOL FILE NAME TYPE  SOSPLM      MEMO                        33      VV3
                                                                                  V3
           CREATION DATE         04/06/19 14:47:24                                 3
                                                                                3333
           SPOOL FILE ID         0860                                           3333
                                                                                   3
           RECORD COUNT          4680                                              3
                                                                         33        3
                                                                         33333333333
                                                                          3333333333

                                                                            VM/370 R



                                  MM        MM   AAAAAAAAAA    IIIIIIIIII   NN        NN
                                  MMM      MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN       NN
                                  MMMM    MMMM  AA        AA       II       NNNN      NN
                                  MM MM  MM MM  AA        AA       II       NN NN     NN
                                  MM  MMMM  MM  AA        AA       II       NN  NN    NN
                                  MM   MM   MM  AAAAAAAAAAAA       II       NN   NN   NN
                                  MM        MM  AAAAAAAAAAAA       II       NN    NN  NN
                                  MM        MM  AA        AA       II       NN     NN NN
                                  MM        MM  AA        AA       II       NN      NNNN
                                  MM        MM  AA        AA       II       NN       NNN
                                  MM        MM  AA        AA   IIIIIIIIII   NN        NN
                                  MM        MM  AA        AA   IIIIIIIIII   NN         N



                                  MM        MM   AAAAAAAAAA    IIIIIIIIII   NN        NN
                                  MMM      MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN       NN
                                  MMMM    MMMM  AA        AA       II       NNNN      NN
                                  MM MM  MM MM  AA        AA       II       NN NN     NN
                                  MM  MMMM  MM  AA        AA       II       NN  NN    NN
                                  MM   MM   MM  AAAAAAAAAAAA       II       NN   NN   NN
                                  MM        MM  AAAAAAAAAAAA       II       NN    NN  NN
                                  MM        MM  AA        AA       II       NN     NN NN
                                  MM        MM  AA        AA       II       NN      NNNN
                                  MM        MM  AA        AA       II       NN       NNN
                                  MM        MM  AA        AA   IIIIIIIIII   NN        NN
                                  MM        MM  AA        AA   IIIIIIIIII   NN         N



****************************************************************************************
****************************************************************************************
****************************************************************************************
****************************************************************************************
****************************************************************************************

****************************************************************************************
****************************************************************************************
****************************************************************************************
```

```
*****************************************************************************************
** VM/370 VERSION 06 LEVEL es PLC Pack **************** CLASS A *** DEV 00E **********


           USERID    ORIGIN      MAINT       MAINT                                      VV
                                                                                        VV
           DISTRIBUTION CODE     MAINT                                           3333333333
                                                                                33333333333
           SPOOL FILE NAME TYPE  SOSPLM      MEMO                               33      VV3
                                                                                         V3
           CREATION DATE         04/06/19 14:47:24                                        3
                                                                                       3333
           SPOOL FILE ID         0860                                                  3333
                                                                                          3
           RECORD COUNT          4680                                                     3
                                                                                33        3
                                                                                33333333333
                                                                                 3333333333

                                                                                   VM/370 R



                                    MM        MM   AAAAAAAAAA    IIIIIIIIII   NN        NN
                                    MMM      MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN       NN
                                    MMMM    MMMM  AA        AA       II       NNNN      NN
                                    MM MM  MM MM  AA        AA       II       NN NN     NN
                                    MM  MMMM  MM  AA        AA       II       NN  NN    NN
                                    MM   MM   MM  AAAAAAAAAAAA       II       NN   NN   NN
                                    MM        MM  AAAAAAAAAAAA       II       NN    NN  NN
                                    MM        MM  AA        AA       II       NN     NN NN
                                    MM        MM  AA        AA       II       NN      NNNN
                                    MM        MM  AA        AA       II       NN       NNN
                                    MM        MM  AA        AA   IIIIIIIIII   NN        NN
                                    MM        MM  AA        AA   IIIIIIIIII   NN         N



                                    MM        MM   AAAAAAAAAA    IIIIIIIIII   NN        NN
                                    MMM      MMM  AAAAAAAAAAAA   IIIIIIIIII   NNN       NN
                                    MMMM    MMMM  AA        AA       II       NNNN      NN
                                    MM MM  MM MM  AA        AA       II       NN NN     NN
                                    MM  MMMM  MM  AA        AA       II       NN  NN    NN
                                    MM   MM   MM  AAAAAAAAAAAA       II       NN   NN   NN
                                    MM        MM  AAAAAAAAAAAA       II       NN    NN  NN
                                    MM        MM  AA        AA       II       NN     NN NN
                                    MM        MM  AA        AA       II       NN      NNNN
                                    MM        MM  AA        AA       II       NN       NNN
                                    MM        MM  AA        AA   IIIIIIIIII   NN        NN
                                    MM        MM  AA        AA   IIIIIIIIII   NN         N
```

Brown University
Student Operating System
Program Logic Manual

by

Sidney H. Gudes

Program in Computer Science

Brown University

Box F

Providence, Rhode Island   02912

May 20, 1978

1 INTRODUCTION

The Brown University Student Operating System is an assembler-interpreter developed for use in the introductory assembler programming course at Brown (CS100). The (simulated) machine executes a simplified version of the IBM S/360 instruction set (basically the elimination of floating point, decimal, RR, and SS instructions). Memory consists of 4K 32-bit words, the first sixteen of which are GPRs.

The assembler is similar in form to the S/360 version.

The purpose of this manual is to describe the internal structure of SOS so that those who wish to make modifications or add features to the system (or to just understand it) may do so without too much trouble.

1.1 Purpose

SOS was designed to be used as an instructional aid to those who are learning assembler language and machine-level aspects of a computer.

It was felt that S/360 assembler, although powerful, would not be appropriate for a beginning student since it would force the absorption of a large instruction set, base-displacement addressing, and other advanced features of the 360. This seemed too overpowering for someone who has never been exposed to the inner workings of a computer. SOS provides a "crutch" for the beginning assembler programmer, providing basic assembler concepts and a sound background in machine architecture.[1]

In order to provide a less complicated learning environment and a more helpful debugging environment, the following features are included in SOS:

- 4096 words of core, addressed absolutely from 0 to 4095 (X'FFF'), all of core being available to the user (i.e., no in-core operating system)
- sixteen GPR's, occupying the first 16 words of core
- simple and powerful I/O capabilities, including automatic data conversion
- extensive trace, snap, and abend dump facilities.

1.2 History

SOS was originally conceived by Andries van Dam while at Brown University in the mid-1960's. A version of SOS was written in 1968. This version was somewhat primitive, requiring fixed format, a

[1]Many minicomputers today have an instruction set similar to SOS.

confusing indirect address specification mechanism, only 32 columns of input, and a non-deterministic macro generator. Over the years some changes were made by patching code, but due to the non-structured manner in which the old system was written, it was difficult to do much with it.

In the spring of 1976 it was decided that SOS was due for a rewrite and a few improvements. A team consisting of Sidney H. Gudes, Peter J. Relson, Carl C. Gallagher, Robert F. Gurwitz, and Philip J. Wisoff, with the advice and suggestions of SOS users, proposed and implemented the new version of SOS during the fall of 1976, with further work being done by S. H. Gudes through the spring of 1978.

1.3 Changes to SOS

Aside from internal improvements in documentation and modularity, the new version of SOS has several system changes which are (hopefully) more user oriented or clearer. These are:

- cataloguing and editing facilities are no longer available
- /JOB is now coded as /SOS
- the studentname and jobname parameters have been eliminated
- assembler labels may be up to eight characters in length
- symbolic macro parameter names may consist only of alphamerics
- continuation cards have been eliminated, except for macro calls and prototypes
- CCW 5 now recognizes the count field as the number of lines to skip after the carriage has been returned
- CCW 7 has been added to set page headers at execution time
- /SOS parameters have been radically changed
- SOS core is fixed at 4096 fullwords
- SOS core is not zeroed prior to loading
- the END statement no longer requires an operand; the default execution address is X'10'
- machine code programs are indicated by a /SOS card parameter, and use "*" in column one for comments
- assembler source statements are free format
- the first 72 columns of all statements are scanned
- TITLE, EJECT, and SPACE assembler pseudo-ops have been added
- EBCDIC as well as hex is given in dumps
- a cross-reference of symbols is available
- the "greater than" symbol (>) is the indirecting indicator
- C type character constants are permitted in literals
- the macro generator works (!)
- PUTC, PUTX, and PUTD work normally while the trace is on
- the TM instruction has its mask in the core location.

1.4 Using This Manual

It is assumed that the reader is familiar with the operation of SOS as described in the SOS Reference Manual[1].

Furthermore, it is assumed that the reader is familiar with the basic concepts of two-pass assemblers, macro generators, and software simulation of machine architectures.

This manual is divided into several distinct sections. Section two describes the general structure of the SOS system, including flow of control through modules, data structures used, and design decisions. Also included is a list of features and possible modifications which one may wish to make to the SOS system.

Section three contains in-depth looks at the various modules which make up the system. Each module is presented with a one or two page description of its function and any special considerations which it may take into account. Preceding each of these is a short paragraph describing the function of the routine. Also, a cross-reference of the routines called by the module, the routines which call the module, external references, and library macros used is included. For external references, if the reference is within another module, the name of that module is placed in parentheses following the reference name. For routines called and library macros used, if the routine or macro is not within SOS, the source is placed in parentheses following the routine or macro name.

Section four describes the format of various control blocks used within the SOS system.

Section five describes the format and usage of SOS internal macros. Some macros are used under CMS only; these are indicated by "CMS" in parentheses following the macro name.

Section six describes the various entries contained within SOSCB, the central SOS information area. These are arranged in alphabetical order and contain short descriptions of the functions of the entries.

There is no one section of this manual that can be read in and of itself to gain a complete understanding of the function of a particular aspect of SOS. The overviews in section two, the internal documentation of the code itself, as well as a general understanding of the contents of the Reference Manual, are necessary in order to see how the pieces of SOS fit together.

Throughout this manual, references of the form SOSxxxxx generally refer to entries in SOSCB, an alphabetical listing of which can be found in section six.

[1]Gudes, S., R. Gurwitz, and P. Relson, "Brown University Student Operating System Reference Manual," Program in Computer Science Technical Report 25, Brown University, Providence, Rhode Island, January 1977 (revised August 1977).

2 OVERVIEW

The SOS system is broken up into three logical parts: the batch supervisor (referred to as "phase 0" in this document); the assembler (including the macro generator) and the machine code loader (referred to as "phase I"); and the executor (referred to as "phase II").

The general flow of control through the system is as follows: phase 0 is initially invoked when SOS is given control by the operating system. Free storage is allocated, SYSIN and SYSPRINT are opened, and the SOS environment is initialized. The first statement from SYSIN is read (if it is not a /SOS card, SYSIN is flushed until a /SOS is read; if EOF occurs a message is printed and processing ceases). Phase I is then invoked, which scans the /SOS card (and any continuation), sets any parameters into SOSCB, opens and loads the macro library (if both ASM and MACRO are in effect), assembles or loads the source program (depending on whether the ASM or MACHINE option is in effect), and finally prints the user data if the DATA option is in effect. Phase 0 next checks to see whether the number of errors which occurred during phase I exceeds the error limit specified on the /SOS card (or defaulted from DEFBLOCK). If it does not, phase II is invoked to run the user program. On return from phase II, phase 0 reads the next record from SYSIN to see whether it is a /SOS card. If it is, the above is repeated; if not, we flush to a /SOS card. If a /SOS is not found before EOF, free storage is returned to the operating system, files are closed, and we return to the caller.

The design of SOS is such that all I/O is done through two central routines: IOMATIC, which performs I/O to SYSIN, SYSPRINT, SYSUT1, and SYSUT2; and MACOPEN, which performs I/O to SYSLIB. All modules in the SOS system perform I/O through these routines. This, coupled with the fact that free storage is allocated from the operating system in one place (namely the SOS mainline) and the GETBLOK and FREEBLOK management routines are used to allocate storage within this block, allows SOS to be "tuned" to the operating system on which it is running by merely rewriting four routines: SOS, PHASEI, IOMATIC, and MACOPEN. At present, versions exist for CP-67/CMS and OS/370.

Two special entry points in phase 0, named ABENDSOS and HOPEFUL, may be invoked by any phase at any time. ABENDSOS is invoked when an unrecoverable error is found in the SOS system. Free storage is released and files are closed, an informative message is printed, and an abend exit appropriate to the operating system on which SOS is running is taken. HOPEFUL is invoked when an error severe enough to terminate the current job, but not severe enough to terminate the running of the rest of the jobs in the batch, is detected. The SOS supervisor is re-entered at a point which will scan for the next /SOS card, effectively flushing the current job.

2.1 Free Storage Management

SOS free storage is divided into two parts: SOS core, which is used by the macro generator as a scratch work area, and a large free storage area which is allocated at SOS initialization and freed before SOS returns to the operating system.

The GETBLOK and FREEBLOK routines are used to perform storage management within this area. Returning free storage to the pool is not sophisticated; if the block to be returned is at the end of the allocated blocks, the free storage management pointer is decremented by the length of the block; otherwise nothing is done. This scheme is used because: a) all storage allocated by the assembler is needed throughout the job anyway; and b) the macro generator only allocates via GETBLOK/FREEBLOK if SOS core is full, a very unlikely occurrence.

This scheme fits well into a batch processing environment because resetting free storage for each job is merely a matter of pointing the free storage management pointer to the start of the free area.

In order to avoid extra work, MACOPEN reserves a portion of storage at the beginning of the free area into which it reads the SYSLIB macro definitions. IOMATIC, in some implementations, allocates buffers at the beginning of the free storage area. The management pointer starts beyond these, so that the SOS library need be opened and scanned only once, and IOMATIC buffers need be allocated only once.

CMS storage management

The SOS mainline attaches a segment (of 64 blocks) at device three. This segment (which can be treated as core due to the peculiarities of Brown's segment system) is used as a half-megabyte of scratch storage by setting SOS@FROR, SOS@FRCR, and SOS@FRTP to X'300400', and setting SOS@FRND to X'3FFFFF'. The internal GETBLOK and FREEBLOK routines are called to allocate storage from this pool and maintain the appropriate pointers. Before returning to CMS, the segment at device three is detached. SOS core is allocated in user core using the CMS FREE routine and is freed via the FRET routine before returning to CMS.

OS storage management

The SOS mainline does a variable GETMAIN for all free storage (asking for at least 35K bytes). If it is not available, SOS will abend. If it is, 6K is then released for use by the operating system. SOS@FROR, SOS@FRCR, and SOS@FRTP are set to point to the start of free storage, and SOS@FRND is set to point to the end of storage so that GETBLOK and FREEBLOK can manage free storage successfully. SOS core is then allocated in the first 16K of this area. Before returning to OS, SOS does a FREEMAIN of the storage which was not freed by the initial 6K FREEMAIN.

2.2 File Management

The SOS system consists (logically) of five datasets: SYSIN, containing SOS JCL, assembler or machine language programs, and data; SYSPRINT, which contains output produced by SOS; SYSUT1, to which the assembler source is written so that it may be read by pass 2 of the assembler; SYSUT2, which contains card images generated by the macro generator; and SYSLIB, which contains the SOS system macro definitions (this is handled exclusively by MACOPEN).

SYSIN, a card image R/O file, is accessed by SOS, PHASEI, and JCLYZER, to perform JCL processing; by MASHPROC, to perform machine language loading; by PASS1, to process the statements for pass 1 of the assembler; and by PHASEI to print the data cards.

SYSPRINT, a VBA W/O file, is accessed by SOS to print headings; JCLYZER to print the JCL options; PHASEI to print the message block and user data; ERRORHND to print error messages; MASHPROC to produce the machine code listing; PASS2 to produce the assembler listing; SVCALL to produce output from PDUMP, PUTD, PUTX, and PUTC; TRACE and ABNDTRCE to produce trace information; DUMP to produce a full or partial core dump; and EXECUTE to produce execution statistics messages.

SYSUT1 is a card image R/W file. Every time a read is done from SYSIN, the SOSWRRD flag in SOSCB is checked. If it is on, the card read from SYSIN is written to SYSUT1. In this way pass two of the assembler can re-read the source statements to generate code, and SVCALL can read user data cards.

SYSUT2, a card image R/W file, is written to by MACEXPND when a macro statement is generated. It is then read by pass two of the assembler in order to generate code.

The SOS mainline, PHASEI, and EXECUTE are responsible for opening, closing, and repositioning the files. IOMATIC is responsible for doing the physical I/O operations. Thus all routines other than these four (and MACOPEN) see only the logical file system described above, and are not dependent on the physical implementation.

Physical file system - CP-67/CMS

SYSPRINT is treated as a variable length file, lrecl 121. SYSUT2 is a card image file, blocked ten items per block. SYSIN is a card image file, blocked ten items per block. SYSUT1 does not physically exist for this implementation; rather, after JCLYZER has scanned the JCL for the current job, SYSIN is noted, is read sequentially from there, and is then pointed to the statements following the JCL to be re-read sequentially. After assembly, SYSIN is set to point to the first data card for execution. After execution, SYSIN is pointed to the statement following the last statement read by PHASEI, which in most cases will be the statement following the /END card.

Thus when SYSUT1 is being logically accessed, SYSIN is actually being physically read. Only SOS, PHASEI, and IOMATIC are aware of the physical reality; the rest of the system looks only at the logical functioning.

This system is advantageous to the logical system in that it saves the overhead necessary to access a physical SYSUT1.

<u>Physical file system - OS/370</u>

There are actually two different versions of the OS file system. One assumes that NOTE/POINT macros can be used on the SYSIN data set (i.e., the SYSIN data set is on a direct access device, on tape, or from JES) and the other assumes that SYSIN cannot be NOTE/POINTed. The difference is indicated to the OS version by the PARM field of the EXEC statement. PARM=DA means that SYSIN can be NOTE/POINTed, PARM=SQ that it cannot. The default is SQ. The DA version is equivalent to the CMS version described above: SYSUT1 does not physically exist, but is simulated by remembering the starting location of the user's program and resetting SYSIN to that point when necessary. In the SQ version, SYSUT1 does exist, and the physical file system is equivalent to the logical file system described above. Needless to say, for any non-trivial (in length) program, the DA version will be much cheaper in terms of cutting down on I/O operations.

<u>MACOPEN - accessing the SOS macro library</u>

The last file in the SOS system, SYSLIB, is accessed only by the MACOPEN routine, and then only if MACRO and ASM have both been specified (or defaulted) on the /SOS card. The library is set up as a partitioned dataset under OS or as a standard format MACLIB under CMS.

The directory records are read into core, the macros are read, Macro Control Blocks are set up for each macro, and the definitions are read into free storage. Once the macros have been successfully read into the free area, the free storage management pointers are set to indicate that the start of free storage is beyond the definitions, thus allowing the definitions to remain in core between runs in the batch without being re-read. When the library has been exhausted, it is closed and is not accessed for the remainder of the batch.

2.3 Supervisor

The supervisor is responsible for allocating and initializing the free storage area and SOS core, for processing JCL, and for performing file opens, closes, and points in the correct sequence.

The supervisor is contained in three routines: SOS, PHASEI, and EXECUTE. SOS controls free storage management and some file management, as well as scanning for JCL. PHASEI controls scanning of /SOS cards for parameters, scanning user data, and searching for /DATA and /END cards, if necessary. EXECUTE is responsible for positioning SYSUT1 to point to the start of the user's data cards so that they may be read by the executor.

JCL Scanning

Initially, SOS reads from SYSIN until it finds a /SOS card. If SYSUT1 physically exists, it is opened for OUTIN; if it only logically exists, it is NOTEd and the note information is saved. PHASEI is then invoked, which parses the parameters off the /SOS card and places them into the appropriate SOSCB locations. Subsequent cards are read and parsed until one that is not a /SOS card is encountered.

Assembly and Loading

If both ASM and MACRO are in effect, MACOPEN is called to load the SOS macro library. If there were any errors in the library macros, an error summary is printed for the user's benefit. If the ASM option is in effect, SYSUT2 is opened for OUTIN. At this point the assembler (or machine code loader) is invoked to process the user's program. On return, if the last statement processed by the assembler or loader was an END or FFF card, a scan for a /DATA card is initiated. If one is found, or the last statement processed by the assembler or loader was a /DATA, the user's data up to /END, /SOS, or EOF is scanned (printed if DATA was specified as a /SOS option). Subsequent /DATAs are accepted as data cards.

Once EOF, /SOS, or /END are hit, or if these were hit by the assembler or loader, that statement is saved for later processing. We then see whether the program is in assembler or machine format; if assembler, the second pass of the assembler is called to generate code and a listing.

SYSUT2 is then closed if necessary.

Executing the Program

SOS tests the error count to see whether it is less than the error limit; if so, EXECUTE is called to run the user's program. SYSUT1 is pointed to the first statement of user data, and the fetch-increment-execute cycle is started until execution is completed (see section 2.6). At this point the number of instructions executed is printed.

Batch Processing

If there is a physical SYSUT1, it is now closed. A logical SYSUT1 (i.e. SYSIN) is pointed to the final statement of the job which was just processed.

SOS now checks to see whether the last card saved by PHASEI is a /END card. If it is, we read the next statement from SYSIN. If it is not, we print a warning message and use that statement as the next SYSIN input record. We then return to JCL Scanning above until EOF.

ABENDSOS

This entry point is invoked if a truly fatal error occurs (see Appendix F of the Reference Manual for a list of these). All files are closed, free storage is returned to the operating system, an active SPIE is freed if there is one, and the informative error message is written to the terminal (CMS) or written to programmer (WTO with ROUTCDE=11) under OS. Under CMS we return with an error code; under OS we abend so that a SYSUDUMP card may be used to get a core dump.

HOPEFUL

This entry point is invoked when an error severe enough to preclude running the current job, but not serious enough to prevent running the rest of the jobs in batch, is encountered (these are the 4000 series errors listed in Appendix E of the Reference Manual). This routine frees an active SPIE if there is one, closes SYSUT2 if it is open, restores the SOS supervisor registers, and transfers control to Batch Processing above.

2.4 Assembler

The SOS assembler is implemented as a two pass assembler. The supervisors for each pass are called PASS1 and PASS2, respectively.

Assembler Pass One

The purpose of pass one is to scan all the source statements in the user's assembler program, placing symbols into the symbol table, keeping track of the location counter (i.e., processing ORGs, DSs, and DCs), determine where the literal pool can start, invoke the macro generator when necessary to save macro definitions or expand macro invocations, and set up and save a Card Image Status Word (CISW) for each statement which is processed.

ORG, DS, and EQU statements are completely processed during pass one. The location counter is initially set to X'10' and is incremented by one for each instruction, two for a CCW, one to seventeen for a DC, one to 4095 for a DS, or reset for an ORG. A label is stored if there is one on an instruction, CCW, DC, or DS, or for an EQU. The largest value taken on by the location counter during pass one is used to determine the top of the literal pool to be filled by pass two.

PASS1 has relatively few functions; it basically reads card images from SYSIN and passes them to DOPASS1, which does the main processing. This allows DOPASS1 to be invoked from either PASS1 or MACEXPND (the macro generator) without having to know which one invoked it.

Interfacing With the Macro Generator

When a MACRO statement is encountered in the user's input stream, MACSAVE is called to do whatever it will with the statement. MACSAVE stores the CISWs for the MACRO statement up to and including the MEND statement.

When a macro invocation is encountered, MACEXPND is called with the invocation statement and the MCB for the macro. MACEXPND generates each model statement and passes the statement to DOPASS1 to be processed. For this reason, DOPASS1 is recursive to two levels. MACEXPND handles nested macro calls by recursively calling itself, so that the card image passed to DOPASS1 is an assembler op-code or pseudo-op.

Assembler Pass Two

Pass two of the assembler re-reads the card images that were processed during pass one and scans the operand fields to generate code for each instruction.

Since CISWs were saved for each card image processed by pass one, needless re-scanning of the card images is avoided. For statements that were completely processed during pass one (EQU, DS, ORG), the CISW contains the information gathered in pass one, so that pass two need merely print the statement and update the location counter if necessary. For other statements, the CISW contains the starting location of the operand field, so that scanning need not involve the label or opcode fields. Since all symbols were defined by pass one, there should be no trouble in evaluating the operand fields, which are merely expressions of symbols and constants.

The instructions are assembled into machine code (or constants in the case of DC) and are loaded into core. The statements are printed along with the statement number, LC value, and assembled machine code. Errors are indicated by bits 0 and 16 of the CISW (see section 4.3) and appropriate error messages are generated.

If the statement was macro generated, bit 1 of the CISW is on, indicating that the card image to be read is in SYSUT2 (rather than SYSUT1, where card images read from SYSIN by pass one reside) and that a plus sign is to be placed into the listing (if PRINT GEN is in effect).

2.5 Macro Generator

The macro generator can be divided into three basic sections: fetching and processing macro definitions from the SOS macro library (SYSLIB); saving user macro definitions; and generating code from macro invocations.

Fetching SYSLIB Macro Definitions

The procedure for processing SYSLIB is the same under both CMS and OS, although the actual mechanisms used to do the fetching are different in each case. SYSLIB is in MACLIB format under CMS and a partitioned data set under OS.

Processing consists of fetching each directory record, parsing the macro name and disk information off the record, and then going to each macro and reading it into core. The macro is checked for MACRO, MEND, and prototype statements. The prototype and its continuation (if any) are scanned and a Macro Control Block (MCB) is set up (see below). The macro model statements are then read into the free storage area. Each group of model statements is preceded by a fullword of zeros (a null forward pointer) and a fullword containing the number of statements in the definition. Needless to say, if there is not enough room in the free storage area to contain the macros, SOS will abend.

The MCBs are linked together with the head pointed at by SOS@LBTP. Each MCB contains a pointer to the head of the group of model statements.

Saving User Defined Macros

When the assembler detects a MACRO statement, MACSAVE is called to save the user macro definition. The CISW for the MACRO statement is stored. The prototype and any continuation are scanned, CISWs are stored for them, and an MCB for the macro is set up (see below). The MCB is linked onto the SOS@MCNM list. The model statements are then read into SOS core, preceded by a forward pointer (initially null) and a fullword of the number of statements in the current block. If we run out of space in SOS core, the SOS core block is linked to blocks which are allocated in the free storage area, each block preceded by a forward pointer and a block item count. A CISW is stored for each model statement and for the MEND statement. We cannot merely move model statements into the free storage area sequentially (as we do in reading definitions from SYSLIB) because CISWSTOR might allocate a block in the middle of processing.

Setting Up A Macro Control Block

The MCB is diagrammed in section 4.5. The label and name fields of the prototype are parsed and placed into the MCB. The library flag is then set to one if the macro was from SYSLIB or to zero if it was from the user's input stream. The parameters are then parsed off and set into the MCB. Positional parameters are stored as seven character names in an eight byte field for each name. Keyword parameters are stored as seven characters of parameter name, followed by a one-byte length of the default value, followed by the default value itself. If the default value is null, the length field is set to X'FF'. Once the parameters have been inserted, the number of positional parameters and the number of keyword parameters is set into the MCB. The processor which caused the MCB to be set up will set the model statement pointer to the linked list of model statement blocks.

It is not known a priori how much room the MCB will take up. Therefore, it is assembled at the first available location in the free storage area, and a pointer is returned to indicate how far it has gotten.

Processing a Macro Invocation

The heart of the macro generator is MACEXPND, which expands and processes macro invocations.

When the assembler detects a macro invocation statement, it passes the statement to MACEXPND. The statement is parsed and an Active Macro Table (AMT) is set up (see below) to define equivalences between the symbolic parameter names and the parameters' value for this invocation. The model statements associated with the definition are then fetched, one at a time, symbolic parameter substitution is performed (see below), the statement is written to SYSUT2, and the expanded statement is sent to the assembler (DOPASS1) to be parsed and have its CISW stored. On return from the assembler, the next statement is fetched, expanded, and sent to the assembler, until there are no more model statements.

MACEXPND stores the CISWs for the invocation statement and its continuation, if any. DOPASS1 stores the CISWs for the generated statements. It is necessary for DOPASS1 to be recursive to two levels of call, which is implemented by using a nineteenth word of the caller's save area as a pointer to the previous save area in a stack.

MACEXPND parses each generated statement to isolate the label and opcode fields. If the statement is a comment or has a missing opcode, the CISW is stored at that point and the assembler is not invoked for that statement. If the opcode is an assembler mnemonic or pseudo-op, DOPASS1 is invoked as described above. If, however, the opcode is a macro name, the fun begins. MACEXPND calls itself recursively, passing itself a parameter list which contains a pointer to the MCB of the new macro as well as the invocation statement. Since MACEXPND allocates a dynamic work area each time it is entered, it is recursable to any level. Thus it continues to generate statements, invoking itself or DOPASS1, until all the statements have been generated. Once they have been generated, the free storage area is returned to the pool, and we return to caller, which can be either MACEXPND or DOPASS1. If it is MACEXPND, expansion of the previous invocation is resumed; if it is DOPASS1, the next source statement from the user's input stream is read.

Setting Up the Active Macro Table

Since the number of parameters is known (from the MCB), the AMT can be allocated to the appropriate size beforehand. It is allocated in SOS core, unless there is not enough room, in which case it is allocated in the free storage area. The current value of &SYSNDX is converted to character and made an entry in the AMT. The label parameter, if any, is then placed into the AMT. The invocation statement (and its continuation, if any) is then moved to the bottom of the AMT. AMTSETUP is called to do the actual parsing of the invocation statement. Each positional parameter on the invocation statement is matched to the positional parameters indicated on the MCB. If any are missing, their values are set to null for this invocation. Keyword parameters are then processed by placing a length and pointer to the value in the AMT. If the value has been re-defined, the pointer points to the string in the invocation statement at the bottom of the AMT; if the default is in effect, the pointer points to the value in the MCB. In this way, there is no need for the routine which expands the statements to distinguish between keyword and positional parameters.

Expanding a Model Statement

The model statement is scanned for an ampersand. If none is found, we are done. If the character following the ampersand is another ampersand, the remainder of the model statement is moved up one character and we resume the scan from the character after the ampersand. Otherwise we scan for a delimiter, look the parameter up in the AMT, spread or squeeze the statement to make room for the value, and move the value in. Scanning resumes at the point following the moved-in value. In order to take care of the case in which a rather long value occurs at the beginning of the statement followed by null values for the rest of the statement, a 144 character expansion buffer is used.

After all the expansion has been performed, the statement is parsed and re-formatted so that the opcode begins in column ten (if possible) and the operand field begins in column sixteen (if possible).

2.6 Executor

The purpose of the executor is to simulate the SOS machine by fetching and executing instructions in SOS core until one of three conditions is met: a program abend occurs; a /SOS card limit is exceeded; or a HALT instruction is encountered. A SPIE is set on a divide exception so that SOS, rather than the operating system, will abend the user if this occurs.

Firing It Up and Keeping It Going

The address at which execution is to begin is contained in SOSSTART on entry to the executor. This is an SOS address (i.e. in the range X'000' to X'FFF'). Starting at this address, the instruction is fetched (this involves converting the SOS address to a /360 address by multiplying by four and adding to SOS@CORE), the program counter is incremented by one to point to the next instruction, and the instruction which was just fetched is broken into its constituent parts, the effective address is calculated if need be, and the instruction is simulated (note that branch instructions may alter the contents of the program counter).

User Errors

There are many error conditions which can arise, most of them involving the program counter or an effective address exceeding the bounds of SOS core. Other error conditions involve the specification of register X'F' in a double register instruction, invalid opcodes, exceeding the instruction or print limit, and more. These are tested for in the appropriate routines, and if a condition which warrants halting execution occurs, SOSABEND is set to an abend code and abend actions, as described below, are performed.

Execution Time Trace Functions

Tracing functions are extensive. For each instruction executed, information on the instruction (register, register contents, effective address, etc.) is saved in TRACEBLK (see section 4.10) as it is simulated. The TRACE routine is then called. If the instruction is a branch, the appropriate information (section 4.2) is stored on a stack of the last ten branch instructions executed (external name BRBLKS), shoving the tenth previous instruction out if necessary. If the trace is on (indicated by SOSTRST), FORMTRCE is called, passing it TRACEBLK, to format a line of trace information, which is then printed. If the trace is off, the information in TRACEBLK is saved on a stack of the last ten instructions executed (external name TRBLKS), again shoving the tenth last instruction out if necessary.

Warning Actions

If execution of an instruction causes a warning (overflow, trace count exceeded, or a divide exception), the last ten branch and executed instructions are formatted and printed to aid the user in finding the error. An informative error message is also printed.

Abend Actions

If the user's program abends, the last ten branch instructions and last ten executed instructions are formatted and printed to aid in debugging. A full dump of SOS core is also produced, and the contents of SOSPLINE are printed. The number of instructions simulated is then calculated and printed.

HALT Instruction Actions

If a HALT instruction is encountered, a message to that effect is printed, and if the DUMP option was specified (or defaulted) on the /SOS card, SOS core is dumped, as well as SOSPLINE. The number of instructions executed is then calculated and printed.

I/O

When an SVC instruction is encountered, a loop is entered which starts at the top of the user's CCW chain and chases down it until the chain bit is zero or an abend-forcing condition occurs.

The PUT instructions cause the specified data to be translated and placed into the user's output buffer (SOSPLINE). A cursor along this buffer (SOSCURSE) is used to keep track of where we are. This cursor is reset by the SKIP and BKSP functions. RET causes the current user buffer to be written to SYSPRINT, the buffer to be cleared to blanks, and the cursor to point to the beginning of the buffer. The tab stops are implemented by setting a vector consisting of 120 bytes (SOSTABS) to all zeros. Each time a tab is set, the corresponding position in SOSTABS is set to one; each time a tab is cleared, the position is set to zero. In order to tab to the next stop, SOSCURSE is used to determine our present position within SOSTABS, and we TRT for a one.

The GET functions cause a record to by read from SYSUT1. If EOF occurs, the appropriate bit of the CCW status byte is turned on. If EOF occurs a second time, execution abends (a double-flag system is used to remember whether EOF has occurred; this is due to the way SOS remembers whether there was any user data). Once the card image has been read, it is parsed in the manner appropriate to the function called (GETX, GETD, GETC) and the information is placed into the specified SOS core locations.

Dumps (PDUMP) are performed by printing a heading identifying the dump instruction and then calling the DUMP routine with the specified core bounds as an argument. This is the same routine which produces the final dump of SOS core.

Finally, the trace is turned on and off (TON, TOFF) by setting SOSTRST to the appropriate value and setting the output headers (SOSHDR, SOSHDPR1) to either blanks (if TOFF) or the trace headers (located at external label FIRSTL in EXECUTE) if TON.

2.7 Making SOS Do What You Want It To

Message Block: to change the contents of the message block, the DEFBLOCK macro should be updated to the new message (note the number of lines and line length fields), and the DEFBLOCK csect should then be assembled and link-edited.

Macro Generator: to disable macro generation facilities, set SOSMXNST to zero.

Setting Defaults for /SOS Card Parameters: change the appropriate item in the DEFBLOCK macro, then assemble and link-edit the DEFBLOCK csect.

/SOS Card Parameters: to add new parameters, delete old parameters, or change parameters, merely add (delete, change) the new flag entry to SOSCB and make an entry in the options table in JCLYZER (if it is a numeric parameter, a default will have to be set up in DEFBLOCK also). The order of default parameters in DEFBLOCK must correspond exactly to their order in SOSCB; expansion space has been left in SOSCB for this purpose. If parameters are being added or deleted, the MVCs which set the default values into SOSCB must have their length fields adjusted accordingly. These MVCs are in the INTSOSCB internal routine in the SOS module.

Indirecting Levels: to turn off indirecting capabilities, set SOSINDLV to zero. Indirecting statements will still assemble or load correctly, but they will abend in execution.

Using PARM=DA: if this parameter is specified, the device on which SYSIN is contained must be able to be NOTEd and POINTed; that is, it must be tape, direct access device, or JES. Even if this condition is met, if SYSIN consists of a concatenation of datasets residing on different volumes, NOTE/POINT will not work (OS gets confused as to which volume is to be POINTed); PARM=SQ must be used in this case.

Changing JCL: So you say you don't like /SOS, /DATA, and /END? Since all card images are checked to see whether they are JCL by the same central routine (namely ISITJCL), modifying this one routine will allow the use of alternate JCL. See ISITJCL (section 3.2.7) for more details.

Adding an Opcode: adding a new operation code to SOS is not very difficult, but there are many tables that would have to be modified. These include those in OPSRCH, FORMTRCE, DOPASS1, and EXECUTE. In addition, code to emulate the function of the opcode would have to be written and placed into a csect (probably SINEXEC), with appropriate modification to the csect table in EXECUTE.

3 MODULE DESCRIPTIONS

3.1 Supervisor

3.1.1 ABENDSOS (CMS)

This entry point in the SOS mainline is invoked when a system error so severe that processing cannot continue is encountered. Files are closed, free storage is returned to the operating system, and an abend exit is taken.

CODED BY: S. H. Gudes

CALLED BY: ERRORHND IOMATIC JCLYZER MACOPEN MACSAVE MCBSETUP PHASEI SOS

ROUTINES CALLED: ERASE(CMS) FINIS(CMS) FRET(CMS) SEGMENT(CMS) SPIE(CMS) TYPE(CMS)

LIBRARY MACROS: CMS SPIE(CMS)

On entry, R1 should point to a parameter list consisting of a one byte error code, followed by a one byte message length, followed by a message.

Once addressability has been established: the message is typed to the terminal via SVC of the TYPLIN routine; SOS core is freed by a call to FRET; if there is an active SPIE it is terminated; a FINIS * * * is performed to close all files; the SYSUT2 data set is erased; the segment at device 3 is detached; and finally, the error code is loaded into R15 and the CMS nucleus is returned to.

3.1.2 ABENDSOS (OS)

This entry point in the SOS mainline is invoked when a system error so severe that processing cannot continue is encountered. Files are closed, free storage is returned to the operating system, and an abend exit is taken.

CODED BY: S. H. Gudes

CALLED BY: ERRORHND IOMATIC JCLYZER MACOPEN MACSAVE MCBSETUP PHASEI SOS

ROUTINES CALLED: ABEND(OS) CLOSE(OS) FREEMAIN(OS) SPIE(OS) WTO(OS)

EXTERNAL REFERENCES: SYSINDCB(IOMATIC) SYSPRDCB(IOMATIC) UT1DCB(IOMATIC) UT2DCB(IOMATIC)

LIBRARY MACROS: ABEND(OS) CLOSE(OS) FREEMAIN(OS) SPIE(OS) WTO(OS)

On entry, R1 should point to a parameter list consisting of a one byte error code, followed by a one byte message length, followed by a message.

Once addressability has been established: if there is an active SPIE it is terminated; the free storage allocated at the beginning of the job is returned to OS via a call to FREEMAIN; the addresses of all file DCB's are obtained and CLOSE is called to close all the files; WTO is called to write the error message to the user's JCL listing; and finally, the error code is loaded into R1 and the OS ABEND routine is called with this code and the DUMP option (to generate a core dump if a SYSUDUMP DD statement was included in the JCL).

3.1.3 HOPEFUL

This entry point in the SOS mainline is invoked when an error severe enough to cancel the current job, but not severe enough to halt processing, occurs. The SOS mainline is invoked to flush to the next job and continue processing.

CODED BY: P. J. Wisoff

CALLED BY: ERRORHND

ROUTINES CALLED: IOMATIC SPIE(CMS/OS)

LIBRARY MACROS: IOMATIC SPIE(CMS/OS)

Addressability is established. The registers in the SOS save area are loaded, effectively establishing the supervisor environment previous to the call which caused the error.

If there is an active SPIE it is freed. SYSUT2 is closed. If EOF occurred, the section of the supervisor which cleans up and returns to the operating system is given control. If not the section of the supervisor which scans for a /SOS card is given control.

3.1.4 JCLYZER

Parses out /SOS cards, setting the appropriate flags and/or counts in SOSCB, and then prints the option list.

CODED BY: R. F. Gurwitz and S. H. Gudes

CALLED BY: PHASEI

ROUTINES CALLED: ABENDSOS ERRORHND IOMATIC

EXTERNAL REFERENCES: DEFBLOCK SOSCB

LIBRARY MACROS: ABENDSOS CALL CMSREG DEFBLOCK IOMATIC SOSCB SOSENTER

It is assumed that the first /SOS card is in SOSCOMIN when this routine is entered. SOSTRTBL and SOSTRTNO in SOSCB are set to treat commas as blanks in scanning. A loop is entered in which the current JCL statement is written to SYSPRINT, the parameters are parsed off and set into SOSCB, and the next statement is read from SYSIN. This continues until the statement read is not a /SOS card.

Next the state of all flag and numeric parameters is written to SYSPRINT, whether or not they were specified on the /SOS card.

Errors from IOMATIC cause ABENDSOS to be invoked. Errors in parameter arguments or the parameters themselves cause an error message to be written to SYSPRINT and the parameter to be ignored (or a default to be assumed).

The processing is table driven, and the table consists of the following information for each parameter: its offset in SOSCB, its SOSMAX offset in DEFBLOCK (if any), a byte indicating whether it is a numeric(1) or flag(0) parameter, the value of the flag (if it is that type of parameter), the length of the parameter string (0 as the first counting number), and finally the string itself, left-justified in a nine character field.

3.1.5 MASHPROC

Invoked if the MACHINE option was specified on the /SOS card. The machine code program is read into SOS core, FF0 and FFF cards are processed, and appropriate error reports are generated.

CODED BY: P. J. Relson

CALLED BY: PHASEI

ROUTINES CALLED: ERRORHND IOMATIC

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CALL CMSREG IOMATIC SOSCB SOSENTER

No parameters are passed to this routine.

After initialization of counts and output headers, a processing loop is entered. If the current statement is JCL, a suitable return code is set and we return to caller. Otherwise it is sent to be processed. If it was an FFF card, the loop is exited, otherwise the next card is read and processed. If EOF occurs an appropriate return code is set and we return to caller.

Processing involves testing for the four types of cards. If it is a comment, it is merely printed. If it is an FFF card, the operand is evaluated. If it is valid, SOSSTART is set to that value. If it is missing or invalid, an error message is printed and SOSSTART is made X'10'. In any case, the card image is printed.

If it's an FF0 card, the operand is evaluated. If it is valid, the LC is set to that value and the card printed. If it is missing or invalid, the statement is printed along with an error message and the LC remains unchanged.

Otherwise, there had better be a blank in column nine. The word is scanned, making sure that there are exactly eight hexadecimal digits, and if it is valid, it is converted to binary. The LC is then tested for overflow (> 4095). If it has overflowed it is set to 4095, and if it is the first time that it has done so a warning message is printed. An overflow flag is set so that we do not print the message again. The flag is reset when an FF0 card is encountered. The word is then stored at (SOS@CORE) + 4*LC. The card is then printed.

If an FFF card did not end the machine code, ERRORHND is called to print a missing FFF card message and SOSSTART is made X'10'.

3.1.6 PHASEI (CMS)

Supervises assembly or machine language processing, printing of user data, and printing of number of error messages.

CODED BY: P. J. Wisoff and S. H. Gudes

CALLED BY: SOS

ROUTINES CALLED: ABENDSOS ERRORHND IOMATIC JCLYZER MACOPEN MASHPROC PASS1 PASS2

EXTERNAL REFERENCES: DEFBLOCK MACTAB(ERRORHND) SOSCB

LIBRARY MACROS: ABENDSOS CALL CMSREG DEFBLOCK IOMATIC MCB SOSCB SOSENTER

No parameters are passed to this routine, although the first /SOS statement of the current job is expected to be in SOSCOMIN. JCLYZER is called to process the /SOS card and any continuation. The current SYSIN pointer is then saved in SOSNXJOB and SOSOURCE so that we can point SYSIN to re-read the user's source program and data.

The message block is printed. If both the ASM and MACRO options are on, MACOPEN is called to open the system macro library. The library MCBs are scanned for errors, and if any are found the macro name and error description are written to SYSPRINT.

If the job is in assembler (ASM option), SYSUT2 is opened via IOMATIC (ABENDSOS is invoked if we cannot do this), control pointers are set up, and PASS1 is called to perform the first pass of the assembler. Otherwise, SOS core is munged up and MASHPROC is called to process the machine code program.

On return from the processor, the next action is based on the return code. If the program ended with an END or FFF card, a /DATA card is searched for. If found (before /END, /SOS, or EOF), the data card initial position is noted into SOSUDATA and the user data is printed (if the DATA option was specified). If the program ended with a /DATA card, the data card initial position is noted into SOSUDATA and the user data is printed (if the DATA option was specified). Nothing is done for the other cases.

The current contents of SOSCOMIN (basically the next JCL statement from the input stream) are saved for the batch supervisor. IOMATIC is called to note the current statement, which is saved in SOSNXJOB.

If the job is in assembler, SYSUT1 is pointed to re-read the input stream, and SYSUT2 is closed and opened for input (to re-read macro generated statements). SOS core is munged, and PASS2 is called to generate code. SYSUT2 is then closed.

Finally, we return to the caller.

3.1.7 PHASEI (OS)

Supervises assembly or machine language processing, printing of user data, and printing of number of error messages.

CODED BY: P. J. Wisoff and S. H. Gudes

CALLED BY: SOS

ROUTINES CALLED: ABENDSOS ERRORHND IOMATIC JCLYZER MACOPEN MASHPROC PASS1 PASS2

EXTERNAL REFERENCES: DEFBLOCK MACTAB(ERRORHND) SOSCB

LIBRARY MACROS: ABENDSOS CALL CMSREG DEFBLOCK IOMATIC MCB SOSCB SOSENTER

Note: this section describes PARM=SQ. PARM=DA is as under CMS above.

No parameters are passed to this routine, although the first /SOS statement of the current job is expected to be in SOSCOMIN. JCLYZER is called to process the /SOS card and any continuation.

The message block is printed. If both the ASM and MACRO options are on, MACOPEN is called to open the system macro library. The library MCBs are then scanned for errors, and if any are found the macro name and error description are written to SYSPRINT.

If the job is in assembler (ASM option), SYSUT2 is opened via IOMATIC (ABENDSOS is invoked if we cannot do this), the contents of SOSCOMIN are written to SYSUT1 for PASS2's benefit, SOSWRRD is turned on so that the source program will be written to SYSUT1, some control pointers are set up, and PASS1 is called to perform the first pass of the assembler. Otherwise, SOS core is munged up and MASHPROC is called to process the machine code program.

The next action is dependent on the return code. If the program ended with an END or FFF card, a /DATA card is searched for. If found (before /END, /SOS, or EOF), the data card initial position is noted into SOSUDATA and the user data is printed (if the DATA option was specified). If the program ended with a /DATA card, the data card initial position is noted into SOSUDATA and the user data is printed (if the DATA option was specified). Nothing is done for the other cases (/END, /SOS, or EOF).

The current contents of SOSCOMIN (basically the next JCL statement from the input stream) are saved for the batch supervisor. IOMATIC is called to rewind SYSUT1 and open it for input.

If the job is in assembler, SYSUT2 is closed and opened for input (to re-read macro generated statements). SOS core is munged, and PASS2 is called to generate code. SYSUT2 is then closed.

Finally, SOSWRRD is turned off and we return to caller.

3.1.8 SOS (CMS)

The system is initialized by allocating free storage and calling IOMATIC to open major files. A batching loop is entered which initializes SOSCB, calls PHASEI to process the input file, and if no errors were found, calls EXECUTE to run the program. This continues until SYSIN is exhausted, at which time free storage is returned, files are closed, and we return to CMS.

CODED BY: P. J. Wisoff and S. H. Gudes

CALLED BY: (CMS)

ROUTINES CALLED: ABENDSOS ERASE(CMS) ERRORHND EXECUTE FINIS(CMS) FREE(CMS) FRET(CMS) IOMATIC PHASEI SEGMENT(CMS) SPIE(CMS)

EXTERNAL REFERENCES: DEFBLOCK FVS(CMS) SOSCB

LIBRARY MACROS: ABENDSOS CALL CMS CMSREG DEFBLOCK FVS(CMS) IOMATIC NUCLEUS SOSCB SOSENTER SPIE(CMS)

On entry, R1 is expected to point to the SOS command and its arguments as per standard CMS conventions. If the filename is missing or is "*", we invoke ABENDSOS with argument 8. If not, we set the printer/listing parameter, if it was specified. The CMS FREE routine is called to allocate SOS core and I/O buffers. SOS will abend if there is not enough free storage for these.

IOMATIC is called to open the specified input file. If IOMATIC returns a non-zero return code, ABENDSOS is invoked with argument 1. IOMATIC is then called to open the output file, and ABENDSOS is invoked with argument 2 if the IOMATIC return code is non-zero.

Since in general the CMS version will be used (at Brown) under CMSBATCH, no page eject is forced since CMSBATCH will do it for us. IOMATIC is called to read the first card, transferring to the internal EOF/error handler if the return code is non-zero. We next pretty the output somewhat and enter the batching loop.

The current contents of SOSCOMIN are examined. If they are not a /SOS card, an error message is printed (unless we're coming from HOPEFUL) and we flush to a /SOS card, transferring to the EOF/error handler if necessary. The two output heading lines are written, and PHASEI is called to process the JCL and assembly or loading.

We print the error count, and if it is less than the error limit, EXECUTE is called to run the user program. Otherwise we print an error message. If EOF occurred for SYSIN, we print a "missing /END" card message and terminate. Otherwise, we clear out the output headers and point SYSIN to the next job in the stream (the point value as set by PHASEI).

If the last card read by PHASEI was not a /END, an appropriate error message is printed and the statement is assumed to begin the next job; otherwise, the next card is read from SYSIN. A page eject is forced, and we branch to the top of the loop, where the current card is tested to see whether it is a /SOS, etc.

If an I/O error occurred for SYSIN, an appropriate error message is printed. If EOF occurred, nothing further need be done. In either case, SOS core is freed via the CMS FRET routine, an active divide exception SPIE is freed if there is one, all files are closed, the utility file is erased, and the storage segment is detached. R15 is zeroed, and we return to CMS.

3.1.9 SOS (OS)

The system is initialized by allocating free storage and calling IOMATIC to open major files. A batching loop is entered which initializes SOSCB, calls PHASEI to process the input file, and if no errors were found, calls EXECUTE to run the program. This continues until SYSIN is exhausted, at which time free storage is returned, files are closed, and OS is returned to.

CODED BY: P. J. Wisoff and S. H. Gudes

CALLED BY: (OS)

ROUTINES CALLED: ABEND(OS) ABENDSOS CLOSE(OS) ERRORHND EXECUTE FREEMAIN(OS) GETMAIN(OS) IOMATIC PHASEI SPIE(OS) TIME(OS)

EXTERNAL REFERENCES: DEFBLOCK SOSCB

LIBRARY MACROS: ABEND(OS) ABENDSOS CALL CLOSE(OS) CMS CMSREG DEFBLOCK FREEMAIN(OS) GETMAIN(OS) IOMATIC SOSCB SOSENTER SPIE(OS) TIME(OS)

After establishing addressability, the passed parameter is examined. If it is "DA", SOSSYSRC is set to "D"; if it is "SQ", SOSSYSRC is set to "S"; otherwise SOSSYSRC remains unchanged. GETMAIN is called to allocate all of free storage. SOS will abend if there is not enough free storage (35K bytes). After saving the storage information, FREEMAIN is called to return 6K to the operating system for its own use. SOS core is allocated within this chunk, and SOSCB free storage management pointers are set to point past SOS core.

IOMATIC is called to open SYSIN. If IOMATIC returns a non-zero code, ABENDSOS is invoked with argument 1. IOMATIC is then called to open SYSPRINT, and ABENDSOS is invoked with argument 2 if the return code is non-zero.

A page eject is forced and the first card from SYSIN is read (transferring to the internal EOF/error handler if the return code is non-zero). We then enter the batching loop.

The current contents of SOSCOMIN are examined. If they are not a /SOS card, an error message is printed (unless we're coming from HOPEFUL) and we flush to a /SOS card, transferring to the EOF/error handler if necessary. The two output heading lines are written, and PHASEI is called to process the JCL and assembly or loading.

We print the error count, and if it is less than the error limit, we call EXECUTE to run the user program, otherwise we print an error message. If EOF occurred for SYSIN, we print a "missing /END" card message and terminate. Otherwise, we clear out the output headers and point SYSIN to the next job in the stream (the POINT was done by PHASEI).

If the last card read by PHASEI was not a /END, an appropriate error message is printed and the statement is assumed to begin the next job; otherwise, the next card is read from SYSIN. A page eject is forced, and we branch to the top of the loop, where the current card is tested to see whether it is a /SOS, etc.

In the EOF/error handler, if an I/O error occurred for SYSIN, an appropriate error message is printed. If EOF occurred, nothing further need be done. In either case, free storage is returned to OS via a call to FREEMAIN, and all files are closed. R15 is zeroed, and we return to OS.

3.2 Support Routines

3.2.1 CISWSTOR

Passed a card image status word (CISW), saves it for later use by PASS2.

CODED BY: P. J. Relson

CALLED BY: DOPASS1 MACEXPND MACSAVE

ROUTINES CALLED: GETBLOK

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CALL CMSREG SOSCB SOSENTER

On entry, R1 should contain the CISW to be stored. The CISW block count field of the current block (pointed to by SOS@LAST) is checked. If the block is full, a new 256-byte block is allocated by calling GETBLOK, chained to the previous block, and SOS@LAST is set to point to the current block. In either case, the count field is then incremented by 4 to point to the next available location, and the CISW is placed there.

Nothing is returned to the caller, and no error conditions are tested.

3.2.2 ERRORHND

Passed an error code, prints the associated error message.

CODED BY: R. F. Gurwitz

CALLED BY: EXECUTE GETBLOK JCLYZER MACSAVE MASHPROC MCBSETUP PASS2 PHASEI SOS

ROUTINES CALLED: ABENDSOS HOPEFUL IOMATIC

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: ABENDSOS CALL CMSREG IOMATIC SOSCB

On entry, R1 contains the code of the error message to be printed. The code is passed in the low order 16 bits of the register, specified as four hex digits. The first digit is the error type, followed by the code itself. For example, 0000301C would be macro generator error code 28.

The code is split into its constituent parts, the proper table is accessed, and the error message is moved into SOSERRBF and printed by calling IOMATIC; if IOMATIC returns a non-zero code when the message is printed, ABENDSOS is invoked with an abend code of 6.

In order to allow the executor to print helpful debugging information, the last twenty bytes of SOSERRBF are not cleared if an executor error message is being printed. If a type 4 error message is encountered, the HOPEFUL entry point in the SOS mainline is transferred to, which then attempts job stream recovery.

The error type codes are:

0: supervisor and machine code loader
1: assembler
2: executor
3: macro generator
4: job abend

Note: R13 need not point to a save area on entry; the caller's registers are saved internally. This is done because several routines call ERRORHND and IOMATIC (which also does its own register saving) and no other routines, and it seemed wasteful to have these routines generate a save area for this sole purpose.

3.2.3 FREEBLOK

Attempts to reclaim the free storage returned by the caller.

CODED BY: R. F. Gurwitz

CALLED BY: MACEXPND

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG SOSCB SOSENTER

On entry, R1 points to the free storage block to be freed and R0 contains the size (in doublewords) of the block. The block is merged into current free storage only if it is at the boundary of the beginning of available free storage. If not, it is merely ignored (the SOS storage management scheme permits this to be done without great disadvantage; see section 2.1).

Specifically, the size in R0 is multiplied by 8 and added to the pointer in R1. If it is equal to SOS@FRCR, the pointer in R1 replaces the value in SOS@FRCR. Otherwise, nothing is done. Error conditions are ignored.

3.2.4 GETBLOK

Satisfies requests for free storage from the free storage pool.

CODED BY: R. F. Gurwitz

CALLED BY: CISWSTOR MACEXPND MACSAVE PASS1 SYMBTAB

ROUTINES CALLED: ERRORHND

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG SOSCB SOSENTER

On entry, R0 contains the size (in doublewords) of the block of storage desired. The SOS@FRCR and SOS@FRND pointers are accessed to see whether there is a block of this size available. If there is, R1 is set to point to it, SOS@FRCR is incremented by 8 times the value in R0, and we return to caller. If not, ERRORHND is called with code 4001, which abends the current job and attempts batch recovery.

Any error, such as a negative value in R0, causes the same call to ERRORHND.

3.2.5 IOMATIC (CMS)

Central I/O processor for the CMS version of SOS. SYSIN, SYSPRINT, SYSUT1, and SYSUT2 are manipulated by this routine. Typing to the terminal and reading the system macro library are performed by ABENDSOS and MACOPEN, respectively.

CODED BY: R. F. Gurwitz and S. H. Gudes

CALLED BY: AMTSETUP DOXREF DUMP ERRORHND EXECUTE HOPEFUL JCLYZER MACEXPND MACSAVE MASHPROC PASS1 PASS2 PHASEI PROCARD SOS SVCALL TRACE

ROUTINES CALLED: ABENDSOS ADTLKP(CMS) ADTNXT(CMS) ERASE(CMS) FINIS(CMS) PRINTR(CMS) RDBUF(CMS) STATE(CMS) WRBUF(CMS)

EXTERNAL REFERENCES: FVS(CMS area) SOSCB

LIBRARY MACROS: ABENDSOS ADT(CMS) CMSREG FCB FVS(CMS) SOSCB

On entry, the caller's registers are saved in the internal save area and addressability is set up. R13 is set to point to the CMS FVS area, which is used to communicate with the CMS I/O routines. The parameters are parsed from the parameter list. These consist of: a fullword pointer to the user's buffer (used in READ, WRITE, NOTE, and POINT); a one-byte function code (0: read; 4: write; 8: close; 12: open; 16: note; 20: point); a one-byte file ID (0: SYSIN; 4: SYSPRINT; 8: SYSUT1; 12: SYSUT2); a one-byte buffer length (used in WRITE to SYSPRINT only); and a one-byte option code (for CLOSE, 0 is normal, 1 is type T; for open, 1 is input, 2 is output, 3 is outin; for READ, 0 is move mode and 1 is locate mode). We then load pointers to the appropriate control blocks (FCBs) and perform the appropriate function (see below). The caller's registers are restored and we return to caller.

SOSOPEN and SOSEOF in SOSCB, along with internal data areas, are used to remember the status of the files. The various functions which are performed are:

READ
If the file is not open for input, we return with error 16. If EOF has occurred for this file, we return with error 12. If the internal input buffer is empty, we reset the buffer cursor and count of the number of records, and go read the next block. If EOF occurs, we return with error 4. If some other error occurs, we return with error 8. If the file is SYSIN, we could have read a short block, so we check on the number of records and adjust the count accordingly. We now point to the next record in the block. If the request was for locate mode, we set the caller's R1 to point to the buffer; otherwise we move the record to the area pointed at by the caller's first parameter. The buffer counter and cursor are updated for the next call.

WRITE

If the file is not open for output, we return with error 20. We then take two courses of action, depending on whether the file is SYSPRINT or SYSUT2.

SYSUT1; The record pointed at by the user's first parameter is moved into the next available location in the buffer. If this fills the buffer (10 records), it is written to disk and the buffer counter and cursor are reset. Otherwise, the cursor and counter are incremented by 80 and 1, respectively.

SYSPRINT: The carriage control character in column 1 of the user's buffer (pointed at by the first parameter) is examined to determine whether it is one of the ANSI control characters "+-01 ". If it is not, the first character is set to a blank. If it is one of " -+0", the number of lines is subtracted from SOSPRTCT. If this goes zero or negative, or if "1" was specified as the carriage control, a skip to a new page is instituted: SOSPRTCT is set to the value of SOSLINCT; if we are in phase I, the page number is incremented and edited into the header line; if we are in phase 0, we print only SOSHDR; in the other phases we print SOSHDPR1 and SOSHDPR2 as well (SOSHDR has a "1" in column 1 to skip to a new physical page); the carriage control character is then made a blank. The line is printed, and the original carriage control character is restored, if it had to be changed. To print a line, a call is made to either WRBUF (if output is to a listing file) or to PRINTR (if it is directly to the virtual printer). An error from WRBUF returns an error code of 8 to the caller; an error from PRINTR causes ABENDSOS to be invoked with code 11.

CLOSE

If the file is open for output, any partial buffer is written out. The SOSOPEN flag for the file is set to indicate that the file is closed, and the CMS FINIS routine is called to close the file. If the TYPE=T option was specified on the CLOSE command, the IOMATIC open function is called with the correct argument to open the file for input.

OPEN for INPUT

If the file is already open, we return with error 24. The filename field of the FCB is set to the value of SOSFILNM. The CMS STATE routine is called to see if the file can be found on a logged-in disk. If not, error 28 is returned.

If the file is SYSIN, we set the open flag for SYSUT1 also. If the FSTB for SYSIN does not indicate fixed length 80-byte records, ABENDSOS is invoked with arguments 13 or 12, respectively. SOSMODE is set to the file mode contained in the FSTB, and the number of items to read is set to 10 (the blocking factor).

The initial record no. in the FCB is set to one, the buffer count is set to force a physical read with the next logical read, the buffer size is set to 800, the FCB filemode is set from SOSMODE, the EOF flag for the file is turned off, and a buffer is allocated in the free storage area and management pointers are set to point to the buffer.

OPEN for OUTPUT (and OUTIN)

If the file is already open, we return with error 24. The FCB filename and filemode are set from SOSFILNM and SOSMODE, respectively. If we have not determined whether there is a writable disk logged in and we need a writable disk (SYSPRINT going to a listing file or assembler job requiring SYSUT2), we look for one by chasing down the ADTs until we find one with its R/W flag on. The order of search is as follows: the disk containing SYSIN is checked to see whether it is R/W; if so, we use that disk. We then check to see whether the disk containing SYSIN is an R/O extension of an R/W disk; if so, we use the mode of the R/W disk. Otherwise, we search for an R/W disk using the standard order of search. If one cannot be found, ABENDSOS is invoked with error 21. If one can be found, the FCB filemode and SOSMODE are set to it, and we set a flag to remember that we've gone through the search.

The file is erased if it exists. The open flag for the file is set to indicate open for output; the buffer size is set to 800; the file pointer is set to one; the blocking factor is set to one (SOS does its own output blocking); a buffer is allocated and management pointers are set to this buffer.

NOTE

The record number of the current record is calculated (the FCB number is the start of the next block of records; thus the actual number is the FCB number less the number of records not read from the current buffer). This number is stored into the area provided by the user.

POINT

The item number in the user's area is moved into the FCB item number, and the block count is set to force a physical read on the next logical read.

3.2.6 IOMATIC (OS)

Central I/O processor for the OS version of SOS. SYSIN, SYSPRINT, SYSUT1, and SYSUT2 are manipulated by this routine. Reading the system macro library is performed by MACOPEN; write to programmer by ABENDSOS.

CODED BY: S. H. Gudes

CALLED BY: AMTSETUP DOXREF DUMP ERRORHND EXECUTE HOPEFUL JCLYZER MACEXPND MACSAVE MASHPROC PASS1 PASS2 PHASEI PROCARD SOS SVCALL TRACE

ROUTINES CALLED: ABENDSOS CHECK(OS) CLOSE(OS) NOTE(OS) OPEN(OS) POINT(OS) READ(OS) SYNADAF(OS) SYNADRLS(OS) WRITE(OS) WTO(OS)

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: ABENDSOS CHECK(OS) CLOSE(OS) CMSREG DCB(OS) DCBD(OS) NOTE(OS) OPEN(OS) POINT(OS) READ(OS) SOSCB SYNADAF(OS) SYNADRLS(OS) WRITE(OS) WTO(OS)

On entry, R1 should point to a parameter list as described under the CMS version of this routine. After establishing addressability and parsing and loading the passed parameters, the function is cased on and the appropriate operation is performed. After performing the operation, the caller's registers are re-loaded and we return with the appropriate return code (described in the individual function descriptions).

OPEN for INPUT

If the file is already open, we return with error 16. The flag byte associated with the file is cleared (aside from the buffer allocation flag), and we physically open the file for input by calling OS OPEN. If the file did not open properly, we return with error 16. If the file is not lrecl 80, recfm FB, and blocksize a multiple of 80, we invoke ABENDSOS with codes 12, 13, or 15, respectively. The blocksize is divided by 80 and this blocking factor is stored in the file information vector. The count field is set to force a physical read on the next logical read. A buffer is allocated from the beginning of the current start of the free storage area, and the free storage management cursor is incrmented past the buffer. SOSOPEN is set to reflect that this file is open. The "input allowed" flag in the file information vector is turned on. If the file is SYSIN, and SYSIN is on a direct access device (determined by a parameter passed to SOS): the SYSUT1 open flag is turned on; the SYSUT1 EOF flag is turned off; the SYSUT1 DCB and file information vector pointers are set to point to those for SYSIN; and the short block count is set to the blocking factor.

OPEN for OUTPUT

If the file is already open, we return with error 16. The file is physically opened by calling OS OPEN. If it could not be opened, we return with error 16. The open flag (SOSOPEN) for the file is turned on, and the "output allowed" flag in the file information vector is turned on also.

OPEN for OUTIN

If the file is already open, we return with error 16. The file is physically opened for OUTIN by calling OS OPEN. If the open fails, we return with error 16. If a buffer has not yet been allocated for the file (indicated by the appropriate flag in the file information vector), the current free storage origin pointer is used to indicate the start of the buffer, and is then incremented past the buffer, along with the other appropriate storage management pointers (SOS@FROR, SOS@FRTP, SOS@FRCR). The count field of the file information vector is set to zero to indicate an empty buffer, the buffer cursor is set, the open flag (SOSOPEN) for the file is turned on, the "buffer allocated" and "output allowed" flags in the file information vector are turned on.

READ

If the file information vector flag for "input allowed" is not on, we return with error 20. If EOF has occurred for this file (checked via SOSEOF), we return with error 12. If the current buffer is empty, we point to the buffer and do a physical read via OS READ. If EOF occurs, we set SOSEOF for the file and return with error 4. The blocksize and residual count are then used to determine whether we read a short record, and the buffer count is set accordingly (to the number of records <u>not</u> read).

The pointer to the next record in the block is accessed. The buffer count is incremented by one, and the buffer record cursor by 80. If locate mode was specified, the record pointer replaces R1 in the caller's save area; otherwise the record is moved into the buffer specified by the user. If the file is SYSIN, SYSIN is on a sequential device, and we are in write-through mode (SOSWRRD is on), parameters are set for writing the record just read onto SYSUT1, and we invoke the code to do this. Otherwise we return.

WRITE

If the "output allowed" flag in the file information vector is not on, we return with error 20. There are two cases: the file is SYSPRINT (a VBA file) or the file is SYSUT1 or SYSUT2 (card image files). These are handled as follows:

SYSPRINT: The first character of the user's buffer is examined to see whether it is one of the ANSI control characters " 01-+". If so, the appropriate number of lines to skip is determined; if not the first character of the line is made a blank. The number of lines to print is subtracted from the number of lines left on the page. If this is non-positive, or if "1" was specified as the carriage control, a page eject is performed as follows: SOSPRTCT is set to the value of SOSLINCT; if we are in phase I, the page number is edited into the header and then incremented; if we are in phase 0, we print SOSHDR; otherwise, we print SOSHDR, SOSHDPR1, and SOSHDPR2 (SOSHDR contains a "1" in column 1 to do the actual skipping); the user carriage control is then set to a blank to avoid superfluous skips. The line is then printed by saving the eight bytes preceding the buffer, setting those eight bytes to the block descriptor and the record descriptor, and doing an OS WRITE. The eight bytes are then restored. The buffer's carriage control is then restored, if it had to be changed.

SYSUT1 and SYSUT2: If the buffer is full, we write it out as follows: an OS WRITE is done to do the physical writing; the buffer cursor is reset to the top of the buffer; the count is set to zero (buffer empty); and the "buffer written" flag in the file information vector is turned on. The buffer count is incremented by one, the record to be written is moved into the buffer at the location specified by the buffer cursor, the buffer cursor is incremented by 80, and we return.

CLOSE

If it is a regular close, we close the file via OS CLOSE and turn off the open flag (SOSOPEN) for the file. If it is a type T close, we check to see whether a buffer has been written. If so, we write the current block out (as a short block, if necessary), and, since it will be read in later, we set the file information vector short block field to the blocksize. Otherwise, we reset the buffer cursor to point to the top of the block and set the short block field to the current value of the count field. In either case, the file information vector flag for "output allowed" is turned off, the one for "input allowed" is turned on, an OS CLOSE with REREAD and TYPE=T is performed, and the EOF flag for the file (SOSEOF) is turned off.

NOTE

The current position of the file is obtained via OS NOTE. If the file is SYSUT1, we add one to the returned note information. This is due to the fact that we note SYSUT1 when we are writing to it. The record noted is the one in the current (in-core) buffer, whereas the information returned by OS NOTE refers to the previous record written to disk. In order to point to the correct block later on, we force the OS NOTE information to point to the next block. If the file is SYSIN, however, we are reading from it, and the block we will wish to point to later is exactly the same as the one we have now, therefore the information is not adjusted. In either case, the block count from the file information vector is stored in the caller's buffer area as well.

POINT

The short block count in the file information vector, in conjunction with the block count in the caller's buffer, is used to determine the number of records that can be expected in the block that we are pointing to. The passed count field is then used to determine the buffer cursor location of the record within the block to which we are pointing. If the file has been physically written to (indicated by the appropriate flag in the file information vector), or if the file is SYSIN and it is on a direct access device, we point to the appropriate block on disk via OS POINT, read the record into core via OS READ, and use the residual count to adjust the buffer cursor.

3.2.7 ISITJCL

Passed a card image, returns a code indicating whether the card is a /DATA, /END, /SOS, or not an SOS JCL statement.

CODED BY: S. H. Gudes

CALLED BY: JCLYZER MACEXPND MACOPEN MACSAVE MASHPROC PASS1 PHASEI SOS SVCALL

LIBRARY MACROS: CMSREG

On entry, R1 should point to the card image to be tested.

The card image is tested to see whether it is a /SOS, /END, or /DATA card. If it is, R15 is set to 16, 12, or 8, respectively. If it is not, R15 is set to zero. An LTR R15,R15 is performed and we return to caller.

Since this routine is called quite often (at least once for each statement read in), it is optimized for time as follows: no register saving is performed (R15 is used as the base register); the first character of the passed card image is tested to see whether it is a slash; if it is not, zero is returned in R15 immediately, without testing the rest of the card image. Thus if a statement is not JCL, this routine will execute only four instructions, including the BR R14.

Since this is the only routine that tests JCL, it is possible to change the definition of SOS JCL by merely changing this routine to test for the new JCL. Note, however, that if the length of the /SOS JCL statement is changed (e.g., $START), JCLYZER must be changed so that it starts its option list scan at the right place. This involves changing the LA instruction at the start of its scan loop to point to the appropriate location.

3.3 Assembler

3.3.1 BINSERCH

Does a binary search through a passed tree, looking for a passed string.

CODED BY: P. J. Relson

CALLED BY: DOPASS1 SYMBTAB

On entry, R1 should point to the top of the tree to be searched through. The nodes of the tree consist of an eight byte character string, followed by a left and a right pointer (needless to say, the character string should be aligned on a fullword boundary). Other information may follow the right pointer, but will not be looked at by this routine.

R3 should point to an eight byte character string (not neccessarily aligned), the string to be searched for in the tree.

Null pointers are indicated by a fullword of zeroes; the tree itself (i.e., R1) must not be null.

If the string pointed at by R3 is found in the tree, R1 is set to point to that entry and we return to caller. If it is not found, R1 is set to point to the node prior to where the string would have been, and we return.

Before returning to the caller, R15 is set to zero if the string was not found in the tree, or to four if it was. An LTR R15,R15 is done before returning.

3.3.2 DOPASS1

Does the first pass scan of the assembler. The label field (if any) is peeled off and added to the symbol table. The opcode and operand fields are scanned and a CISW for the card image is set up and stored.

CODED BY: P. J. Relson

CALLED BY: MACEXPND PASS1

ROUTINES CALLED: BINSERCH CISWSTOR EVAL MACEXPND MACSAVE SYMBTAB

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CALL CMSREG MCB SOSCB

On entry, SOSCOMIN is expected to contain the card image to be processed. If the statement is a comment, the CISW is stored. If it is not, the opcode is tested to see whether it is one recognized by the assembler or whether it is a macro name. If it is either, it is processed as described below. If neither, the location counter is incremented by one to make room for an abend-forcing statement (appropriate action is taken if wraparound occurs) and the label, if there is one, is added to the symbol table. The error CISW is then stored. In any case, we then return to caller. An error in a statement may be detected in this routine or may be passed by the caller, specifically by MACEXPND in some instances.

Different statement types are processed in different ways; these are as follows:

SPACE
TITLE
EJECT
The statement is tested to see whether it has a label. If so, the second non-fatal error code is set to indicate this. The CISW is then stored.

MACRO
If it is not allowed (i.e., if something other than a SPACE, TITLE, EJECT, or comment statement (or another macro definition) preceded it), the CISW fatal error bit and code are set and the CISW is stored. Otherwise MACSAVE is called to save the definition (MACSAVE reads sequentially from SYSIN until it hits the MEND statement, and stores CISWs for all statements in the definition, including the MACRO statement).

macro invocation
The MACEXPND parameter list is set up (pointer to invocation statement, pointer to MCB, offsets to operand and opcode) and MACEXPND is called to expand the call. MACEXPND stores the CISW for the invocation and its continuation statement (if any). The expanded statements are returned to DOPASS1 to be processed

themselves (this is done recursively between the two routines; see section 2.5 for more information).

END

The appropriate return code is set. If there is a label, the non-fatal error code is set for "label ignored."

CCW

machine instruction

If wraparound has occurred, various flags and counts are set and reset. The symbol table parameter list is set and the LC is incremented. If the statement is a CCW, the LC is incremented again and we check for wraparound. The opcode is moved into the CISW, the label (if any) is sent to the symbol table routine, and the CISW is stored.

ORG

If there is a label, the non-fatal error code is set. The operand field is located and EVAL is called to process it. If there is an error (or no operand at all), the CISW error flag and code are set. Otherwise, the literal pool starting address is readjusted and the LC is set to the value of the argument.

DS

If there is a label, the symbol table routine is called to process it, with the appropriate non-fatal error code being set if there is an error. The operand field is isolated and EVAL is called to evaluate it. If there is an error (or if it is missing), the appropriate CISW error flag and code are set. Otherwise, the operand value is added to the current value of the location counter. If this exceeds the bounds of SOS core, the appropriate error flag and code are set and the LC remains unchanged. Otherwise the LC retains its new value. The CISW is then stored.

EQU

If there is a missing or invalid label on the statement, an error CISW is set and stored. The operand field is scanned and evaluated by calling EVAL. If the operand is missing or invalid, an error CISW is set up and stored. SYMBTAB is called to add the symbol to the symbol table. If SYMBTAB returns an error (e.g., the symbol is already in the table), an error CISW is set and stored, otherwise an EQU CISW is set and stored.

DC

If wraparound has occurred, appropriate flags and warning codes are set, and the LC is adjusted. If there is a label, SYMBTAB is called to process it, with the appropriate non-fatal error code being set in case of error. The operand field is then located (if there is none, an error CISW is set and stored) and the column start is saved for pass 2. If the DC is not a C-constant, we are done. If it is, the number of words that the constant will take up is calculated (taking double pops and padding into account), and the LC is incremented by that amount. A missing closing quote causes an error CISW to be stored and the LC to remain unchanged.

3.3.3 DOXREF

Prints the assembly symbol table as an alphabetically ordered cross reference table.

CODED BY: P. J. Relson

CALLED BY: PASS2

ROUTINES CALLED: IOMATIC

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG IOMATIC SOSCB SOSENTER

If there are any symbols in the main symbol table (indicated by the SOSSYMBS flag), headers are set for the cross reference, the pointer to the top of the tree (SOS@SYMB) is loaded, and a postfix walk through the tree is performed, each node being printed as defined by a postfix walk, so that the final cross reference is in alphabetical order (the tree is set up as a standard binary tree).

Each node of the tree is as described in section 4.9.

To print a symbol, the symbol is moved into the output buffer, followed by its value (in character form), the statement at which it was defined (in character form), and finally all references to the symbol (if there are more references than there is room for on the line, the references are continued on successive lines, indented to indicate that they are part of the previous line). The format of a cross reference block is given in section 4.11.

In order to avoid using auxiliary storage, the pointers in the tree are modified to point back to where to return to as we chase down the tree. Thus the printing of the cross reference table <u>will destroy the tree</u>. Also, in order to indicate whether a node has been printed or not, the high order byte of the left pointer is used as a flag. It is therefore necessary that this byte always be zero for every node of the table on entry to this routine.

3.3.4 EVAL

Evaluates a string consisting of sums and differences of symbols, hex constants, decimal constants, and the location counter.

CODED BY: P. J. Relson

CALLED BY: DOPASS1 EVALUATE PROCARD

ROUTINES CALLED: SYMBTAB

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG SOSCB SOSENTER

On entry, R1 should point to a (fullword aligned) parameter list, set up as follows: a fullword pointer to the start of the string to be evaluated, a fullword which will contain the result of the evaluation, a fullword pointing to a byte which will be set to the error code (if there is an error in the string), and finally a byte which will contain the last character of the string (i.e. the terminating delimiter, such as a comma or parenthesis).

The string may be an expression consisting of hexadecimal constants (in X' ' notation), address constants (symbols), character constants (A' ' notation), decimal constants (a decimal integer), or the location counter (*). These may be added to or subtracted from each other, with an optional unary plus or minus preceding the whole string. The string is assumed to terminate at the first delimiter, a delimiter being defined as one of ")(,".

After checking for an initial sign and setting the appropriate indicator, a loop is entered. The first two characters are tested to see whether they are "X'", "A'", "*", greater than or equal to "0", or anything else, and the hex, address, lc, decimal number, or symbol is then processed (error checking for invalid digits, etc.). Once the operand has been evaluated and added to or subtracted from the previous one (we are now pointing at the character following the operand), the next character is checked to see whether it is a plus or a minus sign. If it is, we increment our cursor and repeat the loop. Otherwise we set the current character as the terminating delimiter, set the passed pointer to the terminating character, and return to caller.

Overflow is ignored in performing the arithmetic operations, but the values of the operands have to fall within 32 bits worth of information (two's complement for decimal numbers, absolute for hex and A-constants).

3.3.5 EVALUATE

Checks for indirecting and literals, then calls EVAL to evaluate the operand field.

CODED BY: P. J. Relson

CALLED BY: PROCARD

ROUTINES CALLED: EVAL

EXTERNAL REFERENCES: PASS2CB SOSCB

LIBRARY MACROS: CMSREG PASS2CB SOSCB SOSENTER

On entry, R1 should point to a thirteen byte parameter list, fullword aligned, consisting of: a fullword pointer to the string, a fullword which will be set to the value of the string, a fullword pointer to a byte which will be set to the error code (if any), and a byte which will contain the terminating delimiter of the string.

The first character of the string is checked. If it is a ">", the given string pointer is incremented by one, the indirecting flag in PASS2CB is turned on, and EVAL is called to evaluate the rest of the string (i.e. to perform the arithmetic). If it starts with an "=" sign, it is processed as a literal. Otherwise, EVAL is called with the string itself.

A literal is processed in one of two ways. If it does not begin with "C'", the passed string pointer is incremented by one and EVAL is called to evaluate the expression. If it does begin with "C'", the string is moved into a buffer and padded with blanks to make a multiple of four bytes.

Once the literal has been evaluated, it is searched for in the current literal pool. If it is found, its address (in the literal pool) is placed into the result field of the parameter list. If it is not found, it is moved into the literal pool, the literal pool management pointers are updated, and the address of the new literal is placed into the result field of the parameter list. In either case, error checking consists of whether there is enough room in SOS core for the new literal, whether the literal terminated correctly (single pop for a character string, blank for an expression). (For a character string, the ending character in the parameter list is set to blank, since EVAL is not called to do the evaluation.)

Returning to the caller, R15 is set to zero if there was an error and to four if the string evaluated successfully. An LTR R15,R15 is done before returning.

3.3.6 PASS1

Reads the assembler program. Processes JCL and EOF, otherwise passes the statement to DOPASS1 for parsing and CISW setup.

CODED BY: P. J. Relson

CALLED BY: PHASEI

ROUTINES CALLED: DOPASS1 GETBLOK IOMATIC SYMBTAB

EXTERNAL REFERENCES: NOMACRO(DOPASS1) SOSCB WRAPARND(DOPASS1)

LIBRARY MACROS: CALL CMSREG SOSCB

On entry, the first statement of the assembler program is assumed to be in SOSCOMIN.

After clearing various flags, allocating the first CISW block, and initializing the symbol table, the assembly loop is entered.

If the current statement is JCL, the appropriate return code for the type of JCL is returned to the caller. Otherwise, a "virgin" CISW is set up. If the statement is not a comment, the label and opcode fields are parsed off. In either case, DOPASS1 is then called to do the actual processing. If the statement was an END pseudo-op or if a fatal error occurred, we return to the caller appropriately. Otherwise we read the next card into SOSCOMIN and repeat the processing.

Before returning to the caller, the largest value of the location counter is set as the beginning of the literal pool so that PASS2 can generate literals without fear of over-writing the user's program.

3.3.7 PASS2

Chases down the CISW blocks, reading the statements back in, processing them appropriately, and finally printing the cross-reference if desired.

CODED BY: P. J. Relson

CALLED BY: PHASEI

ROUTINES CALLED: DOXREF ERRORHND IOMATIC PROCARD SYMBTAB

EXTERNAL REFERENCES: PASS2CB SOSCB

LIBRARY MACROS: CALL CMSREG IOMATIC PASS2CB SOSCB SOSENTER

After setting up parameter lists and output headers, the assembly loop is entered.

The next CISW is fetched (if there are no more, we exit the loop). If the statement it referred to was macro generated, it is read from SYSUT2; if it was from SYSIN, it is read from SYSUT1. The statement number and macro generated indicated (if any) are placed into the output buffer. If there were any fatal errors, the LC is put into the listing buffer and the statement is marked as an error. Otherwise, PROCARD is called to evaluate the operand field and generate the code. The statement and any error messages are then printed (if necessary, an abend-forcing instruction is generated and put into SOS core). This continues until there are no more statements to process.

If there was no END statement, an appropriate error message is printed. If the print and listing flags are on, and there are literals, the literal pool is then printed. If the XREF flag was on, the symbolic register XREF blocks are merged with the XREF table and DOXREF is called to print the XREF table. The assembler error count is stored in SOSCB and we return to caller.

Since the number of CISWs must correspond exactly to the number of source and macro generated statements, a non-zero return code from IOMATIC causes control to transfer (via ERRORHND) to HOPEFUL in the SOS module.

3.3.8 PROCARD

Using the CISW for the current statement, the appropriate code is generated (or pseudo-op action is performed) and a listing line is generated, if desired.

CODED BY: P. J. Relson

CALLED BY: PASS2

ROUTINES CALLED: EVAL EVALUATE IOMATIC SYMBTAB

EXTERNAL REFERENCES: PASS2CB SOSCB

LIBRARY MACROS: CALL CMSREG IOMATIC PASS2CB SOSCB SOSENTER

On entry, R4 is expected to point to the card image to be processed, R5 is expected to point to the CISW, and R7 is expected to contain the current value of the location counter. After clearing out the appropriate fields in the Pass 2 Control Block (section 4.6), the statement type is fetched from the CISW and cased on. We then return to caller.

The various cases are as follows:

PRINT
The operand field is scanned for an argument. If it is "ON", SOSPRTON is set to X'0F'; if "OFF" SOSPRTON is set to X'7F'; if "GEN" SOSPRTGN is set to X'FF'; if "NOGEN" SOSPRTGN is set to X'00'. If the operand is missing or invalid, the error code in PASS2CB is set to 13. We then return to caller.

EJECT
The don't-print-this-card flag in PASS2CB is turned on and SOSPRTCT is set to zero to force a page eject on the next write to SYSPRINT.

macro continuation statement
The statement number is decremented by one since this statement counts as part of the previous, and the statement number in the output buffer is cleared to all blanks.

Macro invocation statement
The current value of the location counter is placed into the output buffer.

SPACE
If there is no operand, the spacing counter is set to one; otherwise the two-digit operand is converted to a number using Horner's method (if there is a bad digit or the number is too long, the PASS2CB error code is set to 14 and we return). The don't-print-this-card flag in PASS2CB is turned on. We return to caller if NOLIST was specified on the /SOS card or if PRINT OFF is in effect or if PRINT NOGEN is in effect and the statement was macro generated. Otherwise, we process the spacing. If the number of lines to space is greater than the number of lines left

on the page, a page eject is forced by setting SOSPRTCT to zero. Otherwise the number of lines to space is divided by two; the string "0 " is written the quotient number of times, and the string " " is written the remainder number of times.

TITLE

If there is no operand, or if the first character is not a quote, the PASS2CB error code is set to 10 and we return. Otherwise, the old title is blanked out, the title character counter is set to a max of 65, and we scan the title, moving it character by character into SOSHDR. Special casing is done for double quotes. If there is no closing quote, the PASS2CB error code is set to 11 and we return (the title to this point remains in SOSHDR). Otherwise, we test to see whether the statement is to be printed (NOLIST not specified, PRINT ON in effect, PRINT GEN in effect if statement is macro generated). If it is, a page eject is forced by setting SOSPRTCT to zero, and the PASS2CB don't-print-this-card flag is turned on.

DS
ORG

The current value of the location counter is placed into the output buffer. If the error flag in the CISW is set, the error code in the CISW is moved into the error code in PASS2CB, and the generate-an-abend flag in PASS2CB is turned off. Otherwise, the argument replaces the location counter, and if the statement is ORG, the new value of the LC is placed into the output buffer.

EQU

If the EQU was bad, the error code in the CISW is moved into the error code in PASS2CB and the prevent-error-handling flag is turned on. Otherwise, the label is put into a SYMBTAB plist, SYMBTAB is called to retrieve the value of the symbol (the symbol must be in the symbol table since pass one put it there). The value is then converted and placed into the output buffer.

END

The we-have-an-END flag in PASS2CB is turned on. The default execution address (SOSSTART) is set into the listing in case the operand is missing or invalid. We scan for the operand field, returning immediately if there is none. Otherwise the operand field is evaluated, and if it is a valid address, it is placed into SOSSTART and the listing buffer. If it is invalid, the appropriate code is set into the PASS2CB error byte.

DC

The location counter is placed into the output buffer. Two courses of action may now be taken, depending on whether the constant is a C-type character string or an expression.

C-type character string

The length of the string is fetched from the CISW. If the length is zero, we merely move in four blanks. If not, the string is scanned, and characters are moved into SOS core one at a time. (Double pops are special cased.) This is done for the number of characters specified in the CISW. The string is then padded with blanks to make it a multiple of four bytes, and the location counter is incremented by the number of words taken up by the string. The first word of the string is then placed into the listing line.

Expression of A-cons, X-cons, symbols, the LC, and decimal constants

EVAL is called to evaluate the expression (if the expression is invalid, EVAL will set the PASS2CB error byte). If the result is bad, we increment the LC by one and return. Otherwise, if the ending delimiter is a blank, we store the evaluated value into SOS core, and display it in the buffer. If it is not a blank, the PASS2CB error byte is set to 3 and we return.

BR

The LC is put into the output buffer. The register field is evaluated, and if it is invalid, the PASS2CB error byte is set and we return. Otherwise, the opcode and register are placed into SOS core (the rest of the instruction is set to zero) and the instruction is placed into the output buffer.

B

The LC is placed into the output buffer. The address field is evaluated, and if it is invalid the PASS2CB error byte is set and we return. Otherwise, the opcode and address are moved into SOS core, the other fields are zeroed, and the final instruction is placed into the output buffer.

AXAI
SXAI

The LC is placed into the output buffer. The register field is evaluated, with PASS2CB's error byte being set if it is invalid. If it is not invalid, there must be a comma following the register field, otherwise we have another error. The immediate field is evaluated (up to the open parentheses or blank). If it is invalid, or less than -32768, or greater than 32767, we return with error. If it did not end with a blank or open paren, we return with error. If it ended with a blank, we move the instruction into core and return. Otherwise, the index register field is evaluated, returning with error if it is invalid. If it ended with a closing parenthesis, the instruction is assembled into SOS core and placed into the listing line. If not, we return with error.

other instructions

The LC is placed into the listing line. The register field is evaluated by calling EVAL. If it is invalid, the PASS2CB error byte is set to an appropriate code and we return. Otherwise, if a comma does not follow the register field, we set the error byte and return. Finally, the address field is evaluated (including any index register), and if it is valid, the instruction is assembled into SOS core and placed into the listing line. If it is invalid, an error return is taken.

CCW

The LC is placed into the listing line. The first field is evaluated (its value must fit into a byte). If it is invalid, an appropriate code is placed into the PASS2CB error byte and we return. Otherwise, if there isn't a comma following the field, we take an error return. The byte is then saved as the opcode of the CCW. The next field is evaluated and if it is invalid or doesn't end with a comma we take an error return. The byte is saved as the status byte. The next field is evaluated, again with an error return if it is invalid or isn't followed by a comma. This byte is saved as the count field. The address field is then evaluated, with an error exit being taken if it is invalid or not followed by a blank. The address is then assembled into the second word of the CCW. If an error occurred, the do-the-error-stuff flag in PASS2CB is turned off, and the doubleword is assembled with an opcode of zero and a return count and count field which is the current error number. In any case, the two words of the CCW are placed into the listing line.

3.3.9 SYMBTAB

Processes requests pertaining to the assembler symbol table, including addition and lookup of symbols, initialization, and final register/symbol tree merging.

CODED BY: P. J. Relson

CALLED BY: DOPASS1 EVAL PASS1 PASS2 PROCARD

ROUTINES CALLED: BINSERCH GETBLOK

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG SOSCB SOSENTER

On entry, R1 points to a 14-byte parameter list as follows: the first 8 bytes are the symbol to be manipulated; the next fullword is the vaule of the symbol (if the symbol is being added) or will be set to the symbol's value (if it is being retrieved); the next byte is X'FF' to retrieve the symbol or 00 to insert it; and finally, a byte which is either x'0F' to indicate that the symbol table is to be initialized, X'FF' to indicate that the final register insertion is to be performed (see text), or 00 to indicate that a symbol is being retrieved or added.

The symbol table routines are divided into two general categories: those that handle the symbol table and cross reference for symbolic register symbols (R0, R1, ..., R15) and those that handle the table and cross reference for the other symbols.

If the operation is a symbol table initialization or a final register insertion, these are performed (see below). Otherwise, the symbol is scanned to see whether it is one of the symbolic registers (in which case it is processed as described below under STREG), whether it is a symbol which starts with "R" and a digit (in which case it is processed as described under STREST, with the symbol table pointer being the pointer to the register tree), or whether is it something else (in which case it is processed as described under STREST, with the symbol table pointer being the main symbol tree (SOS@SYMB)).

We then return to the caller. R15 contains either a 0, which means that an error occurred (a retrieve could not find the symbol or a store found the symbol already in the table), while 4 indicates that no error occurred.

Initial Symbol Table Setup
A copy of the template for the register symbol tree is moved into the register symbol table and SOSSYMBS is turned off (indicating that there are no symbols in the table).

Final Register Insertion
This occurs when the register and general symbol tables are to be merged in order to print the symbol table. STREST is called, which will merely add "R2" (the root node of the register symbol

tree) to the main symbol tree. Since the register tree contains all symbols beginning with "R" and a digit, all entries in the register table will be alphabetical in the main tree by merging that one node.

STREG - Process Register Symbol

This routine handles all stores and retrievals to and from the register symbol table, treating it as a contiguous string of 16 symbol entries rather than as a tree.

Store: If the register has been stored previously, we merely return an error return code. Otherwise, if SOSXREF is on, we turn on SOSSYMBS, allocate an XREF block, link it to the symbol block, and set the statement number into it (i.e., the statement at which it was defined). Regardless of XREF status, the value of the symbol is then placed into the symbol block, the register-in-use flag is turned on, and we return.

Retrieve: If the symbol has not been stored yet, we return an error code. Otherwise, the statement number is placed into the XREF list for the register (if SOSXREF is on), the value is retrieved from the symbol table block, placed into the plist, and we return to caller.

STREST - Processing Other Symbols

This processing section can be divided into several parts:

Store - symbol table is empty: If this is the final register store, the address of the top node in the register tree is moved into SOS@SYMB; if it is a regular store, a symbol table entry is allocated, the fields are filled in from the available information, the daughter pointers are zeroed, and if SOSXREF is on the first cross reference block is allocated, linked to the symbol table entry, and initialized (with the statement where it is defined).

Store - the tree already exists: If the symbol is found in the tree, an error code is returned. Otherwise, if it is the final register store, the appropriate pointer in the main symbol table entry is set to point to the root node of the register binary tree. Otherwise, a symbol entry is allocated and initialized (its cross reference blocks are initialized if SOSXREF is on) and the symbol is added to the main symbol table at the point indicated by the previous binary search.

Retrieve: If the symbol is not found, an error code is returned; otherwise the symbol's value is placed into the plist and we return.

3.4 Macro Generator

3.4.1 AMTSETUP

Sets up an AMT given an MCB and a macro invocation statement.

CODED BY: S. H. Gudes

CALLED BY: MACEXPND

ROUTINES CALLED: IOMATIC

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: IOMATIC MCB SOSCB SOSENTER

On entry, R3 is to point to the MCB for the macro involved, while R4 is to point to the first available parameter location in a previously allocated AMT skeleton.

An AMT skeleton consists of the following (see section 4.1 for more information on AMTs): a fullword containing the total number of parms in the AMT; 12 bytes of &SYSNDX information; 12 bytes of label parameter information, if any; 12 bytes * (#keyword parameters + #positional parameters); the invocation statement, and the invocation continuation, if any. The pointer in R4 is to point to the start of the group of 12 bytes * (#keyword parameters + #positional parameters) area.

The default AMT is set up. This involves making entries for each parameter, setting positional parameters to null (by setting their length fields to X'FF') and setting keyword parameters to their default values (i.e., the value pointers point into the MCB for this macro). At this point the MCB is no longer needed and is discarded.

We return if there is a ", " in the operand field (by definition, this is a null operand field). (It is assumed that whoever set up the AMT skeleton was nice enough to place only the operand field of the invocation statement into it, preceded by an arbitrary number of blanks.)

We now enter a scanning loop. Each parameter is peeled off, a parameter being defined as the string from the current scan position to the next comma or blank (special casing is performed to allow any characters within pops to be accepted). Once the parameter has been peeled off, it is checked for an embedded equal sign (pops being special cased). If there is not one (or if there is one as the first character of the string), the parameter is assumed to be positional and is placed into the next position in the AMT. The positional parameter counter is decremented, and if it is negative, error code 18 is returned.

If the parameter is a keyword parameter, a flag is set so that following positional parameters will be flagged as errors. The keyword is then searched for in the MCB. If it is not found, error 19 is returned. Otherwie, the keyword parameter length and pointer are set to point to the string in the invocation buffer. A flag is then set so that if this parameter is found again, it will be flagged as an error. If the terminating delimiter was not a blank, the above scan is repeated for the next parameter, otherwise the program returns, with the AMT set for use by XPNDSTMT. If the terminating delimiter is ", ", an internal continuation card processor is invoked to reset the scan pointer to the next card, with appropriate checking for a label field. A flag is then set so that another continuation card will be flagged as an error.

3.4.2 CHECK - CKEND, CKMACRO, CKMEND

Tests the passed card image to see whether it is an END/MACRO/MEND statement (depending on the entry point used), and returns an appropriate code.

CODED BY: S. H. Gudes

CALLED BY: MACSAVE

EXTERNAL REFERENCES: TRTFRBL(SOSCB) TRTFRNON(SOSCB)

LIBRARY MACROS: SOSENTER

On entry, R1 should point to the 80 character field to be tested. The label and opcode are parsed off, and the opcode is checked to see whether it is END/MACRO/MEND.

If it is a plain old END/MACRO/MEND, R15 is set to zero. If there is a label in the label field and the opcode is END/MACRO/MEND, R15 is set to one. Otherwise there is no opcode, the opcode is not END/MACRO/MEND, or the statement is a comment, in which case R15 is set to negative one.

3.4.3 MACEXPND

Called by DOPASS1 when it encounters a macro call. The appropriate definition is fetched, an AMT and dynamic control area are set up, the model statements are expanded by calling XPNDSTMT, and the expanded statements are passed to DOPASS1 to be assembled. If a nested call is encountered, MACEXPND calls itself recursively to process the new call.

CODED BY: S. H. Gudes

CALLED BY: DOPASS1 MACEXPND

ROUTINES CALLED: AMTSETUP CISWSTOR CKJCL DOPASS1 FREEBLOK GETBLOK IOMATIC MACEXPND XPNDSTMT

EXTERNAL REFERENCES: NAMELIST(OPSRCH) NUMOPS(OPSRCH) SOSCB

LIBRARY MACROS: CALL IOMATIC MCB SOSCB SOSENTER

On entry, R1 is to point to a five word parameter list. The first word is a pointer to the MCB for the macro; the second points to the invocation buffer; the third contains the length of the label on the statement (0 if none); the fourth contains the offset into the invocation statement which is the first blank following the opcode field; and the fifth is an error code block (see text).

Returning to the caller, R15 contains the number of continuation cards (zero or one) (this is done so that the caller does not have to re-scan the invocation statement in order to determine whether it is continued). This routine stores CISWs for the invocation statement and its continuation, while it is assumed that DOPASS1 will store the CISWs for the generated statements.

Two control blocks are used in macro generation. The Active Macro Table (AMT) equates symbolic parameter names with their values for this particular macro invocations (see section 4.1). It is created from the MCB for the macro. The second control block is the dynamic storage area (DSA), which contains save areas and work buffers used during expansion. This area must be dynamically generated since the macro generator is recursive. Processing of a macro expansion proceeds as follows:

The AMT and DSA pointers are set to null. SOSYSNDX (which contains the last used value of &SYSNDX) is incremented by one. The nesting level indicator is incremented by one. Any non-fatal error code which was determined by the caller is ORed into a fullword which will be the CISW for a continuation card, if any (and hence has its macro bit set accordingly).

If the maximum nesting level is exceeded, we set an error CISW, flush any continuation, and leave. If there was an error in the definition of this macro (i.e., if the error code in the MCB is not zero), an error CISW is stored and we leave. Otherwise, an AMT and DSA are set up (see below), and AMTSETUP is called with the MCB, AMT, and invocation statement to set up the AMT. If AMTSETUP found an error, the CISW is stored and we leave. Otherwise, the CISW for the invocation statement is stored, and if there was any continuation its CISW is stored. The macro is then expanded (see below).

Leaving consists of freeing the AMT and DSA (if they were allocated), moving the dynamic save area information into the static save area, decrementing the nesting level, and returning to caller.

Allocating the AMT and DSA

The DSA is allocated first. If we are at nesting level 1, the first few hundred bytes of SOS core are used as a DSA. Otherwise, it is allocated at the next available location in SOS core, or, if SOS core is full, GETBLOK is called to get the storage. The static save area is moved into the DSA and save areas are re-linked. SOSYSNDX is unpacked into character form, and flags and counters are set. The size of the AMT is then calculated (#keyword parms + #positional parms + 1 + (1 if label parms, 0 if none)) * 12 + 4 (no.-of-entries field) + 160 (invocation statement and continuation). A block of this size is then allocated (in SOS core if there is room, by GETBLOK if not) and the pointer is saved in the DSA. The SYSNDX and label fields are then set into the AMT, as well as the invocation statement and its continuation. The label and opcode fields of the invocation statement within the AMT are then blanked out. Information is set within the DSA in case a recursive call is necessary.

Expanding the Macro Definition

Processing is done within a loop. The next record in the definition is fetched (if there are no more, processing is complete). XPNDSTMT is called with the model statement and the AMT to perform expansion. If the statement is a ".*" comment, no more processing is performed for the statement. Otherwise, the label length, opcode length, and opcode starting position are saved for later processing. The model statement is written to SYSUT2 to be read later by pass 2 of the assembler. If a non-fatal error occurred in expansion, the code is saved for later use. If the statement is a comment, it's CISW is stored and no further processing is done. If a fatal error occurred, the error CISW is stored and no more processing is done with the statement. Otherwise, the opcode is isolated (if there is no opcode, an error CISW is stored and no further processing of this statement takes place). The operand scan start location is saved. If the opcode is "END", "MEND", "MACRO", or the statement is JCL, an error CISW is stored and no more processing is performed. Otherwise, the opcode is tested to see whether it is a macro name or not. If not, DOPASS1 is called with the statement. If so, the MCB pointer is placed into the recursive parameter list in the DSA. The next record in the definition is fetched and expanded in case it is a continuation of the macro call. MACEXPND is then called recursively to expand the definition. On return from MACEXPND, if the statement was not continuation we set the model statement cursor to re-read the previous statement for processing.

3.4.4 MACOPEN (CMS)

Opens the SOS system macro library, scans the directory, reads in the macros, sets up their MCBs and saves them in free storage.

CODED BY: S. H. Gudes

CALLED BY: PHASEI

ROUTINES CALLED: ABENDSOS CKMACRO CKMEND FINIS(CMS) MCBSETUP OPSRCH RDBUF(CMS) STATE(CMS)

EXTERNAL REFERENCES: FVS(CMS area) SOSCB

LIBRARY MACROS: ABENDSOS CALL DSCT FCB FVS(CMS) MCB NUCLEUS RDBUF SETUP SOSCB SOSENTER

On entry, we set R13 to point to the CMS FVS area. If the macro library has already been scanned, we leave, since all the definitions have been saved in free storage. Otherwise, we open the file (SOSMLIB MACLIB), checking to make sure that it is in card image form. The first record is read, and if the first six characters are not "MACLIB" (which they must be by convention), we set the macro error flag, close the file, and return. Otherwise, we get the logical record number of the first directory record (the 7th and 8th bytes of the initial record) and the number of bytes in the directory (the 11th and 12th bytes of the initial record). If the directory is empty, there is nothing to do; otherwise we enter a processing loop.

Processing consists of getting each 12-byte entry in the directory, setting up an MCB for the macro in the free storage area, linking the MCB to the previous MCBs, and reading the definition into core.

If the first record of the definition is not a MACRO statement, the MCB error code is set to reflect this and the definition is ignored. The prototype statement is then read, and MCBSETUP is called to process the MCB and set the MCB fields appropriately. If the prototype is valid, the statements of the definition are read into core (except for ".*" comments) and saved. The appropriate free storage management pointers are updated. The directory record cursor is then incremented, a new directory record is read if necessary, and this continues until the directory is exhausted.

Extensive error checking, including checks for EOF and not enough free storage at every point of the game is performed.

3.4.5 MACOPEN (OS)

Opens the SOS system macro library, scans the directory, reads in the macros, sets up their MCBs and saves the definition in free storage.

CODED BY: S. H. Gudes

CALLED BY: PHASEI

ROUTINES CALLED: ABENDSOS CHECK(OS) CKMACRO CKMEND CLOSE(OS) FIND(OS) MCBSETUP NOTE(OS) OPEN(OS) OPSRCH POINT(OS) READ(OS)

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: ABENDSOS CALL CHECK(OS) CLOSE(OS) DCB(OS) DCBD(OS) DSCT FIND(OS) MCB NOTE(OS) OPEN(OS) POINT(OS) READ(OS) SOSCB SOSENTER

If the macro library has been scanned already, we return to caller. If not, the library (ddname SYSLIB) is opened. If the DD statement is missing, we return to caller. The blocksize and record format are check to make sure that it is a card image file, blocking factor of at least 3 and not more than 2048 (this is due to the available buffer size and the scanning algorithm for possible macro continuation cards). If these conditions are satisfied, the PDS directory records are read into a buffer and scanned.

Processing consists of getting each entry in the directory, setting up an MCB for the macro in the free storage area, linking the MCB to the previous MCBs, and reading the definition into core.

If the first record of the definition is not a MACRO statement, the MCB error code is set to reflect this and the definition is ignored. The prototype statement is then read, and MCBSETUP is called to process the MCB and set the MCB fields appropriately. If the prototype is valid, the statements of the definition are read into core (except for ".*" comments) and saved. The appropriate free storage management pointers are updated. The directory record cursor is then incremented (a new directory record is read if necessary), and this continues until the directory is exhausted.

Extensive error checking includes EOF at every stage of the game, invalid prototype, bad directory or member, etc. If there is not enough free storage to hold the definitions, ABENDSOS is invoked with argument 14.

3.4.6 MACSAVE

Invoked by DOPASS1 when it encounters a MACRO statement, this routine sets up an MCB for the macro definition, saves the definition in core, and sends the CISWs for the statements to be stored by CISWSTOR.

CODED BY: S. H. Gudes

CALLED BY: DOPASS1

ROUTINES CALLED: ABENDSOS CISWSTOR CKEND CKJCL CKMACRO CKMEND ERRORHND GETBLOK IOMATIC MCBSETUP OPSRCH

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: ABENDSOS CALL IOMATIC MCB SOSCB SOSENTER

After addressing SOSCB and SOS core, we call CKMACRO to see whether the MACRO statement has a label, and set an appropriate non-fatal error code. The CISW for the MACRO statement is stored. The prototype is then read. If it is JCL, an END statement, or EOF, the appropriate abend is taken. If it is a MEND, the appropriate error code is set, the CISW is stored, and we return to caller. The next card in the input stream is then read and saved.

If there is not enough room in free storage for the MCB, we abend. The new MCB skeleton is allocated and linked into the MCB chain, several fields are set, and MCBSETUP is called to scan the prototype and set up more of the MCB (MCBSETUP will tell us whether the card read in after the prototype is a continuation of the prototype). If there was an error in the prototype or there is a previous definition of the macro, the definition is flushed, an error CISW is stored, and we return to caller. Otherwise, the prototype CISW is stored. If the macro definition is null (i.e., no model statements, a MEND CISW is stored and we return to caller. Otherwise, the model statements are read into core (except ".*" type comments) and saved. The final CISW is then stored, and we return to caller.

The model statements are read into SOS core, unless it is full, in which case they are read into linked lists in the free storage area (this is necessary since CISWSTOR may allocate a new CISW block at any time). They are read into blocks which have a fullword count field followed by that number of 80-byte card images. If the statements are in the free storage area, the count field is one. If both SOS core and the free storage area overflow, the current job is abended.

3.4.7 MCBSETUP

Passed a macro prototype statement (and its continuation, if any) and a free area pointer, this routine parses out the prototype statement, creating an MCB in the free storage area, and returns a pointer to the end of the new MCB to aid in free storage management.

CODED BY: S. H. Gudes

CALLED BY: MACOPEN MACSAVE

ROUTINES CALLED: ABENDSOS

EXTERNAL REFERENCES: SOSCB TRTFRNON(SOSCB) TRTFRPOP(SOSCB)

LIBRARY MACROS: ABENDSOS MCB SOSCB SOSENTER

On entry, R4 should point to the 80-byte prototype (if there is a continuation card, it must be 80 bytes long and immediately follow the prototype). R3 should point to the start of a free storage area where the MCB can be built. It is unknown at entry how big this area should be; the smallest MCB is 32 bytes long, the largest may be as long as 600 bytes. When this routine returns to the caller, R1 will point to the first available byte following the MCB (since the MCB is allocated in fullwords, this will be on a fullword boundary). R15 will contains the number of continuation cards (zero or one), so that the caller need not re-scan the source statement.

The free storage area is checked to see if there is at least 32 bytes following the location pointed at by R3. If not, ABENDSOS is invoked with argument 17. The label field (if any) is then peeled off the prototype and saved. The opcode is parsed off and set into the MCB. A loop is then entered which parses each parameter and sets them into the MCB. Appropriate provisions are made to ensure that a positional parameter is not defined after a keyword parameter, that there are not multiple definitions of the same keyword parameter, that &SYSNDX is not used as a parameter name, etc. Pops are recognized in the default value field of a keyword parameter, and continuation cards are processed if necessary (no I/O is done as the continuation is assumed to follow the prototype in core).

3.4.8 OPSRCH

Passed an MCB (containing at a minimum the MCBMACNM and MCBLIBRY fields), chases down the macro list and opcode list to see whether the macro has been previously defined.

CODED BY: S. H. Gudes

CALLED BY: MACOPEN MACSAVE

LIBRARY MACROS: CMSREG MCB SOSENTER

On entry, R1 should point to an MCB whose MCBMACNM field contains the opcode to be searched for. The opcode is first searched for in a list of opcodes which are defined for the SOS system. If it is found, we return such an indication to the caller. If not, we chase down the MCB chain, looking for a macro with the same name (a macro is a duplicate iff both names match and both macros come from the same place, i.e. library or user-defined). If it is a duplicate, an indication of this is returned to the caller. Otherwise, an indication that the name is unique is returned to the caller.

Returning to the caller: if the macro is a duplicate opcode, R15 is set to one; if it is a duplicate macro, R15 is set to negative one and R1 is set to point to the MCB which has been previously defined; if it is unique, R15 is set to zero.

3.4.9 XPNDSTMT

Passed an AMT and a model statment, this routine performs macro parameter substitution on the statement, using the values in the AMT. Substitution overflow and invalid parameters are checked for.

CODED BY: S. H. Gudes

CALLED BY: MACEXPND

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: SOSCB SOSENTER

All arguments are passed in registers. R1 points to where the expanded statement should be put (72 bytes), R3 points to the model statement to be expanded (72 bytes), and R4 points to the AMT (section 4.1) to be used in the expansion.

On return to the caller, R15 contains the return code (0: successful expansion; 3: truncation to 72 characters during expansion; 25: invalid symbolic parameter; 26: undefined symbolic parameter). R0 is set to the length of the label (zero if none) and R1 is set so that the low order 3 bytes contain the offset to the beginning of the opcode and the high order byte contains the length of the opcode (zero if none). R0 and R1 <u>are</u> <u>not</u> <u>set</u> if the statement is a comment (either standard or macro).

An internal buffer of 144 bytes is used to perform expansion since an initial very long parameter can be offset by several trailing null or short parameters. After the passed model statement is moved into the buffer, a loop is entered until the end of the statement is reached.

An ampersand is scanned for using a TRT. If one is found and an ampersand follows, the entire statement is moved back one character (to get a single ampersand) and the card cursor is set to point after the ampersand. Otherwise, the ending delimiter of the symbolic parameter is found, and the parameter is tested to see if it is invalid (contains non-alphamerics). If it is, we return to caller with error code 25. The AMT table is then searched for the parmeter. If it is not found, we return to caller with error code 26. If the delimiter of the parameter was a period (special macro concatenation symbol), the length of the parameter (as sent to the internal substitution routine) is incremented by one, so that no actual data movement need be done at this point. The substitution routine is then called to substitute the actual value of the parameter for the parameter itself. The ampersand scan resumes past the substitution point, continuing until no more ampersands are found on the statement.

Substitution involves determining whether the parameter is shorter than, longer than, or the same size as, the value, spreading the model statement (or squeezing it, as appropriate), and moving the value into the appropriate position. If the value of a parameter is the null string, only squeezing will be performed.

If the statement is not a comment, the label, opcode, and operand fields are parsed out, the opcode is moved to column ten and the operand to column 16. The statement is tested to see whether it expanded past column 72 (with the appropriate return code being set), the statement is moved to wherever the caller asked that it be moved, R0 and R1 are set (if it is not a comment), and we return to caller.

3.5 Executor

3.5.1 ABNDTRCE

Prints the last ten instructions and last ten branches executed.

CODED BY: S. H. Gudes

CALLED BY: EXECUTE

ROUTINES CALLED: FORMTRCE IOMATIC

EXTERNAL REFERENCES: BRBLKS(TRACE) FIRSTL(FORMTRCE) SOSCB TRBLKS(TRACE)

LIBRARY MACROS: BRANCH CALL CMSREG IOMATIC SOSCB SOSENTER TRACEBLK

The external TRBLKS control area (in the TRACE csect) is accessed to see whether there are any trace instructions to be printed. If there are, headings are printed and FORMTRCE is called successively to print each instruction in the order in which they occurred. It is possible that fewer than ten instructions can be printed. If there are no instructions to print, headings are not generated.

The BRBLKS control area (also in the TRACE csect) is then accessed to see whether there are any branch instructions to print. If so, headings are printed and the internal FORMBRCH routine is called for each branch instruction to be printed. The branch instructions are printed in the order in which they occurred. It is possible that fewer than ten lines may be printed. If there are no branch instructions to trace, nothing is printed.

If anything was printed, a blank line is printed to space out the listing. We then return to caller.

The internal FORMBRCH routine puts the disassembled instruction into the output buffer. If an index register was specified and the instruction is not BR, the index register contents are put into the buffer. If the instruction is not a B, the register contents are put into the buffer. If indirecting was used and the instruction is not a BR, the indirecting level is put into the buffer. Finally, the new PC value (that is, the value of the PC after the branch was executed) is put into the buffer.

3.5.2 DUMP

Prints out selected portions of SOS core, 8 words to the line.

CODED BY: S. H. Gudes

CALLED BY: EXECUTE SVCALL

ROUTINES CALLED: IOMATIC

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG IOMATIC SOSCB SOSENTER

On entry, R1 points to a two-fullword argument. The first word is the dump starting address, and the second is the dump ending address. The first word must be less than or equal to the second, and both must be in the range 0 to X'FFF'. If either of these conditions are not met, SOSABEND in SOSCB is set to seven and we return to caller.

Once the arguments have been loaded, the 360 core address of the appropriate SOS core locations is calculated, loop counters and pointers are set so that the initial word is indented properly into the line, and a branching loop is entered. Note that this loop is entered in the middle of the body of code. This was necessary to avoid duplication of code, and although not the clearest of all algorithms, it is localized to this routine.

Inside the loop, the EBCDIC equivalent of the core is moved into the buffer and unprintables are translated to periods. The starting location for each line is put into the buffer, followed by a colon. A loop is then entered which converts each word on the line to EBCDIC and puts the numbers into the buffer at the appropriate position. Once the line is full (or all the requested words have been dumped), the loop exits and the line is printed. A loop is then entered which checks succeeding lines until it finds one which is not the same as the line just printed. If lines are skipped over, the message "**SAME AS ABOVE**" is printed to the listing (this message is printed only once, no matter how many lines are skipped over). The outer loop then proceeds to process the next line, until all the requested words have been dumped.

Special provisions are made to not print the trailing part of a line of dump if those words were not requested, and also to print the last line of the dump even if it is the same as a previous line.

A word of warning: this code is very compact and will take a bit of hand simulation before its inner workings are unveiled.

3.5.3 EFFECT

Performs effective address calculation, including processing of indirecting and setting common information in TRACEBLK.

CODED BY: C. C. Gallagher

CALLED BY: SINEXEC SVCALL

EXTERNAL REFERENCES: SOSCB TRACEBLK

LIBRARY MACROS: CMSREG SOSCB SOSENTER TRACEBLK

On entry, R5 is to contain the register and index numbers in the high order byte and the loc value in the next two bytes, the low order bytes being ignored (basically the instruction shifted left 8 bits).

The register is parsed, its contents are fetched from SOS core, and then saved in TRACEBLK as the old register contents.

The index register is parsed. If it is not zero, the register is fetched and saved for the trace. The loc field is then isolated and added to the index register value (or zero if it was not specified). The current level of indirecting is stored. The indirecting flag is accessed; if it is on, the word at the effective address is fetched, shifted left by 12 bits, the indirecting counter is incremented by one, and we repeat the above.

If more indirects occurred than allowed by SOSINDLV, SOSABEND is set to 9. If the effective address was outside the range of SOS core and the instruction was not a shift, SOSABEND is set to 8. If everything went A-OK, the old location contents are moved into the trace block, as is the effective address itself. We then return to caller.

3.5.4 EXECUTE

After initialization, a fetch- increment- execute loop is entered and the user program is interpreted until a halt instruction or error occurs. Halt, AXAI, SXAI, BR, and invalid opcodes are handled by internal routines, while SVCALL is called to handle SVC and SINEXEC is called to handle all other instructions. After execution, a dump is printed (if required) and the instruction count is printed.

CODED BY: C. C. Gallagher

CALLED BY: SOS

ROUTINES CALLED: ABNDTRCE DUMP ERRORHND IOMATIC SINEXEC SVCALL TRACE

EXTERNAL REFERENCES: FIRSTL(FORMTRCE) SOSCB TRACEBLK

LIBRARY MACROS: CMSREG IOMATIC SOSCB SOSENTER SPIE(CMS/OS) TRACEBLK

This is the supervisor for phase II of the SOS system.

On entry, a SPIE is set up to trap a divide exception (see the divide instruction in SINEXEC). Headings are set up. If there was user data (determined by phase I), IOMATIC is called to point SYSUT1 to the start of that data. The tracing facilities are initialized. SOSSTART is fetched to find the address of the first instruction to be executed. The processing loop is then entered until an abend or a halt instruction is encountered.

Processing consists of: checking the current value of the program counter to make certain that it is within the bounds of SOS core (if not we set SOSABEND to 8 and exit the loop). The instruction is fetched and saved for the trace, as is the program counter. The program counter is incremented to point to the next instruction to be executed. The opcode of the current instruction is parsed off and used to case into a table which separates instructions into six categories and invokes the appropriate routine for the category (these are listed below). If an error occurred during execution, we exit the loop. If a warning occurred, we print the warning message and call ABNDTRCE to print the last ten instructions and branch instructions executed. We then repeat the above as long as the instruction count is not exceeded (an error).

Instructions are broken into six groups: invalid opcodes; AXAI and SXAI; BR; H; SVC; and all others. These are handled by the internal routines INVALOP, AXAI, BREG, HALT, and the external routines SVCALL and SINEXEC, respectively. The internal routines are described below.

INVALOP: SOSABEND is set to 14.

AXAI: The register number is parsed and the contents of that register are fetched. The contents are saved in TRACEBLK. The index register is parsed, and if it is not zero the index register contents are added to (AXAI) or subtracted from (SXAI) the register contents. The index register contents are saved in TRACEBLK. The immediate data is then parsed off and added to the register contents (as modified by the xreg contents). The sum is placed into the appropriate SOS register, as well as into TRACEBLK, and TRACE is called to perform user trace functions.

BREG: The register number is parsed. The register contents are loaded into the program counter and saved in TRACEBLK. If the address is out of the range of SOS core, SOSABEND is set to 8; otherwise TRACE is called to perform tracing functions.

HALT: The address at which the halt instruction occurred is moved into a buffer and printed along with a message indicating that a halt instruction occurred.

After the loop has been exited (for whatever reason), the SPIE is freed and output headers are cleared. If the user abended, a message is printed indicating the cause and location, and ABNDTRCE is called to print a trace of the ten instructions and ten branch instructions prior to the abend. If the user asked for a dump (via the DUMP option) or abended, a dump header is set and DUMP is called with a parameter list encompassing all of SOS core. The contents of the user output buffer (SOSPLINE) are then printed. The number of instructions executed by the processing loop is determined, converted to EBCDIC, and printed with an informatory message. We then return to caller.

3.5.5 FORMTRCE

Formats and prints a trace control block.

CODED BY: C. C. Gallagher

CALLED BY: ABNDTRCE TRACE

ROUTINES CALLED: IOMATIC

EXTERNAL REFERENCES: SOSCB

LIBRARY MACROS: CMSREG IOMATIC SOSCB SOSENTER TRACEBLK

On entry, R6 is expected to point to a Trace Control Block (section 4.10) containing the instruction to be traced. The disassembled instruction and the location counter are put into the buffer, and if the instruction has not abended and has not generated any warnings, the internal FILLUP routine is called to fill in the necessary fields of the instruction.

Each instruction has associated with it a control byte containing flags which indicate the information to be printed for the trace of that particular instruction. The value in the CODE field of the Trace Control Block is used to index into this array of flags.

The diferent types of fields are:

Single register instruction
Old register contents may be printed; if old register contents are printed, new register contents may also be printed.

Double register instruction
The old and new contents are always printed.

Index register instruction
In an instruction in which an index register has meaning (e.g., it has no meaning in a BR), if one was specified, the contents are placed into the buffer.

Indirected instruction
In an instruction in which indirecting has meaning (e.g., it has no meaning in a SXAI), if the instruction indirected, the indirecting level is placed into the buffer.

Effective address calculation
In an instruction in which an effective address was calculated, the effective address is placed into the buffer.

Contents of core location
For those instructions which access core locations, the old contents of the core location are put into the buffer, and if the contents can be modified by the instruction, the new contents of the location are also put into the buffer.

3.5.6 SINEXEC

Interprets all SOS instructions except AXAI, BR, H, SVC, and SXAI. The opcode is cased on, operands are calculated, and the operation performed.

CODED BY: C. C. Gallagher

CALLED BY: EXECUTE

ROUTINES CALLED: EFFECT TRACE

EXTERNAL REFERENCES: SOSCB TRACEBLK

LIBRARY MACROS: CALL CMSREG SOSCB SOSENTER TRACEBLK

On entry, R2 should contain the current value of the program counter, R3 should point to SOS core, R6 should point to SOSCB, and R12 should point to TRACEBLK. The effective address of the instruction is determined by calling EFFECT. If it is invalid, we return to caller. If not, the contents of the register referred to by the instruction are pre-fetched. The word at the effective address is fetched and placed into TRACEBLK. The instruction opcode is the fetched and cased on to be interpreted. On return from execution of the instruction, TRACE is called to perform user tracing functions, and we then return to caller.

The various instructions and their actions are:

Add
Subtract
The contents at the effective address are added to or subtracted from the contents of the register. The new contents are placed into TRACEBLK and the SOS register. If overflow occurred, the appropriate byte in the overflow flag vector is set to X'FF', otherwise it is set to X'00'.

Shift Left Algebraic
The effective address is used to determine the number of bits to participate in the SLA. The contents of the register are stored into TRACEBLK and into the SOS register. If overflow occurred, the appropriate overflow byte is set to X'FF', else to zero.

Shift Right Algebraic
The register contents participate in a /360 SRA, the number of bits shifted being the effective address. The register contents after the shift are stored into the SOS register and TRACEBLK.

Load
The value at the effective address is placed into the specified SOS register and TRACEBLK.

Store
The contents of the specified register are stored at the effective address and into TRACEBLK.

And
Or
Xor
The word at the effective address is anded, ored, or xored with the register contents. The result is stored into the SOS register and TRACEBLK. The logical condition code for the specified register is then set to ones (X'FF'), zeros (X'00'), or mixed (X'0F').

Shift Left Logical
Shift Right Logical
The effective address is used to determine the number of bits to participate in the SRL or SLL. The new contents are stored into the SOS register and TRACEBLK, and the condition code for the appropriate register is set to ones (X'FF'), mixed (X'0F'), or zeros (X'00').

Shift Left Double Logical
Shift Right Double Logical
The register is tested to see whether it is R15. If it is, SOSABEND is set to 2 and we return to caller. Otherwise, the double register is loaded and shifted (SRDL or SLDL) by the number of bits specified in the effective address. The resulting double register is replaced into the SOS register set and stored into TRACEBLK, and the condition code for the first register of the pair is set to ones (X'FF'), mixed (X'0F'), or zeros (X'00').

Branch Low
Branch Equal
Branch High
The specified register is tested via LTR. If it meets the conditions for taking the branch, the effective address is loaded into the program counter.

Branch Ones
Branch Mixed
Branch Zeros
The condition code flag for the specified register is accessed. If it meets the criterion specified in the branch (X'FF' for ones, X'0F' for mixed, X'00' for zeros), the effective address is loaded into the program counter.

Branch Unconditionally
The effective address is loaded into the program counter.

Branch on Overflow
The overflow flag for the specified register is accessed. If it is on, the effective address is loaded into the program counter.

Branch on Count
The contents of the specified register are decremented by one. This new value is stored back into the SOS register set and into TRACEBLK. If the resulting value is not zero, the effective address is loaded into the program counter.

Branch and Link

The program counter is stored into the specified register and into TRACEBLK. The effective address is then loaded into the program counter.

Test Under Mask

The contents of the specified register are logically anded with the mask at the effective address. If the result is zero, this means that there are no bits positionally common to both words, so the condition code is set to zeros (X'00'). If the mask is equal to the result, than the condition code is set to all ones (X'FF'). Otherwise, the condition code is set to mixed (X'0F').

Multiply

If the specified register is R15, SOSABEND is set to 2 and we return to caller. Otherwise, the contents at the effective address are used to multiply the contents of the specified register, with the resulting double-register answer replacing the register and register+1 in the SOS register block. Appropriate information is also stored into TRACEBLK.

Divide

The specified register is checked to see whether it is R15. If it is, SOSABEND is set to 2 and we return to caller. Otherwise, the contents at the effective address are tested to see whether they are zero. If so, SOSABEND is set to 3 and we return to caller. Otherwise, the double register starting with the specified register is divided by the contents at the effective address (EXECUTE has set up a SPIE on a divide exception which will turn on the appropriate overflow flag if a divide exception occurs, then return to the statement following the divide). The resulting values are stored back into the SOS register block and into TRACEBLK.

Shift Left Double Algebraic

The specified register is tested to see whether it is R15. If so, SOSABEND is set to 2 and we return to caller. Otherwise, the second register is fetched, the double register is SLDAed by the number of bits specified in the effective address, and the result is stored back into the SOS register block as well as TRACEBLK. If overflow did not occur, the overflow flag for the specified register is set to X'00'. If it did occur, the overflow flag is set to X'FF' and SOSABEND is set to 15 so that a warning will be printed.

Shift Right Double Algebraic

The specified register is tested to see whether it is R15. If it is, SOSABEND is set to 2 and we return to caller. Otherwise, the contents of the register following the specified one are loaded. Both registers then participate in the shift, the number of bits being specified by the effective address. The resulting pair is stored back into the SOS register set, as well as into TRACEBLK.

3.5.7 SVCALL

Processes an SVC. The CCW chain is chased down, interpreted, and the appropriate I/O operations are performed.

CODED BY: C. C. Gallagher

CALLED BY: EXECUTE

ROUTINES CALLED: DUMP EFFECT IOMATIC TRACE

EXTERNAL REFERENCES: FIRSTL(FORMTRCE) SOSCB TRACEBLK

LIBRARY MACROS: CALL CMSREG IOMATIC SOSCB SOSENTER TRACEBLK

On entry, R6 should point to TRACEBLK and R12 should point to SOSCB. EFFECT is called to determine the location of the SVC chain. If EFFECT sets SOSABEND, we return to caller. Otherwise, we call TRACE to perform user tracing functions. The SVC operand is tested to see whether it is SVC 0. If not, SOSABEND is set to 5 and we return to caller. Otherwise, the CCW is checked to see whether the second word is outside of SOS core. If it is, SOSABEND is set to 6 and we return to caller.

A processing loop is now entered. The CCW is fetched and its opcode parsed. If the opcode is greater than 17, the appropriate status bit is turned on. Otherwise the opcode is cased on and the appropriate function is performed. If there was a fatal error executing the CCW the loop is exited. The chain bit is then checked. If it is on, the next CCW is fetched (making sure that it is still within SOS core) and the loop is repeated. Otherwise, we return to caller.

The various CCW's (arranged by opcode) are:

0: Invalid CCW code; the appropriate CCW status bit is turned on.

1: Backspace the Carriage: The user output buffer cursor is compared against the number of backspaces. If there are more backspaces than the advancement of the cursor, the cursor is reset to the beginning of the line and status bit 6 is set in the CCW. Otherwise the number of backspaces is subtracted from the cursor.

2: Tab the Carriage to the Next Tab Stop: If we are at the right margin, we merely remain there. If not, the current print position is determined, and a TRT is performed within the tab-stop buffer (each byte is zero if there is a tab stop there and one if there is not). If a tab stop is found, the cursor is updated to that position. Otherwise, status bit 5 is set in the CCW and a carriage return is performed by calling the RET routine.

3: Set Tab Stop: The tab stop position is fetched. If it is zero or greater than 120, the command is ignored. Otherwise, the appropriate position in the tab stop buffer is set to one.

4: Clear Tab Stop: The tab stop position is fetched. If it is zero or greater than 120, the command is ignored. Otherwise, the appropriate position in the tab stop buffer is set to zero.

5: Print Current Buffer and Return Carriage: The current print buffer is printed by calling IOMATIC. The buffer is then set to all blanks and the print count is decremented. If the print count is exceeded, SOSABEND is set to 13. Otherwise, the output buffer cursor is set to the beginning of the line. The count field of the instruction is then fetched, and if it is greater than zero, line skipping is performed. If the number of lines to skip is more than the number of lines on the current page, a page eject only is performed (if the print count is exceeded, SOSABEND is set to 13). Otherwise, a loop is entered to print the number of blank lines specified, with SOSABEND being set to 13 if the print count is exceeded.

6: Space the Carriage: The value in the count field is added to the current position of the cursor. If the sum exceeds 120, status bit 5 is set and RET is called to print the line and return carriage, with the cursor being reset to the left margin.

7: Set Page Headings: If the value of the count field is one, the first line of the heading buffer (SOSHDR) is set to the contents of the user output buffer, the buffer is cleared, and the cursor is reset to the left margin. If the count field is two, the second line of the heading buffer (SOSHDPR1) is set to the contents of the user output buffer, the buffer is cleared, and the cursor is reset to the left margin. Otherwise, a page eject is forced by setting SOSPRTCT to zero.

8: Put Hexadecimal Number: The second word of the CCW is fetched and the effective address is calculated (returning to caller if there are errors in the addressing). If the count field is greater than zero, it is tested to see whether it is greater than eight, and set to eight if so. The word is fetched and unpacked and translated into a work area (thus there are eight hex characters in the work area at this point). The digits to be printed are determined, and the number of digits is added to the current print position. If the user output buffer overflows, status bit 5 is set and the cursor wraps to the beginning of the line. The output buffer is then checked to see whether the number will overprint, and if so status bit 6 is set. The number is then moved into the buffer and the output cursor updated. If we had previously attempted to go past the right margin, the internal RET routine is called to print the line and return carriage.

9: Get Hexadecimal Number: The second word of the CCW is fetched and the effective address computed. If there were addressing errors, we return to caller. Otherwise, we calculate the number of core locations from the effective address to the end of core. The count field is fetched from the CCW. If it is zero we return. If EOF has already occurred we set SOSABEND to 12 and return. The return count is zeroed, and the next card is read from SYSUT1. If EOF occurs or the statement read is a /END or /SOS card, the EOF flag is set and status bit 7 is also set.

Otherwise, the SOSTRTNO table is modified to treat commas as blanks and we enter a loop for the number of numbers in the count field.

If we have gone past the end of SOS core, SOSABEND is set to 8 and we return. Otherwise the next non-blank is searched for. If it is not found, status bit 3 is set (fewer numbers on card than expected) and we return. The end of the hex number is searched for. If it is too long or has an invalid hex digit, status bit four is set and we return. The number is then converted from character to hex and stored in the appropriate core location. The SOS core cursor is incremented to point to the next word in core and we repeat the loop if there are more numbers to be read.

A: Put Decimal Number: Same as CCW 8 (Put Hexadecimal Number) but conversion is by CVD and EDMK to get a decimal number. Provisions are also made to allow for an optional negative sign. Thus the maximum count field is 11 rather than eight.

B: Get Decimal Number: Same as CCW 9 (Get Hexadecimal Number) but conversion involves decimal numbers. An optional minus sign is tested for and remembered. If there are more than 10 digits or if there are ten digits and they exceed 2,147,473,468 for a negative number or 2,147,473,467 for a positive number, status bit 4 is set and we return. Otherwise a pack and CVB are used for the conversion to be stored into core.

C: Put Characters: The second word of the CCW is fetched and the effective address is calculated. If in error we return. Otherwise the number of characters desired is fetched and if it is greater than zero we calculate the number of words desired, rounded up if necessary. If the number of words plus the starting address falls outside of SOS core, SOSABEND is set to 8 and we return. Otherwise, the buffer is checked to see whether there is enough room for all the characters. If not, the count is decremented so that it will move in what it can and status bit 5 is set. Overprinting is checked, and if it occurs status bit 6 is set. The characters are then moved into the line and the print cursor is updated. If we went off the right end of the line, the line is printed and a carriage return is performed, abending if the print count was exceeded.

D: Get Characters: The second word of the CCW is fetched and the effective address is calculated. If there is an addressing error, we return. If the count is zero, we trivially return. Otherwise, the number of words we need is calculated (rounding up). If the effective address plus this number falls outside of SOS core, SOSABEND is set to 8 and we return. If EOF already occurred, SOSABEND is set to 12 and we return. Otherwise, IOMATIC is called to read the next card from SYSUT1. If EOF occurs or it is a /SOS or /END card, status bit 0 is set as well as internal EOF flags, and we return. Otherwise, the data is moved into SOS core and, if necessary, padded with one to three blanks.

E: Partial Dump: The second word is fetched and the effective address calculated. If there is an error we return. Otherwise, we parse the second CCW of the PDUMP. If it is outside of SOS core, SOSABEND is set to 6 and we return. Its effective address is calculated and if in error we return. The two effective addresses are put into a DUMP parameter list. An identifying message is printed and DUMP is called to do the actual dumping. If the print count has been exceeded, SOSABEND is set to 13.

F: Invalid CCW. Status bit 6 is turned on.

10: Turn Trace Off: If the count field is not zero, it is decremented by one and replaced into the CCW. If it is zero, SOSTRST is set to zero (turning the trace off) and the execution page headers are cleared.

11: Turn Trace On: If the count field is not zero, it is decremented by one and replaced into the CCW. If it is zero, SOSTRST is set to X'FF' (turning the trace on) and the trace headers are moved from FIRSTL into the execution header area.

3.5.8 TRACE

Called to perform execution time tracing functions. If the current instruction is a branch, it is saved in the branch block area so that ABNDTRCE can print out the last ten branch instructions if necessary. If the trace is off, the Trace Control Block for the instruction is saved for ABNDTRCE if needed; otherwise TRACEBLK is passed to FORMTRCE to be printed immediately as a trace line.

CODED BY: C. C. Gallagher and S. H. Gudes

CALLED BY: EXECUTE SINEXEC SVCALL

ROUTINES CALLED: FORMTRCE

EXTERNAL REFERENCES: SOSCB TRACEBLK

LIBRARY MACROS: BRANCH CALL CMSREG SOSCB SOSENTER TRACEBLK

The current instruction is tested to see whether it is a branch (TRACEBLK code in the range X'10' to X'1A'). If it is, the branch counter is incremented by one (if it is less than 10), the last used block is pointed to, the next available block is calculated (with wrapping if necessary), and the necessary information is saved in the branch control block.

If the trace is off, the trace counter is incremented by 1 (if it is less than 10), the next available block is calculated (with wrapping if necessary), and the current TRACEBLK is moved into this block.

If the trace is on, the trace blocks are cleared (so that information is not printed twice). The trace count is decremented by one, and if the trace limit is exceeded, the trace is turned off and a warning flag is set (the warning message will be printed by EXECUTE). Otherwise, the updated trace count is saved, and FORMTRCE is called to print the trace line.

In order to speed up execution time (since this routine is called for every instruction executed), no save area linking is performed unless it is necessary (i.e. unless FORMTRCE is called). A circular queue of 10 elements (zero to nine and back to zero again) is used instead of the simpler MVC type queue since it was found that the MVC implementation tripled (that's right, tripled) execution time.

4 CONTROL BLOCKS

4.1 Active Macro Table (AMT)

	0			3
0	number of entries			
4	S	Y	S	N
8	D	X	b̸	3
12	pointer to value of SYSNDX			
16	name of first macro prototype parameter			
20	"			length
24	pointer to value of first macro parameter			
	. . .			
	name of last macro prototype parameter			
	"			length
	pointer to value of last macro parameter			
	invocation statement (80 bytes)			
	invocation continuation (if any) (80 bytes)			

Following a fullword containing the number of entries, each symbolic parameter consists of a seven character name, a one byte length of the value, and a pointer to the value.

The AMT is used to set up an equivalence between a symbolic parameter name and its value for a particular macro invocation. The names and values for all macro parameters are in the AMT, with pointers to either the value in the invocation statement at the end of the AMT or the default keyword value in the associated MCB.

There is no need to distinguish between label, positional, and keyword parameters, and indeed &SYSNDX is merely another entry in the table.

An AMT is created by MACEXPND each time a macro is to be expanded, and is freed when macro expansion has been completed.

4.2 Branch Control Block (BRBLK)

	0		3
0	register contents		
4	instruction image		
8	index register contents		
12	effective address	PC before branch (loc)	
16	indirecting level	type	unused
20	PC after branch	unused	

The Branch Control Block is used to save information on each branch instruction so that the "trace of the last ten branch instructions executed" may be printed.

The information is obtained from the trace control block (section 4.10) for the current instruction. The register contents field is the old contents (from TRACEBLK) unless the instruction is BCT, in which case it is the new contents. The type field is as described under TRACEBLK. The new value of the PC is either one greater than the old value (if the branch was not taken) or the effective address (or register contents for BR) if the branch was taken.

4.3 Card Image Status Word (CISW)

The CISW is used to remember information between pass 1 and pass 2 of the assembler. There is one CISW generated for each card image read by the assembler (including macro generated statements). These are then used in pass 2 to avoid re-scanning for information which is already known.

The CISWs are one fullword (32 bits) long and divided into several fields. These are:

Byte 1

bit 0: 1 if pass 1 detected a fatal error in the statement, 0 otherwise (if this bit is on, the assembler will generate an abend-forcing instruction (opcode X'FF')).

bit 1: 0 if the card image is from the user's input file, 1 if the statement was generated by the macro generator (basically indicates whether PASS2 should read the card image from SYSUT1 or from SYSUT2)

bits 2-7: the statement type, as follows:

01 SPACE	0B macro continuation
02 TITLE	0C macro invocation
03 EJECT	0D comment, macro model stmt
04 PRINT	0E MEND
05 END	0F macro definition prototype
06 DC	10 general instruction
07 EQU	11 BR
08 DS	12 B
09 ORG	13 AXAI, SXAI
0A MACRO	14 CCW

FF and BF are special codes used for a bad macro prototype which indicates that even though an error has occurred, an abend-forcing instruction is not to be generated (since a macro prototype is not an executable instruction).

Byte 2

Two four-bit non-fatal error codes, the first from the macro generator, the second from the assembler. Zero in either indicates no non-fatal error for the statement. The non-fatal error codes are:

code	assembler error	macro generator error
1:	1034 label ignored	3022 badly formed MACRO stmt
2:	1035 effect addr out of range	3023 badly formed MEND stmt
3:	1036 duplicate label	3024 substitution overflow
4:	1037 label too long	
5:	1038 invalid character in label	

Bytes three and four are used for various purposes depending on the type of statement, as follows:

CCW
END
PRINT
SPACE
TITLE
The third byte is not used. The fourth byte indicates the column in which the scan for the operand field should begin (generally the first blank following the opcode).

comment
EJECT
MACRO
macro model statement
macro prototype
macro continuation
macro invocation
MEND
Bytes three and four are not used for these statements.

DS
ORG
The first bit of the third byte is on if a fatal error occurred during pass one (this is used rather than bit 0 of byte 1 so that an abend-forcing statement will not be generated). If there was no fatal error, bytes three and four are the new value to which the location counter should be set.

EQU
The first bit of the third byte is on if a fatal error occurred during pass one (this is used rather than bit 0 of byte 1 so that an abend-forcing statement will not be generated). If there was no fatal error, byte three contains the length of the label field on the statement, and byte four is ignored.

AXAI
B
BR
general machine instruction
SXAI
Byte three is the opcode of the statement. Byte four indicates in which column the scan for the operand field should start (generally the first blank following the opcode).

DC
The fourth byte contains the offset to the beginning of the operand field (the scan was performed in pass one). The first bit of the third byte is either off, to indicate an expression as the operand, or on to indicate a C-type character string as the operand. If it is an expression, the rest of byte three is ignored; if it is a string the rest of byte three contains the length of the string (in bytes) as determined during the pass one scan.

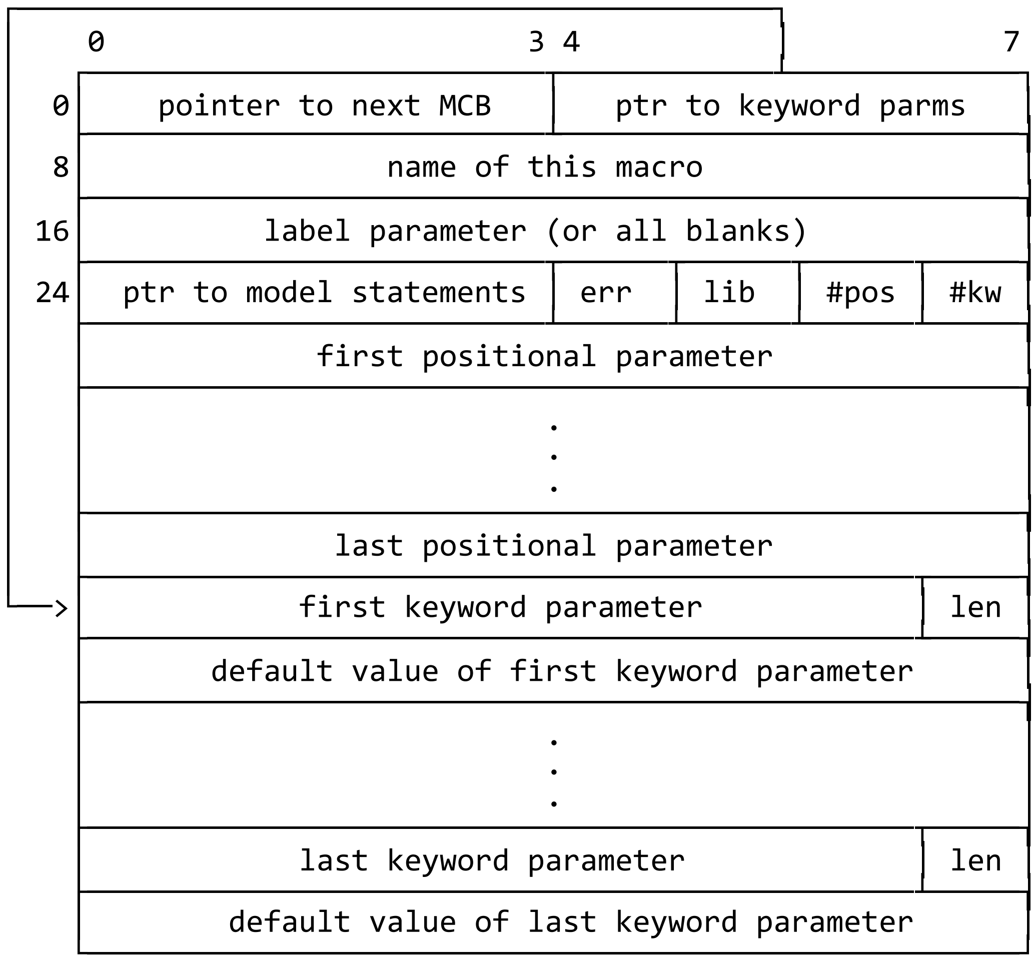
4.4 Default Parameter Block (DEFBLOCK)

```
DEFMAXTR DS    F           Maximum Number of Instructions to Trace
DEFMAXPR DS    F           Maximum Number of Lines to Print
DEFMAXIN DS    F           Maximum Number of Instructions to Execute
DEFMAXER DS    F           Maximum Number of Errors Allowed
DEFMAXLN DS    F           Maximum Number of Lines per Page

DEFTRACE DS    F           Default Number of Instructions to Trace
DEFPRINT DS    F           Default Number of Lines to Print
DEFINSTR DS    F           Default Instruction Limit
DEFERROR DS    F           Default Maximum Number of Errors
DEFLINCT DS    F           Default Number of Lines per Page

DEFDUMP  DS    X           Whether to Dump on Abend
DEFMACRO DS    X           Whether to Fetch System User Macros
DEFWARN  DS    X           Whether to Print Warnings
DEFTRFLG DS    X           Initial Trace Status
DEFLIST  DS    X           Whether to Print the Listing
DEFXREF  DS    X           Whether to Print a Cross-Reference
DEFINDAT DS    X           Whether to Print User Input Data
DEFASM   DS    X           Whether Program is Assembler or Machine

DEFHDR   DS    120C        Initial Header Information

DEFMGNUM DS    H           Number of Lines in Message Block
DEFMGLEN DS    H           Length of Each Line in Message Block
MSGBLOCK EQU   *           The Message Block Itself
```

The default block contains initial values for SOS card parameters. These values are moved into the appropriate SOSCB positions at the beginning of each job, and may later be modified by values on the /SOS card.

Modifying the default value block is done by simply changing the appropriate parameters in the DEFBLOCK macro, re-assembling the DEFBLOCK csect, and re-linkediting the csect into the SOS load module.

4.5 Macro Control Block (MCB)


0
3 4
7
0
pointer to next MCB
ptr to keyword parms
8
name of this macro
16
label parameter (or all blanks)
24
ptr to model statements
err
lib
#pos
#kw
first positional parameter
.
.
.
last positional parameter
first keyword parameter
len
default value of first keyword parameter
.
.
.
last keyword parameter
len
default value of last keyword parameter

A Macro Control Block is set up for each macro known to the system. MCBSETUP sets up the MCB. It is called from MACOPEN if the macro is in the system library, or from MACSAVE if the macro is user-defined.

The MCB is used in AMTSETUP to establish an equivalence between a symbolic parameter name and its value for a particular macro call (see Active Macro Table above).

The default values for a keyword parameter are of variable length, therefore the alignment of any keyword parameter following the first is not guaranteed. The length field is specified as it would be for an SS instruction. If the value of a keyword parameter is null, its length field is set to X'FF' and no default value is moved in.

4.6 Pass 2 Control Block (PASS2CB)

	0	1
0	conversion area (9 bytes)	
8		indirecting flag
10	index field	register field
12	make-an-abend flag	error code for stmt
14	print card flag	hit-END-card-yet flag
16	number of errors found	

This serves as a global data area for assembler pass 2 and PROCARD. It contains space for hex - decimal - character conversions, and items describing the statement being currently processed, as well as items describing the state of the assembler in terms of SYSIN EOF and the number of errors detected thus far in the assembly.

4.7 Register Equate Control Block

	0	3
0	8 character label	
4	"	
8	pointer to left daughter	
12	pointer to right daughter	
16	value of label	
20	pointer to top of XREF block list	
24	in use flag	unused

Each register (R0, R1, ..., R15) has a control block associated with it. These blocks are arranged contiguously in core, so that they may be accessed directly rather than by binary search, and also have the necessary binary tree pointers to be added to the symbol table so that the cross-reference may be printed with the registers in proper collating sequence order.

4.8 SOS Control Block (SOSCB)

The SOS Control Block contains global values, buffers, and control areas used throughout the SOS system. The block is divided into several sections, described below. For the actual physical layout of SOSCB, refer to the macro for its expansion (which can be found in SOSMLIB). For a list of entries and their usage, see section 6.

TABLES: 3 256-byte tables used to TRT for blanks, non-blanks, and pops, and a 16-byte hex-to-EBCDIC translation table.

POINTERS: pointers to important control areas, tops of linked lists, SOS core, and free storage.

BOOKKEEPING PARAMETERS: the bookkeeping parameters to be used for this job, set from the Default Value Block and the /SOS card.

BETWEEN PHASE PARAMETERS: information shared throughout either phase, or passed from phase to phase or pass to pass. Contains execution start address, file information, and oft-used parameter lists.

FLAGS AND COUNTS: various information items used throughout the system, e.g. page numbers and line counts, &SYSNDX, nesting levels, storage lengths, error count, location counter, blanks for blanking.

BUFFERS: the three lines of header for each page, execution output line and tab indicator, error buffer, and input save area.

4.9 Symbol Table Entry

	0 3
0	symbol
4	"
8	pointer to left daughter
12	pointer to right daughter
16	value of symbol
20	pointer to top of XREF list

An entry is made for each symbol defined to SOS, whether as a label or as an EQU'ed value. The value of the symbol, and a pointer to the top of its XREF list, as well as symbol table binary tree information, are provided.

4.10 Trace Control Block (TRACEBLK)

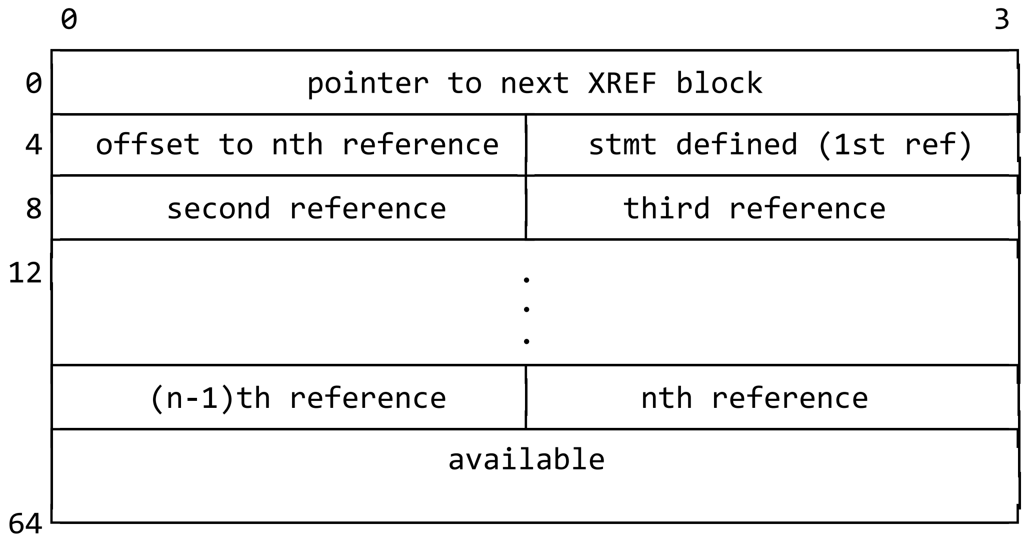
	0							7
0	old location contents				new location contents			
8	old register contents							
16	new register contents							
24	instruction				index register contents			
32	effect addr		program ctr		indirect level		code	unused

This block contains information which is used by FORMTRCE to print an execution trace. The information is set into the block at various points during execution, and is retrieved, converted, and printed (if the trace is on) after the instruction has been executed. The information is also saved to print the last ten instructions if the user abends.

The values which the "code" field may take on are:

01	A	12	BE
02	S	13	BL
03	M	14	BOF
04	D	15	BO
05	H	16	BM
06	L	17	BZ
07	ST	18	BAL
08	SRL	19	BR
09	SRA	1A	BCT
0A	SRDL	1B	SXAI
0B	SRDA	1C	AXAI
0C	SLL	1D	N
0D	SLA	1E	O
0E	SLDL	1F	X
0F	SLDA	20	TM
10	B	21	SVC
11	BH		

4.11 XREF Table Control Block



The XREF block contains the statement numbers of all references to a particular symbol within a particular assembly. The first reference is the statement in which the symbol is defined. XREF blocks are allocated and chained in as needed. The pointer to the head block is contained in the symbol table control block for the particular symbol. The length of the block is 64 for a symbolic register symbol (R0 - R15) and 32 for all other XREF blocks.

5 SOS INTERNAL MACROS

These macros are used extensively by SOS routines. They are available in SOSMLIB MACLIB (under CMS) or in SYS3.P220700.U000.SOS.ASMGLIB (under OS). If the macro library is used, it must be the first library to be encountered in the library search, as some of the names used are the same as those of OS or CMS system macros.

Most of these macros are used in both the CMS and OS versions. Those which are restricted to one system have the name of the system in parentheses following the macro name.

The CMS I/O macros are not compatible with the standard CMS macros, even though they have the same names. The major difference is that these macros invoke the routines by BALR, whereas the CMS macros invoke them by SVC. As such, the CMS SYSLIB macro FVS must be generated when using SOS CMS macros, and R13 must be set to point to the nucleus FVS area prior to invoking any of the CMS macros.

All macros which generate executable code may be assumed to destroy R0, R1, R14, and/or R15.

5.1 ABENDSOS

```
[label]  ABENDSOS  code,message[,LABEL=symbol]
```

Invokes the ABENDSOS entry-point in SOS to terminate operation of the SOS system. Batch recovery is not attempted.

code: the user completion code (OS) or return code (CMS). This must be a term acceptable in the context AL1(code).

message: an explanatory text, enclosed in pops. This message is printed at the terminal under CMS or written to programmer under OS.

LABEL: an optional label to be placed on the DC of the message. This allows information (such as system return codes) to be placed into the message as an offset from the beginning of the message.

5.2 BRANCH

```
         BRANCH
```

Generates a dsect (named BRNCSECT) to describe a Branch Control Block (section 4.2).

5.3 CALL

```
[label]  CALL      routine[,parm][,ERROR=addr]
```

Calls an external routine.

routine: the external entry point to be called.

parm: an optional parameter. If specifed, LA R1,parm is done prior to calling.

ERROR: an optional address to branch to on a non-zero return code. If not specified, error returns are ignored. If used, it is assumed that the routine called will do an LTR R15,R15 prior to returning. May be specified as a label or base-displacement pair.

5.4 CMS (CMS)

```
[label]  CMS       (A1,A2,...,An)[,ERROR=addr]
```

Sets up the parameter list for and invokes the specified CMS command via SVC 202.

An: each of the A's are a word of a CMS command. The word is either specified as is, or, if it contains commas or parentheses, may be specified in apostrophes. For example, one might code:

```
CMS   (LOGIN,193,'B,P','(NOTYPE')
```

The specified command must be transient; if it is an EXEC file the word "EXEC" must be the first item of the list.

ERROR: specifies an address to branch to if the CMS command returns a non-zero error code. Default is to continue execution sequentially. The address must be valid in the context A(addr).

5.5 CMSREG

```
         CMSREG
```

Generates equates for the sixteen GPR's: R0, R1, ..., R15.

5.6 DEFBLOCK

```
         DEFBLOCK  [CSECT=YES|NO]
```

Generates a csect (named DEFBLOCK) or dsect (named DEFDSECT) defining or describing the default value block (section 4.4). CSECT=NO is the default.

5.7 FCB (CMS)

```
[label]  FCB       file,area[,LRECL=number][,RECFM=F|V]
```

Used under CMS only, this macro generates a File Control Block consisting of the specified parameters.

file: a list consisting of filename, filetype, and an optional filemode. For example, (*,SOS,P1) or (DORK,LISTING). Filemode defaults to P5.

area: the buffer used by this file. A(area) must be acceptable to the assembler.

LRECL: the logical record length of the file. Defaults to 80. Must be a self-defining term.

RECFM: the record format of the file. Defaults to F.

5.8 IOMATIC

```
[label]  IOMATIC   func,fileid[,option][,AREA=addr][,LEN=number]
                   [,ERROR=addr][,MODE=MOVE|LOCATE]
```

Generates a parameter list and calling sequence to invoke IOMATIC, the central I/O processor.

func: the function to be performed, namely READ, WRITE, OPEN, CLOSE, NOTE, or POINT.

fileid: the file which is to be accessed, namely SYSIN, SYSPRINT, SYSUT1, or SYSUT2.

option: for OPEN, it may be INPUT, OUTPUT, or OUTIN; for CLOSE, it may be T. All other uses are ignored.

AREA: the input or output area to be used, or the NOTE/POINT information block. Ignored if function is OPEN or CLOSE, or if LOCATE mode is used. Default is SOSCOMIN. NOTE and POINT require an eight byte control area, fullword aligned. May be specified as a label or base-displacement pair.

LEN: the length of the user's buffer. Ignored for input, must be between one and 121 for output. Default is 80. Specify as a self-defining term.

ERROR: the address of an error return. May be specified as a label or a base-displacement pair. Default is to ignore errors.

MODE: valid only for READ, specifies whether data read is to be moved to the buffer specified by AREA, or whether a pointer to the data is to be returned in R1. Default is MOVE.

5.9 MCB

```
         MCB
```

Generates a dsect (named MCBSECT) to describe a macro control block (section 4.5).

5.10 NUCLEUS (CMS)

```
[label]  NUCLEUS   func,object[,ERROR=addr]
```

Invokes a nucleus-resident CMS function. The CMS system macro FVS must be generated, and R13 must be pointing to the FVS low core control area.

func: the name of the function, e.g. ERASE, FINIS, STATE.

object: a label or base-displacement pair specifying the address of the parameter list.

ERROR: the address to branch to if the nucleus routine returns a non-zero condition code. Defaults to no action on error. May be specified as a label or base-displacement pair.

5.11 PASS2CB

```
         PASS2CB   [CSECT=YES|NO]
```

Generates a dsect (named PASS2DSC) or a csect (named PASS2CB) to describe the PASS2 common information control block (section 4.6). CSECT=NO is the default.

5.12 RDBUF (CMS)

```
[label]  RDBUF     fcb[,AREA=addr][,ERROR=addr]
```

This macro generates the linkages needed to invoke the CMS RDBUF routine. It is assumed that the CMS system macro FVS has been generated, and that R13 is pointing to the low core FVS area.

fcb: the address of the file control block to be used. May be specified as a label or base-displacement pair.

AREA: the buffer address. May be specified as a base-displacement pair or label. Defaults to the address specified in the FCB macro.

ERROR: the address to branch to if the RDBUF routine returns a non-zero condition code. May be specified as a label or a base-displacement pair. Defaults to no-operation on error.

5.13 SETUP (CMS)

```
[label]  SETUP     fcb,error
```

Opens the specified file and sets up the FCB from the FSTB. It is assumed that the CMS system macro FVS has been generated, and that R13 is pointing to the lowcore FVS area.

fcb: the address of the file control block containing the name, type, and mode of the file to be found, and which will be changed to reflect the current status of the file if it is found. The address may be specified as a label or as a base-displaceement pair.

error: the address to branch to if the file is not found. May be specified as a label or a base-displacment pair.

5.14 SOSCB

```
         SOSCB     [CSECT=YES|NO]
```

Generates a csect (named SOSCB) or dsect (named SOSDSECT) to define or describe the SOS common information block (section 4.8). CSECT=NO is the default.

5.15 SOSENTER

```
         SOSENTER  [[NOSAVE,]BASE=reg]
```

Generates the entry linkages for an SOS program. If "NOSAVE" is not specified, an eighteen word save area is generated, save area linking is performed, and R13 is made the base register. If "NOSAVE" is specified, a save area is not generated and "reg" is made the base register. In either case, a branch around the program name and a standard STM are generated.

nosave: any non-null string, specifies that a save area is not to be generated, and save area linking is not to be performed.

BASE: specifies that a base register other than R13 is to be used. Must be a self-defining term.

5.16 TRACEBLK

```
         TRACEBLK  [CSECT=YES|NO]
```

Generates a csect or dsect (named TRACEBLK) to define or describe the trace information block (section 4.10). CSECT=NO is the default.

<u>6 SOSCB DEFINITIONS</u>

SOS@CISW:	Points to the first CISW block (section 4.3) in the CISW block chain. Zero if there are no blocks.
SOS@CORE:	Points to SOS user core. Set by SOS at initialization.
SOS@FRCR:	Points to next available location in the free storage area. Set by SOS at initialization, accessed by GETBLOK/FREEBLOK, IOMATIC, and the macro generator.
SOS@FRND:	Points to the end of the free storage area. Usage same as SOS@FRCR.
SOS@FROR:	Points to the free storage area past the I/O buffers and (under OS) SOS core. Used in case the system library blows up and its storage to that point is to be reclaimed.
SOS@FRTP:	Points to the origin of the reusable part of the free storage area. SOS@FRCR is set to this value at the beginning of each job in the batch.
SOS@LAST:	Points to the last CISW block allocated. Zero if none have been.
SOS@LBMC:	Points to the SOS macro library MCB sub-chain. Remains constant from job to job so that the library does not have to be re-read for each job.
SOS@LBTP:	Points to last MCB allocated in the macro library sub-chain.
SOS@LIT:	Points to the top of the literal pool. Set by PASS1 so that PASS2 will know where to start loading literals.
SOS@MCCR:	Points to next available location in SOS core which can be used as scratch by the macro generator. Initially set to (SOS@CORE) + MACEXPND DSA size.
SOS@MCNM:	Points to linked list of MCB's comprising the SOS macro library and any user defined macros.
SOS@SYMB:	Points to the top of the symbol table, that is, the root of the binary tree.
SOS@TRBL:	Points to the trace control block.
SOSABEND:	Set non-zero when the user's job abends (phase I) or when execution of the user job abends (phase II).
SOSASM:	X'FF' if ASM option specified (or defaulted) on /SOS card, 00 if MACHINE.
SOSAVE:	Saves the card image of the last statement read by PHASEI so that the mainline will know whether it was a /END.

SOSBLANK:	144 bytes of blanks, conveniently placed following SOSCOMIN, so that scanning statements in SOSCOMIN and blanking out areas of core are made a little easier.
SOSCOL73:	Columns 73-80 of an assembler input statement are set to blank to be scanned. Their original contents are saved here so that they may be printed on the listing.
SOSCOMIN:	A common input buffer used by the whole system. The IOMATIC macro defaults the input area to this, and many routines expect it to contain the current card image.
SOSCORSZ:	The number of bytes to allocate for SOS core.
SOSCURSE:	Cursor along SOSPLINE at execution time (i.e., the cursor set by SVCs which manipulate the user output buffer).
SOSDASH:	Carriage control used by the SOS3 parameter list.
SOSDUMP:	X'FF' if DUMP was specified (or defaulted) on the /SOS card, 00 if NODUMP.
SOSDYNSZ:	The size to allocate for the macro generator's dynamic storage control area.
SOSEOF:	A block of four bytes (one each for SYSIN, SYSPRINT, SYSUT1, SYSUT2); zero indicating that end-of-file was not reached for that file, X'FF' indicating that it was.
SOSERRBF:	A 121 byte buffer used to accumulate and print various messages. This is a transient buffer which should only be used locally within a routine (i.e., any routine can use it at any time for any purpose).
SOSERROR:	The value of the ERROR parameter (actual or defaulted) on the /SOS card.
SOSEXEOF:	EOF flag for execution of user program (i.e., for GETC, GETX, and GETD).
SOSFILNM:	Under CMS only, the name of the SOS file being processed (this is passed as a parameter and parsed by SOS).
SOSHDPR1:	The first line of sub-heading in phase I, or the second line of heading in phase II. Ignored in phase 0.
SOSHDPR2:	The second line of sub-heading in phase I, a blank line during phase II, and ignored in phase 0.
SOSHDR:	The heading line in phases 0 and I, or the first line of heading in phase II.
SOSHOPE:	If non-zero, HOPEFUL has been invoked. Used generally to indicate flushing procedures to the mainline.
SOSINDLV:	The maximum number of indirecting levels allowed.

SOSINPUT:	X'FF' if DATA was specified on the /SOS card (or if it has defaulted); X'00' if NODATA was specified or has defaulted.
SOSINSC:	The maximum number of instructions which may be executed by the current job. Set either from the /SOS card (if specified there) or from DEFBLOCK.
SOSLC:	The current value of the location counter at assembly time. It is initialized to X'10' at the beginning of PASS1 and then incremented by the size of each statement processed. It is used to set up label definitions in the symbol table, and also in the determination of the starting location of the literal pool. During PASS2, it is initialized to X'10' and incremented as necessary to assist in the loading of the assembled statements into the appropriate core locations.
SOSLINCT:	The number of lines to print on a page of output (i.e., the number of lines to print before a page eject is forced and headings are printed). May be specified on the /SOS card or defaulted from DEFBLOCK.
SOSLIST:	Whether a source listing is to be produced. Set to X'FF' if LIST was specified (or defaulted) on the /SOS card, 00 if NOLIST was specified (or defaulted).
SOSMACRO:	Whether the SOS system macro library is to be scanned and loaded. Set to X'FF' if MACRO was specified (or defaulted) on the /SOS card, 00 if NOMACRO was specified (or defaulted).
SOSMCERR:	Set true if MACOPEN could not scan the SOS macro library due to a fatal error (e.g., bad directory).
SOSMCSCN:	Set true by MACOPEN after it has been invoked the first time, to avoid re-scanning the macro library.
SOSMODE:	Used in the CMS version only, this is set to the filemode of the SOS input file, or, if the input file is on a read-only disk, it is set to the mode of the first available read-write disk.
SOSMUNG:	A flag set in PHASEI, used to remember whether there was garbage between the END (or FFF) card and the /DATA card, and also whether a /DATA card was encountered at all.
SOSMXNST:	The maximum depth of macro nesting allowed. Setting this field to zero prevents use of the macro generator.
SOSNTLVL:	The current macro nesting depth; set by MACEXPND.
SOSNUMLT:	The number of fullwords currently in the literal pool.
SOSNXJOB:	Contains IOMATIC NOTE/POINT information indicating the start of the next job in the batch.

SOSOPEN:	A vector of four flags, one for each of SYSIN, SYSPRINT, SYSUT1, and SYSUT2, indicating whether IOMATIC has opened said file and, if so, whether it is open for input, output, or outin.
SOSOURCE:	Set to IOMATIC NOTE/POINT information pointing SYSUT1 to the first source statement of the assembler program.
SOSOVERF:	Set when there is no more room in SOS core for the macro generator to use as scratch storage.
SOSPAGE:	The current page we are on in assembly or machine code loading. Maintained by IOMATIC. Page numbers are counted and printed during phase I only.
SOSPHASE:	The current phase we are in: 0, 1, or 2.
SOSPIE:	The pointer to the previous PICA (if a SPIE was done by EXECUTE) or zero (if no SPIE was done).
SOSPLINE:	The user's execution-time output buffer.
SOSPRINT:	The maximum number of lines that the user may print during execution of the SOS program. May be set by a /SOS card parameter or from the default value block.
SOSPRLST:	Used under the CMS version only, indicates whether SYSPRINT should go to the virtual printer (X'FF') or to a disk (LISTING) file (X'00').
SOSPRTAR:	A forty byte area preceding SOSCOMIN. Used for printing an assembly, machine code, or data card directly from SOSCOMIN without any unnecessary data movement.
SOSPRTCT:	The number of lines left on the current page. Initialized to the value in SOSLINCT; zero means there are no more lines on the page (to skip to a new page, setting SOSPRTCT to zero is more efficient than writing a line with carriage control of '1').
SOSPRTGN:	Set to X'FF' if a PRINT GEN statement is encountered; to X'00' if a PRINT NOGEN is encountered.
SOSPRTON:	Set to X'FF' if a PRINT ON is encountered; X'00' if a PRINT OFF is encountered.
SOSREAD:	IOMATIC plist to read a record from SYSIN into SOSCOMIN.
SOSSTART:	The SOS core address (i.e., in the range 000 to X'FFF') at which execution of the SOS program is to begin. Defaults to X'10', or may be explicitly specified as an argument on the END or FFF pseudo-ops.
SOSSTMT:	The statement number of the current assembler input statement (or machine code statement). Maintained in both passes since the symbol table routine requires a statement number for references.

SOSSYMBS:	Set to X'FF' if any symbols are entered into the symbol table, so that we know whether or not to invoke DOXREF.
SOSSYSRC:	Used under OS only. Set to "D" if user parm is "DA" (SYSIN may be NOTE/POINTed); set to "S" if parm is "SQ" (SYSIN may not be NOTE/POINTed). Used to determine whether a scratch dataset (SYSUT1) is necessary. Defaults to "S" just to be safe.
SOSTABS:	User tab setting buffer. Each of the locations is either X'00' if a tab has not been set in that location, or X'01' if a tab has been set.
SOSTRACE:	The maximum number of instructions that may be traced during the current job. This value is set either from the default value block or from the /SOS card.
SOSTRHEX:	A sixteen byte table consisting of the characters '0123456789ABCDEF', which facilitates the translation of hexadecimal to character.
SOSTRST:	The status of tracing: X'FF' if TRACE was specified on the /SOS card (or has defaulted); X'00' if NOTRACE was specified (or defaulted), or if the trace count is exceeded. Also set by the TON and TOFF SVCs.
SOSTRTBL:	A TRT table which will scan for a blank (X'40').
SOSTRTNO:	A TRT table which will scan for a non-blank (non-X'40').
SOSTRTPP:	A TRT table which will scan for an apostrophe (X'7D').
SOSUDATA:	Contains IOMATIC note/point data pointing to the first statement of user data in SYSUT1.
SOSWARN:	X'FF' if WARN was specified on the /SOS card (or has defaulted); X'00' if NOWARN was specified (or defaulted).
SOSWRRD:	Used when PARM=SQ. If it is X'FF', any record read from SYSIN by IOMATIC will be written to SYSUT1 by IOMATIC.
SOSXREF:	X'FF' if XREF was specified on the /SOS card (or has defaulted); X'00' if NOXREF was specified (or defaulted).
SOSYSNDX:	The last-used value of &SYSNDX.
SOS1ERR:	The number of assembly (or loading) errors (not warnings!) in the current job.
SOS2ND:	A flag used in addition to SOSEXEOF to avoid a second-read-after-EOF message on the first read if there is no user input data.
SOS3:	IOMATIC parameter list to skip 3 lines on SYSPRINT.

TABLE OF CONTENTS