SPERRY UNIVAC

Assembly Language
Programmer Reference

Mini-Computer Operations

2722 Michelson Drive
P.O. Box C-19504
Irvine, California 92713

ASSEMBLY LANGUAGE PROGRAMMER REFERENCE MANUAL

98A 9952 452

FEBRUARY 1978

CHANGE RECORD

Page Number	Issue Date	Change Description
all	10-76	original issue
misc.	5-77	minor revisions/corrections
misc.	2-78	deleted references to Varian

Change Procedure:

When changes occur to this manual, updated pages are issued to replace the obsolete pages. On each updated page, a vertical line is drawn in the margin to flag each change and a letter is added to the page number. When the manual is revised and completely reprinted, the vertical line and page-number letter are removed.

LIST OF EFFECTIVE PAGES

Page Number	Change in Effect
All	Complete Revision

TABLE OF CONTENTS

SECTION 1
INTRODUCTION

SECTION 2
STATEMENTS

SECTION 3
INSTRUCTION SUMMARY

SECTION 4
ASSEMBLER DIRECTIVES

SECTION 4 *(continued)*

SECTION 5
OPERATING THE ASSEMBLER

SECTION 6
STAND-ALONE FORTRAN/DAS MR LIBRARIES

APPENDIX A
INDEX OF INSTRUCTIONS

APPENDIX B
V70 SERIES ASCII CHARACTER CODES

LIST OF TABLES

LIST OF ILLUSTRATIONS

SECTION 1
INTRODUCTION

This manual describes the assembly language and assembler processing used to write, assemble, and execute programs for the SPERRY UNIVAC V70 series computers.

1.1 V70 SERIES ASSEMBLY LANGUAGE

The assembly language is a symbolic representation of the programmable capabilities of the V70 series computers. Using assembly language, the programmer is able to specify the machine instruction codes symbolically and to address memory locations by alphanumeric symbols of his own choosing, providing a flexibility not attainable with absolute addressing.

Internally, the computer obeys instructions kept in its memory in 16-bit binary format. For example, the instruction:

    001000000001111

when executed causes the A register to be loaded with the contents of location 15 (decimal). In octal the same instruction is written:

    010017

However, it is not necessary to learn the octal or binary representation of the computer's instruction repertoire. Instead, a user can write his program using a symbolic language and then use another computer program, the DAS (Data Assembly System) assembler, to convert the instructions to binary upon input. The instruction given previously is then written:

```
LDA     017
```

or, if decimal working is preferred:

```
LDA     15
```

which is read as "Load the A register with the contents of location 15 (decimal)."

The DAS assembler translates the statement "LDA 15" into its binary machine language equivalent, i.e.:

LDA 15 ⟶ DAS ASSEMBLER ⟶ 001000000001111

Similarly:

```
STX     0177
```

is translated by the DAS program to form the instruction "Store the X register contents in location 0177."

The DAS assembler has many other capabilities than translating source instructions one-for-one into their binary equivalents. A primary feature is allowing the programmer to represent memory locations with symbolic labels instead of requiring absolute addresses. Another feature allows the programmer to define data constants and character constants without prior conversion to binary or octal values. For example, suppose the user wishes to load the A register with the value 64 at some point in his program. He could do this with the following statements:

```
VALU      DATA      64
          .
          .
          .
          LDA       VALU
```

The first statement defines a word of data having the value 64; "VALU" is a symbolic label that can be used to address that data word. The second statement is an instruction to load the A register with the contents of memory location "VALU". The programmer need not be concerned with the absolute location of the data word.

An even simpler version--requiring only one statement--can be written using a "literal" constant:

```
          LDA       =64
```

In this version, the assembler itself will designate a location in which the value 64 is to be placed.

DAS assembly language allows the user to give directions to the assembler, called assembler directives, to perform such functions as defining program loading addresses, data locations (such as the DATA directive above), subroutine linkage, and input/output functions; further control features include conditional assembly directives and a macro capability. Comments can be added between symbolic source statements or appended to the statements themselves to enable easier checkout and program documentation.

By using the DAS assembly language, the programmer is able to write functional application programs and control the operation of the assembler. Symbolic coding reduces machine language bookkeeping and fully utilizes the computer capabilities without a corresponding increase in the time required for programming.

1.2 DAS ASSEMBLERS

The principal objective of any assembler is to translate source programs written in a symbolic machine language into the more precise numeric language of the computer. The assembler (DAS) achieves this objective by converting programmer-prepared symbolically coded instructions, directives, and data (the source program) into their binary machine language equivalents (the object program).

DAS processes source programs in two passes. The first pass defines user-designated symbols. The second pass produces an assembly listing and the object program.

Two versions of DAS are available: DAS 8A and DAS MR, described in the following subsections.

1.2.1 DAS 8A Assembler

DAS 8A is a stand-alone program that can operate on a minimum system (8K of memory). It produces absolute object code that can be loaded by the stand-alone binary load/dump program (BLD II).

Because DAS 8A was designed to operate in a restricted environment, it does not provide some of the features described in this book, principally the macro directives (section 4.11). Appropriate error messages are generated if a source program contains statements not recognized by the DAS 8A assembler.

1.2.2 DAS MR Assembler

DAS MR is a macro assembler which produces relocatable object code that can be loaded into any area of memory. It is available either as a free-standing program or as an integral part of the MOS or VORTEX I/VORTEX II operating system. DAS MR includes all of the features described in this book.

1.3 BIBLIOGRAPHY

The following manuals contain information on Sperry Univac hardware and software that would be helpful to the 70 series computer user (the x at the end of each document number is the revision number and can be any digit 0 through 9):

Title	Manual Number
V70 Architecture Reference Manual	98 A 9906 00x
VORTEX I Reference Manual	98 A 9952 10x
VORTEX II Reference Manual	98 A 9952 24x
MOS Manual	98 A 9952 09x

SECTION 2
STATEMENTS

Input to the assembler is supplied by the user in the form of source statements. A statement constitutes one input record and may be in either a position-dependent fixed format or free format.

Each statement can be classified, according to its operation field entry, into one of the following three groups:

a. Computer instruction statement

b. Assembler directive statement

c. Macro call statement

Computer instructions are instructions which are translated into machine-executable code on a one-to-one basis.

Assembler directives are requests to the assembler to perform certain operations during the assembly. These directives may define symbols, reserve and/or initialize data areas, control the listing, and alter the normal processing of statements. The FORM directive allows the user to symbolically define a bit-placement pattern whose name may subsequently appear in the operation field.

A macro call statement represents a predefined block of statements (usually a block of instructions). The macro allows the entire block to be included, with varying parameters, each time the macro name appears in the operation field of a source statement.

This section describes the syntax of composing source statements. A summary of instructions is given in section 3. Assembler directives and macros are described in section 4.

2.1 CHARACTER SET

Source statements are written with the following DAS character set:

Alphabetical characters	ABCDEFGHIJKLMNOPQRSTUVWXYZ	
Numerical Characters	0123456789	
Teletype characters	CR	(carriage return)
	LF	(line feed)
Special characters	+	(plus sign)
	-	(minus sign)
	*	(asterisk)
	/	(slash)
	.	(period)

	(blank)
@	(at sign)
[	(left bracket)
]	(right bracket)
<	(less than)
>	(greater than)
↑	(up arrow)
←	(left arrow)
=	(equal sign)
,	(comma)
'	(prime)
(	(left parenthesis)
)	(right parenthesis)
/	(backslash)
!	(exclamation point)
"	(quotation mark)
#	(pound sign)
%	(percent sign)
&	(ampersand)
:	(colon)
;	(semicolon)
?	(question mark)
$	(dollar sign)

In addition, any of the 128 ASCII characters (see appendix B) may be used anywhere that characters appear between paired apostrophes or brackets, in comments, literals, and in instruction operands.

2.2 STATEMENT FORMAT

A DAS source program consists of a sequence of source statements. Each source statement is input as one record. A punched card is one record, as is one line punched to paper tape and terminated by a carriage return and line feed.

A source statement may contain a maximum of 80 characters. If a source record contains more than 80 characters, then the record is truncated to 80 characters. If a record contains less than 80 characters, the assembler supplies blank characters to fill out 80 character positions. If an assembler source record is completely blank, the source record is ignored by the assembler.

Each source statement comprises a combination of label, operation, variable, and comment fields, depending on the requirements of the computer instruction or assembler directive. One computer instruction is generated by each instruction source statement. None, one, or more words of object code may be generated by each assembler directive, depending on the operation and variable field entries. A standard format for DAS source statements, where each field is separated by one or more blanks and begins in a standard line position, is shown in figure 2-1. Alternative formats may be used, prime among them being the use of commas as field separators. A detailed treatment of statement item placement for various input media is given in section 5.

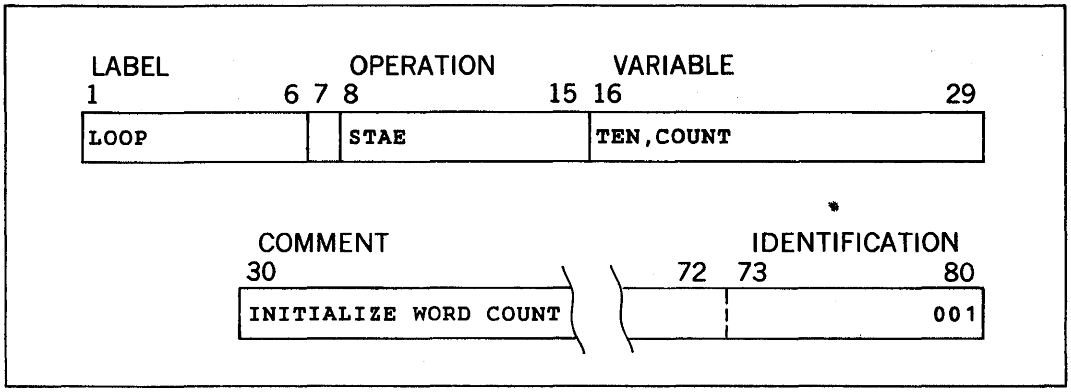


Figure 2-1. Format for Source Statement Records

The fields are described further in the following subsections.

2.2.1 Label Field

The Label Field is the leftmost field on each source statement. It is either blank (no label), or it is used to contain a symbol (section 2.4) created by the programmer. If a label is present, it must begin in character position 1.

For DAS 8A, symbols in the label field comprise one to four alphanumeric characters; for DAS MR there may be from one to six such characters. The first character of a symbol is an alphabetic character, pound sign (#), or dollar sign (the dollar sign and pound sign are used in the Sperry Univac software and should not be used in normal user programs).

Examples

LABEL FIELD 1	8	16	30
ALFRED			valid label (DAS MR)
FRED			valid label (DAS 8A)
F999			valid label
S12X			valid label
Z			valid label
M1			valid label
SUB3			invalid--must begin in position 1
5SYM			invalid--cannot begin with a number
F<=D			invalid characters

An entry in the label field is always optional for instruction statements. It is optional for most assembler directives; however, certain assembler directives (EQU, SET, etc.) require a label field entry.

The programmer generally labels a statement to identify the statement. Symbols in the label field identify program points for reference by other parts of the program. They make a program point or particular numeric value more easily identifiable. The first appearance of a symbol in the label field establishes its identity (most commonly a relative or absolute address) throughout the remainder of the program. A previously established symbol is referenced by placing it in the variable field of the source statement. When the symbol is used, the DAS assembler substitutes the previously assigned value from its symbol table.

Example

```
START   JMPM   FETCH*        Call Fetch routine.
        DAR                  Decrement counter in A.
        JANZ   START         Loop back if A not zero.
```

In this example, the label field is used in the first statement to establish a user symbol for the location of the first statement in a loop. This label, START, is later referenced in the third statement as the return point for another loop iteration.

Label field entries are also used to establish the name of a user-written macro definition (section 4.11).

2.2.2 Operation Field

The Operation Field is to the immediate right of the label field. The entry in this field describes to the assembler the specific type of statement that has been entered, thus determining how it should be processed. Entries in this field are composed of from one to six alphanumeric characters that may describe a machine instruction, assembler directive, or a macro call. An asterisk may follow certain instruction mnemonics to specify indirect addressing (see section 3). It is possible to redefine mnemonics with OPSY assembler directives (section 4.2.1).

An entry in the operation field is always required, and if not supplied by the programmer, will cause an "undefined operation" error code to be generated.

Examples

```
       OPERATION
       FIELD
1      8      16           30
________________________________________
       LDA
       STB*
       JMP
       HLT
       END
```

2.2.3 Variable Field

The Variable Field is to the immediate right of the operation field. The purpose of this field varies according to the requirements of the operation defined by the source statement. The variable field can contain none, one or more symbols, constants or expressions combining symbols and constants. Multiple entries are separated by commas.

The types of entries that may appear in the variable field are described in section 2.3 (constants), section 2.4 (symbols), and section 2.5 (expressions).

Examples

1	8	VARIABLE FIELD 16	30
	LDA	TAB	Load A register with contents of TAB.
	ADDI	16	Add 16 to the A register.
	JMP	PILL	Jump to program location PILL.
	STXE*	SW,2	Store X register indirect, indexed by B.
	LSRA	7	Logical shift right A register 7 bits.
	IAR		Increment A register (has no variable).

2.2.4 Comment Field

An optional comment field follows the variable field in all source statements. This field is used for programming notes. An entire line of comment may be entered if an asterisk is coded in the first position. The assembler ignores all comments in the object code production process, but lists comments and comment lines with the program listing output.

On punched cards, the comment field generally extends from position 30 to position 72. Positions 73 through 80 can be used to sequence cards, simplifying collation if a card deck is accidentally dropped.

Examples

```
1      8       16      COMMENT FIELD
                       30                            72
M1     EQU     40      Master index location.
* SUBROUTINE TO ADD LINK FACTOR
* AND MAINTAIN M1.
SUBL   DATA    0       Subroutine entry.
       LDA*    M1      Fetch word via current index.
       ADDE    LINK    Add link.
       INR     M1      Increment index.
       JMP     *SUBL   Return.
```

2.3 CONSTANTS

A constant is a number, or character string, whose value is specified directly by the programmer in the variable field of a source statement. DAS recognizes decimal integers, octal integers, floating point numbers, and character constants.

In the following descriptions of DAS constants, unsigned numbers are considered positive.

2.3.1 Decimal Integers

A decimal integer is a signed (+ , -) or unsigned string of from one to five decimal digits (0 through 9). The first digit must not be a zero, since a leading zero signifies an octal number.

Decimal integers are converted to a right-justified 15-bit value, in the range -32,768 through +32,767, with the high order bit representing the sign (0 = positive, 1 = negative). Negative numbers are stored in twos complement representation.

Examples

1	Decimal integer +1
20	Decimal integer +20
-3	Decimal integer -3
-9000	Decimal integer -9000
6,099	Invalid--no commas may appear
144000	Invalid--out of range

2.3.2 Octal Integers

An octal integer is a string of from one to six octal digits (0 through 7), preceded by a leading zero. The conversion from octal to binary is straightforward. The number is right-justified in the 16-bit word and may have a range of 0 through 0177777. Octal numbers may optionally be signed (although they normally are not) and will be represented in twos complement form.

Examples

07	Octal constant 7
023	Octal constant 23
0123	Octal constant 123
0677	Octal constant 677
0177777	Octal constant 177777
5612	Invalid octal--no leading zero
07581	Invalid digit

2.3.3 Floating Point Numbers

Floating point numbers may be specified in the following format:

)± integer.fractionE± exponent

where:

)	the right parenthesis indicates a floating point number.
±	is a minus sign (negative number) or an optional plus sign (positive number).
integer	is the integer portion of the number (if any).
.	is the decimal point and must appear.

fraction	is the fractional portion of the number (if any).
E± exponent	is the signed (optional if positive) exponent (if any). The letter "E" may be omitted in the exponent if desired.

At least one digit must appear in the number.

The number is stored in one of the following formats:

Single Precision

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Exponent								Fraction (high)						
0	Fraction (low)														

Double Precision

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	Exponent							
S	Fraction (high)														
0	Fraction (mid)														
0	Fraction (low)														

The exponent is represented in an excess 128 format so that the smallest exponent representable contains all zeros. An exponent field containing 128 (0200) corresponds to an exponent value of 0. The largest exponent representable contains all ones.

The fraction is expressed in a modified sign-magnitude format. Rather than inverting the sign bit for negative numbers, the complete word in which the sign appears is inverted. In single precision, this inverts the exponent, the sign, and the high 7 bits of the fraction. In double precision, the sign and the high 15 bits of the fraction are inverted.

The number is zero represented by all zeros. All other numbers are normalized.

Examples

)5.5	The real number 5.5 (five and a half)
)60.00079	The real number 60.00079
)6. + 10	The real number 60000000000.
)09.E-2	The real number .09
).1E-12	The real number .0000000000001
)-4. +20	The real number -400000000000000000000.
16.E2	Invalid--no right parenthesis.
)16E2	Invalid--no decimal point.
)E2	Invalid--no digit.

2.3.4 Character Constants

A character constant consists of one, two, or more ASCII characters enclosed by primes ('). Any of the 128 ASCII characters may appear in a character term. To code a prime character in DAS MR, use two primes in succession; this cannot be done in DAS 8A, however. Note that blanks are also recognized as characters.

When a single alpha constant is defined by the DATA directive (section 4.4.1), DAS MR left-justifies it in the field and fills the remaining positions with blanks. In other DAS MR and all DAS 8A statements, a single alpha constant is right justified with leading zeros.

Examples

`'STRING'`	Valid character constant.
`'THIS'`	Valid character constant.
`'IS'`	Valid character constant.
`'A'`	1-character constant: = 'A ' in DAS MR, = '0A' in DAS 8A.
`'I CAN''T'`	(DAS MR only)--coded as I CAN'T.
`MMM`	Invalid--surrounding primes missing.

2.3.5 Address Constants

An address constant is a symbol, numer, or expression which may be enclosed in parentheses. It generates a 15-bit direct address (bit 15 = 0).

Examples:

A Address constant
(31)

where A is an address symbol whose value is taken from the symbol table by DAS.

2.3.6 Indirect Address Constant

An indirect address constant is an address constant enclosed in parentheses followed by an asterisk. It generates a 15-bit indirect address (bit 15 = 1).

Examples:

(A+2)* (3)* (A)*

2.3.7 Literals

A literal term or simply, literal, is a constant or expression preceded by an equal sign (=). A literal represents data, rather than an address of data. The appearance of a literal directs the assembler to assemble the data specified in the literal, store this data in an assembler-maintained literal pool, and assemble the address of the data into the current instruction. The literal pool is assigned addresses starting with the value of the literal's location counter when the END directive is processed. Duplicate values are discarded in the literal pool. In general, literals can be used whenever an address is permitted in the variable field.

NOTE

> The literal pool may not be assembled into COMMON areas. Any attempt to place literals into COMMON areas is flagged as an error and the mode of the location counter is changed to program relocatable.

Literals may contain undefined symbols, although use of undefined symbols in literals may cause extraneous words to be allocated within the literal pool.

The use of literal terms allows the programmer to both define and reference a constant word in the same machine instruction statement.

Examples

LDA	=5	Load A register with the constant 5. The value 5 is placed in the literal pool, and its address (in the pool) coded in the LDA instruction.
ADD	=255	Add the value 255 to the A register. The value 255 is placed in the literal pool, and its address coded in the ADD instruction.
ORA . . . ERA	=07077 =07077	Inclusive OR with the A register. The indicated value is placed in the literal pool. For the ERA (Exclusive OR instruction) the same literal pool location is addressed, thus minimizing storage required for the mask word.

2.4 EXPRESSIONS

An expression is a single constant, a single symbol, or any combination of constants and symbols connected by operators. Operators are described in section 2.4.1.

A discussion of multi-term expression evaluation is given in section 2.4.2 (expression evaluation), section 2.4.3 (address expressions), and section 2.4.4 (mode determination). Section 2.4.5 describes literals.

2.4.1 Operators

The following operators are allowed in expressions:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division

Arithmetic operations always involve all 16 bits of the computer words, and are performed from left to right, with multiplication and division occurring before addition and subtraction. Thus, A + B/C * D in DAS is equivalent to A + (B/C) * D in conventional notation.

The rules for coding expressions are:

a. An expression cannot contain two terms or two operators in succession.

b. An expression with a leading minus sign (-) is evaluated as though a zero preceded the minus sign.

c. An expression with a leading plus sign (+) is evaluated as though a zero preceded the plus sign.

d. A multi-term expression cannot contain an external symbol. If it does, an "invalid relocation" error message is printed.

e. Character constants used in mulit-term expressions may contain only one or two characters.

Examples

`A+1`	Valid expression
`'A'+1`	Valid expression
`'A'-'B'`	Valid expression
`6443/2`	Valid expression (evaluates to 3221)
`-1*2`	Valid expression (evaluates to -2)
`10/5*2`	Valid expression (evaluates to 4)
`6+6+6-OMS`	Valid expression (evaluates to 18 minus the value of OMS)
`'A'++'B'`	Invalid--adjacent operators
`'ASM'+2`	Invalid--contains a long character string.

2.4.2 Expression Evaluation

A single-term expression takes on the value of the term involved.

A multi-term expression is reduced to a single value, as follows:

a. Each term is evaluated.

b. Arithmetic operations are performed from left to right.

c. Division always yields an integer result; any fractional portion of the result is dropped.

d. Division by zero is permitted and yields a zero result.

Negative values are carried in twos complement form. The value of the expression must be in the range -32,768 to 32,767 or the results may be meaningless.

2.4.3 Address Expressions

In addition to its evaluated numerical value, the relocatability of an expression is determined. The relocatability of an expression depends upon the term(s) in the expression. The expression is absolute if it contains a single absolute value. The expression is relocatable if it contains a single relocatable value. A multi-term expression may be absolute or relocatable.

Absolute and relocatable expressions are derived from the term or combination of terms composing them, and the way in which these terms are combined. Table 2-2 shows, for each arithmetic operation, whether the result is absolute (abso), relocatable (relo), or illegal.

Table 2-2. Arithmetic Operation Results (DAS MR only)

	A = abso B = abso	A = abso B = relo	A = relo B = abso	A = relo B = relo
A + B	abso	relo	relo	illegal
A - B	abso	illegal	relo	abso
A * B	abso	illegal	illegal	illegal
A / B	abso	illegal	illegal	illegal

2.4.3.1 Absolute Expressions

An absolute expression is a constant, an absolute symbol, or any arithmetic combination of absolute terms. An expression may be absolute even though it contains relocatable terms, alone or in combination with absolute terms, under the following conditions:

a. There must be an even number of relocatable terms in the expression and the terms must be paired. Otherwise, an "invalid relocation" error message will result.

b. Each pair of terms must have opposite signs and the same relocatability. (Program, blank COMMON or the same named COMMON). The paired terms do not have to be contiguous.

c. Relocatable terms entering into multiply or divide operations are considered absolute terms, with the same value.

The pairing of relocatable terms with the same relocatability and opposite signs cancels the effect of the relocation, since both symbols would be relocated by the same amount. Thus, the value represented by the paired terms remains constant, regardless of program relocation.

An absolute expression reduces to a single absolute value.

Examples

If A and B are relocatable symbols and X and Y are absolute symbols or terms, the following are absolute expressions:

`X`	abs = abs
`A-B`	rel-rel = abs
`A-B+X`	rel-rel + abs = abs
`X+Y`	abs + abs = abs
`X*Y`	abs*abs = abs
`X/Y`	abs/abs = abs
`A*B`	rel*rel is interpreted as abs*abs = abs (see discussion below under Relocatable Expressions).

2.4.3.2 Relocatable Expressions (DAS MR Only)

A relocatable expression is a relocatable term or a combination of relocatable and absolute terms under the following conditions:

a. There must be an odd number of relocatable terms with the same relocatability.

b. All the relocatable terms but one must be paired (see the description of pairing under ABSOLUTE EXPRESSIONS).

c. The unpaired term must not be directly preceded by a minus sign (-).

If the above conditions are not met, an "invalid relocation" error message will result.

Relocatable terms entering multiply or divide operations are considered absolute terms with the same value. A relocatable expression reduces to a single relocatable value. This value is the value of the expression, with the relocatability attributes of the unpaired relocatable term.

Examples

If A and B are relocatable symbols and X and Y are absolute symbols, the following are relocatable expressions:

`A`	rel = rel
`A+X`	rel + abs = rel
`X+B`	abs + rel = rel
`A-B+A`	rel-rel + rel = rel
`A+2`	rel + abs = rel
`X+B+Y`	abs + rel + abs = rel
`A*B+A`	rel*rel + rel is interpreted as abs*abs + rel = rel

2.4.4 Mode Determination

The mode of an expression is determined by the mode of the symbols in the expression. The mode is determined by the following rules:

a. If the expression contains any mode E or C symbol, the expression is mode E.

b. If the expression contains only mode A symbols, the expression is mode A.

c. If the expression contains mode A and R symbols, the mode of the expression is R if there is an odd number of mode R symbols. Otherwise, the mode of the expression is A.

The following restrictions apply only to DAS MR and to FORTRAN-compatible output assembly with DAS 8A.:

a. No expression can contain symbols of both modes E and C.

b. A mode E expression comprises a single mode E symbol.

c. No mode E, C, or R expression can multiply or divide a mode E or C symbol.

d. No expression can add or substract a mode C and a mode R symbol, or a mode E and a mode R symbol.

e. No expression can add two or more mode E, C, or R symbols.

f. A mode A symbol can be added to or subtracted from a mode C or R symbol.

Examples

The following program code illustrates expression mode determination rules.

```
EEEE    EXT                     Defines mode E.
CCCC    COMN    6               Defines mode C
RTN     ENTR                    Defines a symbol (RTN) as mode R.
TBL     BSS     50              TBL is mode R.
ABL     BSS     'A'+5           ABL is mode R.
LENG    EQU     *-TBL           LENG is mode A (defines area length).
        CALL    EEEE,TBL,LENG
        LDA     *+6             Legal, one-word relative forward.
        LDA     CCCC+6          Illegal, one-word not R or A.
        LDXI    CCCC+6          Legal, two-word instruction.
        LDA     0,1             Legal, loads CCCC+6 in A register.
        .
        .
        .
        DATA    EEEE+4          Illegal, value not zero.
        DATA    CCCC+4          Legal.
        DATA    CCCC+LENG       Legal.
        DATA    TBL+LENG        Legal, mode is R.
```

2.5 SYMBOLS

A symbol is a character or combination of characters used by the programmer to symbolically define instruction addresses, data addresses, general purpose registers, and arbitrary values. Through their use in label fields and in operand fields they provide the programmer with an efficient method to name and reference program elements. The assembler creates a symbol table and assigns to each of the symbols written in the source program a value and a relocation bias (DAS MR only); it also provides indicator flags when required by the program. This relieves the programmer of having to know the absolute address locations of code and data areas.

Symbols are formed from the following three classes of characters:

a. Alphabetic characters: A through Z

b. Numeric characters: 0 through 9

c. Special character: pound sign (#)

A symbol is formed from one to six characters (DAS MR) or one to four characters (DAS 8A) in length, chosen from the preceding classes. The first character must not be numeric. Symbols cannot contain imbedded blanks.

Symbols may be classified as user symbols (section 2.5.1) and assembler-defined symbols (section 2.5.2).

2.5.1 User Symbols

User symbols are defined and used by the programmer to symbolically reference instruction and data area addresses, the general purpose registers, and arbitrary values.

Although it is possible for the user to define user symbols that begin with the pound sign, he should not do so to avoid conflict with V70 series system software, which uses the pound sign.

Examples

`A`	User symbol.
`MAIN`	User symbol.
`BETA11`	User symbol (DAS MR).
`BUFFER`	User symbol (DAS MR).
`READ1`	User symbol (DAS MR).
`CON90`	User symbol (DAS MR).
`128B`	Invalid--first character is numeric.
`CODE1`	Invalid--more than 4 characters (DAS 8A).
`RECORD1`	Invalid--more than 6 characters (DAS MR).
`RCD+A`	Invalid character in symbol.
`IN AREA`	Invalid--contains an imbedded blank character.

2.5.2 Assembler-Defined Symbols

Assembler-defined symbols are of a specialized nature and are used primarily to control the assembly process. They are unique in that they are not defined by the programmer, but by the assembler itself. All symbols that are not assembler-defined symbols must be properly defined by the user in his source program.

2.5.2.1 Operation Field Symbols

All instruction mnemonics and assembler directives appearing in the operation field are predefined by the assembler and control the processing of the source statement.

CAUTION

> DAS assemblers recognize the complete instruction sets of all SPERRY UNIVAC 70 series computers, even when the system on which they operate lacks the hardware for executing a particular instruction. The programmer, therefore, must have a thorough knowledge of the instructions applicable to his system before attempting to assemble a program.

Any other operation symbols are user symbols; these are comprised of OPSY-defined instruction mnemonics (section 4.2.1), FORM-defined symbols (section 4.4.4), and macro call names (section 4.13).

2.5.2.2 Location Counter Symbols

Current Location Counter (*). The assembler maintains a location counter to assign storage addresses to program statements. It is the assembler's equivalent of the computer's program counter. As machine instructions and data areas are assembled, the location counter is incremented to reflect the length of the assembled code or data. Thus, it always contains the address of the next available word.

The location counter also has an associated relocatability mode, either absolute, program relocatable, or named FORTRAN COMMON relocatable. Modification of the current value and mode of the location counter is accomplished with the ORG directive. The location counter is never negative and is always less than 2^{16}.

The programmer can reference the current value of the location counter by using the asterisk (*) character as a term in an operand. The asterisk term represents the word address of the beginning of the current instruction or data area. Use of the asterisk term in a literal address constant results in the assembler using the word address of the instruction containing the literal.

The relocatability mode of the asterisk term--absolute, program relocatable, or named FORTRAN COMMON relocatable--is dependent on the current mode of the location counter.

Examples

```
JMP      *+4      Jump to the location 4 words down.

LDA      *        Load A with the word at the
                  current location counter (i.e.,
                  the "LDA" instruction itself).
```

DAS 8A Location Counters. DAS 8A has five standard location counters that have predefined names, as described in table 2-1. These location counter names may be used in location counter control directives (section 4.3) for controlling the location counter values used during the DAS 8A assembly process. These names have special significance only in the location counter control directives; if used in instruction statements or other directives, they are considered user symbols.

These five location counters are not applicable in DAS MR programs.

2.5.3 Symbol Values

Associated with every symbol is a value. The value is in the range -32,768 through +32,767. This value is substituted in place of the symbol whenever the symbol appears in the variable field of other source statements.

A symbol's value is defined when it appears in the label field of a statement. The value assigned is one of two types:

- For all instruction mnemonics and most assembler directives, the symbol is assigned the value of the current location counter.
- In certain assembler directives, the symbol is assigned the value of the variable field entry; these directives are: EQU, SET, MAX, MIN, OPSY, ORG, LOC, and BEGI. In addition, special purpose symbols are used in the label field for FORM and MAC directives. (All of these directives are described in detail in section 4.)

2.5.4 Address Symbols and Relocatability

2.5.4.1 Relocatability (DAS MR Only)

In addition to having names and values, all symbols are associated with a set of attributes. These attributes describe how the symbol is handled by the assembler.

The most important attribute is that of relocatability. A relocatable program (DAS MR only) is one that has been assembled with its instruction and directive locations assigned in such a manner that it can be loaded and executed anywhere in memory. When such a program is loaded, the beginning memory address is specified, and a value (known as the relocation bias) is added to the addresses of subsequent relocatable instructions. The relocatable loader is used to load a program in any area of memory and modify the addresses as it loads so that the resulting program executes correctly.

Programs can contain absolute addresses, relocatable addresses, or both. Symbols which refer to addresses that will change during program loading are relocatable. Other symbols, such as register numbers or buffer lengths, do not change with program loading and are called absolute symbols. Programs are usually assembled with a zero relocation bias on the first instruction.

The assembler's location counter contains the (relative) address of the instruction or directive currently being executed. The location counter is absolute when it contains the actual address of the instructions, and relocatable when it contains an address relative to the start of the program.

Symbols can be absolute or relocatable. If a symbol is equated to the location counter, it is relocatable if the location counter is relocatable. Otherwise, the symbol is absolute. Expressions (section 2.5), since they contain symbols, can be absolute or relocatable. Constants are always absolute.

At the beginning of each instruction or data word generated by the assembler, the relocatability can be set by the ORG directive. On encountering an ORG directive, the assembler makes the location counter absolute if the corresponding expression is absolute, or relocatable if the corresponding expression is relocatable.

Table 2-1. Standard DAS 8A Location Counters

Counter	Initial Value	Description
COMN	002000	Controls assignment of memory within an interface area common to two or more programs.
IAOR	000200	Control assignment of memory to indirect pointers.
LTOR	001000	Controls assignment of memory to literals.
SYOR	000000	Controls assignment of memory to all system parameters.
(blank)	004000	Used initially and normally by the assembler for memory assignments until/unless overridden by the use of the ORG directive

2.5.4.2 Absolute Symbols

Absolute symbols are those whose values are independent of the execution address. These symbols are used to represent such things as register numbers, fixed memory locations, buffer lengths, or bit masks.

These symbols can be defined in the following two ways:

a. By appearing in a label field when the location counter is in the absolute mode.

b. By being defined as equivalent to some absolute value in directives (EQU, ORG, etc.).

Examples

	ORG	0500	(Specifies absolute address origin.)
START	LDA	VSYS	The label START is assigned an absolute value of 0500.
TEN	EQU	10	The label TEN is assigned an absolute value of 10.

2.5.4.3 Relocatable Symbols (DAS MR Only)

Values of relocatable symbols are dependent upon the execution address of the program. They can represent such things as instruction addresses, data addresses, and addresses of other programs.

Relocatable symbols may be defined in the following ways:

a. By appearing in a label field while the location counter is in the relocatable mode.

b. By being defined as equivalent to some relocatable value in directives (EQU, ORG, etc.)

There are four major types of relocatable symbols:

a. Program relocatable symbols, whose values depend on the program location.

b. Blank COMMON relocatable symbols, whose values depend on the location of FORTRAN blank COMMON.

c. Named COMMON relocatable symbols, whose values depend on FORTRAN named COMMON.

d. External symbols, whose values depend on the location of separately assembled programs.

Examples

```
*NO ORG DIRECTIVE IN DAS MR ASSEMBLES AS RELOCATABLE.
START   LDA     MERF            The label START is assigned
                                a value of relocatable zero.

HERE    EQU     *               Where the program counter is
                                relocatable, assigns the
                                relocatable value to the label
                                HERE.
```

2.5.5 Symbol Modes

Each symbol has one of the following modes assigned by the assembler:

a. External (E)

b. Common (C)

c. Relative (R)

d. Absolute (A)

The mode of a symbol is determined by the following rules:

a. If the symbol is in an EXT directive, the mode is E.

b. If the symbol is defined by a COMN directive, the mode is C.

c. If the symbol is a symbol in a program, or if * is the current location counter value, the mode is R.

d. If the symbol is a number (numerical constant), the mode is A.

e. If the symbol is defined by an EQU, SET, or similar directive, the mode of the symbol is that of the variable field expression in the directive.

Examples

	EXT	EDAT	Symbol EDAT has mode E.
UNIV	COMN	41	Symbol UNIV has mode C.
START	ENTR		Symbol START has mode R (location counter relocatable) or mode A (location counter absolute).
CONS	DATA	1,2,3	Symbol CONS has mode R (location counter relocatable) or mode A (location counter absolute).
TIME	EQU	24	Symbol TIME has mode A.

SECTION 3
INSTRUCTION SUMMARY

For use with DAS, SPERRY UNIVAC 70 series instructions are divided into six categories: types 1 through 5 and multiple register. Tables 3-1 and 3-2 list the characteristics and mnemonics of the instruction types.

A complete list of V70 series instructions, arranged alphabetically by mnemonic, is given in appendix A. The details of the 16-bit configuration of each individual instruction word are given in the applicable system handbook. Also refer to the handbook for a complete description of addressing modes.

Computer instructions have the general format for source statements described in section 2. A label is always optional in instruction statements. In the following descriptions of the individual instruction groups, the field format:

Operation Variable

is used, with the optional label being understood to precede the operation field when used, and the optional comment field to follow the variable field when used. In cases where the variable field contains more than one item or expression, these are always separated by commas. Mandatory elements of the field are in **bold type**, and optional items, in *italic type*.

Table 3-1. Assembler Instruction Type Characteristics

Parameter	Type 1	Type 2	Type 3	Type 4	Type 5	Multiple Register
Words generated	1	2	2	1	2	(Varies with instruction group)
Memory addressed	Yes	Yes*	Yes	No	Yes	
Indirect addressing	Yes	Yes*	Yes	No	Yes	
Indexing	Yes	No	No	No	Yes	
Variable field expressions	1 or 2	1	2	0 or 1	1 to 3	
Microcoding	No	No	Yes	Yes	No	
* Except for immediate instructions.						

INSTRUCTION SUMMARY

Table 3-2. Summary of Assembler Instruction Types

Type 1	Type 2		Type 3	Type 4		Type 5	Multiple Register
ADD	ADDI	JS3N	BT	AOFA	LLRL	ADDE	AD
ANA	ANAI	JS3NM	IME	AOFB	LLSR	ANAE	ADI
DIV	DIVI	JXNZ	JOF	AOFX	LRLA	DIVE	ADR
ERA	ERAI	JXNZM	JIFM	ASLA	LRLB	ERAE	COM
INR	INRI	JXZ	OME	ASLB	LSRA	IJMP	DADD
LDA	JAN	JXZM	SEN	ASRA	LSRB	INRE	DAN
LDB	JANM	LDAI	XIF	ASRB	MERG	JSR	DEC
LDX	JANZ	LDBI		CIA	NOP	LDAE	DER
MUL	JANZM	LDXI		CIAB	OAB	LDBE	DLD
ORA	JAP	MULI		CIB	OAR	LDXE	INC
STA	JAPM	ORAI		COMP	OBR	MULE	JDNZ
STB	JAZ	STAI		CPA	ROF	ORAE	JDZ
STX	JAZM	STBI		CPB	SEL	SRE	JN
SUB	JBNZ	STXI		CPX	SEL2	STAE	LBT
	JBNZM	SUBI		DAR	SOF	STBE	LD
	JBZ	XAN		DBR	SOFA	STXE	LDI
	JBZM	XANZ		DECR	SOFB	SUBE	SB
	JMP	XAP		DXR	SOFX		SBR
	JMPM	XAZ		EXC	TAB		SBT
	JOF	XBNZ		EXC2	TAX		ST
	JOFM	XBZ		HLT	TBA		T
	JOFN	XEC		IAR	TBX		
	JOFNM	XOF		IBR	TSA		
	JSS1	XOFN		INA	TXA		
	JSS2	XS1		INAB	TXB		
	JSS3	XS1N		INB	TZA		
	JS1M	XS2		INCR	TZB		
	JS1N	XS2N		IXR	TZX		
	JS1NM	XS3		LASL	ZERO		
	JS2M	XS3N		LASR			
	JS2N	XXNZ					
	JS2NM	XXZ					
	JS3M						

3.1 TYPE 1 INSTRUCTIONS

An assembler type 1 instruction occupies one computer word and is memory-addressing. It may optionally specify indirect or preindexed addressing.

Assembler type 1 instructions are:

Normal Load/Store	LDA	Load A register
	LDB	Load B register
	LDX	Load X register
	STA	Store A register
	STB	Store B register
	STX	Store X register
Arithmetic	ADD	Add memory to A register
	SUB	Subtract memory from A register
	MUL	Multiply
	DIV	Divide
	INR	Increment memory
Logic	ANA	AND memory and A register
	ORA	Inclusive OR memory and A register
	ERA	Exclusive OR memory and A register

The format of type 1 instructions varies according to the type of addressing, as follows:

Operation	Variable	
xxx	**address**	Direct addressing
xxx*	**address**	Indirect addressing
or		
xxx	**(address)***	
xxx	**incr,i**	Indexed addressing

where:

xxx	is a type 1 instruction mnemonic
address	is an address expression
incr	is an indexing increment, < 0512
i	specifies an index register: $1 = X$, $2 = B$

If the direct form of instruction is used, DAS selects the addressing mode of the generated computer instruction according to the following rules:

a. Direct Addressing: If the specified address is 2047 or below, direct addressing is used.

b. Relative Addressing: If the specified address is above 2047 but not more than 512 and not less than one word beyond the current instruction, the mode of addressing is relative to the program counter.

c. Indirect Addressing: If neither of the preceding conditions for direct or relative addressing is true, an address within the range 0 through 511 (called indirect pointer) is generated and the indirect pointer address will be used in the instruction in the indirect mode.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expressed in parentheses, e.g.:

```
        LDA*        address

        LDA         (address)*
```

NOTE CAUTION BELOW.

The instruction will be coded to address a location in lower core containing the address of the word to be accessed. Indirect addressing to five levels is permitted and is accomplished by setting the high-order bit at the indirect address location(s).

CAUTION

Only the first form should be used in DAS 8A (i.e., LDA*). In the second form (i.e., address)* DAS 8A will force bit 15 to a 1, changing the instruction.

Indexing is specified by two expressions in the variable field. The first is the indexing increment and is less than 512. The second specifies the indexing register: X register = 1, and B register = 2. Preindexing is used. (Type 1 instructions cannot be postindexed.)

Examples

	LDA	0500	Load A register with the contents of memory location 0500. Addressing is direct.
	LDA	*+12	Load A register with the contents of the word 12 locations down from the LDA instruction. Addressing is program counter relative.
	LDA	070000	Load A register with the contents of memory location 070000. An indirect address is generated pointing to a location in lower core containing the address (070000).
	LDA* . . .	TIN	Load A register with the contents of the location whose address is contained at TIN, i.e., load A register with the contents of location 05100. Addressing is indirect.
TIN	DATA	05100	
	LDA* . . .	IND1	This shows an example of multiple indirect addressing to 3 levels. The A register is loaded with the contents of memory location 050.
.			
IND1	DATA	(IND2)*	
IND2	DATA	(IND3)*	
IND3	DATA	050	

LDA	0300,1	Load A register with the contents of the memory address specified by the sum of the X register contents and 0300. Thus, if the X register contains 0200, the operand for this instruction is in memory address 0500.

3.2 TYPE 2 INSTRUCTIONS

An assembler type 2 instruction occupies two consecutive computer words and is memory-addressing. The second word is the address of a jump, jump-and-mark, or execution instruction; or the operand specified by an immediate instruction.

Assembler type 2 instructions are:

Immediate Load/Store	LDAI	Load A register immediate
	LDBI	Load B register immediate
	LDXI	Load X register immediate
	STAI	Store A register immediate
	STBI	Store B register immediate
	STXI	Store X register immediate
Arithmetic	ADDI	Add to A register immediate
	SUBI	Subtract from A register immediate
	MULI	Multiply immediate
	DIVI	Divide immediate
	INRI	Increment immediate
Logic	ANAI	AND immediate
	ORAI	Inclusive OR immediate
	ERAI	Exclusive OR immediate

Jump	**Jump-and-Mark**	**Execute**	
JMP	JMPM	XEC	Unconditionally
JOF	JOFM	XOF	If overflow set
JOFN	JOFNM	XOFN	If overflow not set
JAP	JAPM	XAP	If A register positive
JAN	JANM	XAN	If A register negative
JAZ	JAZM	XAZ	If A register zero
JBZ	JBZM	XBZ	If B register zero
JXZ	JXZM	XXZ	If X register zero
JANZ	JANZM	XANZ	If A register not zero
JBNZ	JBNZM	XBNZ	If B register not zero
JXNZ	JXNZM	XXNZ	If X register not zero
JSS1	JS1M	XS1	If SENSE switch 1 set
JSS2	JS2M	XS2	If SENSE switch 2 set
JSS3	JS3M	XS3	If SENSE switch 3 set
JS1N	JS1NM	XS1N	If SENSE switch 1 not set
JS2N	JS2NM	XS2N	If SENSE switch 2 not set
JS3N	JS3NM	XS3N	If SENSE switch 3 not set

INSTRUCTION SUMMARY

The immediate instructions have the following format:

Operation	**Variable**
xxxI	**value**

where:

xxxI is an immediate instruction mnemonic

value is any expression value

The format of type 2 program control transfer instructions is the same as for type 1 direct or indirect addressing. Since a full word is allocated to the address, the assembler will never need to code an indirect address pointer for the purpose of reaching a specified location otherwise out-of-range. The programmer may code an indirect address. With two-word instructions, indirect addressing is limited to four levels. Type 2 instructions cannot be indexed.

Examples

`LDAI`	`19`	Load A register with the value 19. The value is coded in the second word of the instruction.
`JMP`	`THERE`	Unconditionally jump to the instruction with the label THERE.
`JXNZ*`	`SM`	If the X register is not zero, jump to the instruction whose address is contained in location SM (may be multi-leveled).
`XAZ`	`IMP`	If the A register is zero, execute the instruction at location IMP. In either case, control passes to the instruction following XAZ.

3.3 TYPE 3 INSTRUCTIONS

An assembler type 3 instruction occupies two consecutive computer words and is memory-addressing. It differs from an assembler type 2 instruction in that the variable field contains two expressions instead of one.

Assembler type 3 instructions are:

Jump	JIF	Jump if condition(s) met
	BT	Jump if bit condition met
Jump-and-Mark	JIFM	Jump and mark if condition(s) met
Execution	XIF	Execute if condition(s) met
I/O	SEN	Program sense and jump if true
	IME	Input to memory
	OME	Output from memory

The format of type 3 instructions is as follows:

Operation	Variable	
xxxx	**code,address**	Direct addressing
yyyy*	**code,address**	Indirect addressing
or		
yyyy	**code,(address)***	

where:

xxxx	is any type 3 instruction mnemonic
yyyy	is any type 3 instruction mnemonic except IME or OME
code	is a condition code (see below)
address	is an address expression

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expression in parentheses as described for the type 1 instructions. Note that IME and OME cannot specify indirect addressing.

The code parameter entries are described in detail below.

JIF, JIFM, and XIF Instructions

For the JIF, JIFM, and XIF instructions, the expression code specifies the conditions required for the jump, jump-and-mark, or execution. The conditions are summarized in table 3-3; they are described in detail in the system handbook. Multiple conditions can be specified by setting additional bits.

Table 3-3. JIF/JIFM/XIF Code Conditions

Variable Field	Jump/Execute if:
0001	Overflow indicator is set.
0002	A register contents are positive.
0004	A register contents are negative.
0006	NOT test of specified conditions.
0010	A register contents are zero.
0020	B register contents are zero.
0040	X register contents are zero.
0100	SENSE switch 1 is set.
0200	SENSE switch 2 is set.
0400	SENSE switch 3 is set.

INSTRUCTION SUMMARY

BT Instruction

For the BT instruction, the expression code is a 6-bit value that specifies the register and bit to be tested, in the form:

5	4	3	2	1	0
z	z	b	b	b	b

where:

zz = 00 Specified bit in A register is 1
= 01 Specified bit in B register is 1
= 10 Specified bit in A register is 0
= 11 Specified bit in B register is 0

bbbb specifies the bit to be tested, from bit 0 (low-order bit) to bit 15 (high-order bit)

SEN Instruction

For the SEN instruction, the expression code is a 9-bit value that specifies the device address and I/O function, in the form:

8	7	6	5	4	3	2	1	0
	q				da			

where:

q is a line number (0 to 7)

da is the device address

Standard device addresses are listed in section 3.4.

IME and OME Instructions

For IME and OME instructions, the expression code is the device address.

Examples

`JIF`	`0222,ALFA`	In this example, the next instruction is taken from symbolic address ALFA if the A register contains a positive number (0002), the B register contains zero (0020), and SENSE switch 2 is set (0200); i.e., 0002 + 0020 + 0200 = 0222.

	BT	056,ADDR	In this example the next instruction from symbolic address ADDR is fetched if bit 14 of the A register contents is zero.
	SEN JMP . . .	0101,ADDR *-2	In this example, the next instruction is fetched from symbolic address ADDR if the write register of the Teletype is ready; OME is executed, which outputs the data in symbolic address LOC to the Teletype. Otherwise, the next instruction in sequence (JMP) is executed, which returns the program to the SEN command.
ADDR	OME	01,LOC	

3.4 TYPE 4 INSTRUCTIONS

An assembler type 4 instruction occupies one computer word and does not address memory. These instructions take none or a single variable operand.

Assembler type 4 instructions are:

Register Transfer	TAB	Transfer A register to B register
	TAX	Transfer A register to X register
	TBA	Transfer B register to A register
	TBX	Transfer B register to X register
	TXA	Transfer X register to A register
	TXB	Transfer X register to B register
	TZA	Transfer zeros to A register (clear A)
	TZB	Transfer zeros to B register (clear B)
	TZX	Transfer zeros to X register (clear X)
	TSA	Transfer switches to A register
Register Modification	IAR	Increment A register
	IBR	Increment B register
	IXR	Increment X register
	DAR	Decrement A register
	DBR	Decrement B register
	DXR	Decrement X register
	CPA	Complement A register
	CPB	Complement B register
	CPX	Complement X register
	AOFA	Increment A register if overflow set
	AOFB	Increment B register if overflow set
	AOFX	Increment X register if overflow set
	SOFA	Decrement A register if overflow set
	SOFB	Decrement B register if overflow set
no	SOFX	Decrement X register if overflow set
Control *operand*	NOP	No operation
	ROF	Reset overflow indicator
	SOF	Set overflow indicator
	HLT	Halt

INSTRUCTION SUMMARY

Shift/Rotation	↓	ASRA	Arithmetic shift right A register
		ASRB	Arithmetic shift right B register
		ASLA	Arithmetic shift left A register
	operand	ASLB	Arithmetic shift left B register
		LASR	Long arithmetic shift right
		LASL	Long arithmetic shift left
		LSRA	Logical shift right A register
		LSRB	Logical shift right B register
		LRLA	Logical rotation left A register
		LRLB	Logical rotation left B register
		LLSR	Long logical shift right
		LLRL	Long logical rotation left
Combined Register Transfer/Modification		MERG	Merge source to destination registers
		INCR	Increment source to destination registers
		DECR	Decrement source to destination registers
		COMP	Complement source to destination registers
		ZERO	Zero (clear) registers.
I/O		EXC	External control
		SEL	External control
		EXC2	Auxiliary external control
		SEL2	Auxiliary external control
		CIA	Clear and input to A register
		CIB	Clear and input to B register
		CIAB	Clear and input to A and B registers
		INA	Input to A register
		INB	Input to B register
		INAB	Input to A and B registers
		OAR	Output from A register
		OBR	Output from B register
		OAB	Output from A and B registers

The format of type 4 instructions appears as follows:

Operation	**Variable**	
xxxx		No variable field
yyyy	**expression**	

where:

xxxx is any of the register transfer, register modification, or control instructions (except HLT) listed above. These instructions take no operand.

yyyy is any of the remaining instructions listed above. Theses instructions take one operand.

expression is an expression value

The expression value is described below for each group that uses it.

HLT Instruction

The HLT variable field expression is optional; if present, it becomes the coded value of the instruction (otherwise zero). The HLT number can be displayed from the I register whenever a halt occurs to determine which halt was reached.

Shift Instructions

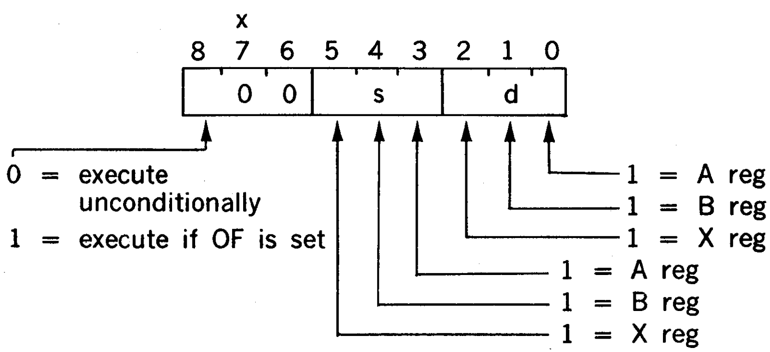
For the shift instructions, the variable field expression is the shift count (31 maximum).

Combined Register Transfer/Modification Instructions

For the combined register transfer/modification instructions, the variable field expression is a number of the form:

Oxsd

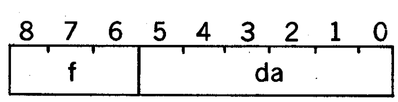
composed as shown below:



For the ZERO instruction, the code must be of the form "0x0d".

I/O Instructions

For EXC, SEL, EXC2, and SEL2, the expression specifies the I/O function and the device address in the form:



where:

f is the control function

da is the device address

INSTRUCTION SUMMARY

For the remainder of the I/O instructions in this group, the expression is the device address only (the I/O function being specified by the mnemonic).

Examples

`HLT`	`066`	Codes an instruction of the operand value that may be displayed when a halt at this location occurs.
`ASLA`	`1`	Arithmetic left shift A register 1 bit (equivalent to multiplying by 2).
`COMP`	`035`	Unconditionally takes the inclusive OR and complements the contents of the A (0010) and B (0020) registers, and places the result in the A (0001) and X (0004) registers. Note that if bit 8 were one in the operand, the instruction would execute only if the overflow indicator is set.
`CIB`	`030`	Clears the B register and loads it from the peripheral specified by device address 030.

Standard device addresses are given in table 3-4.

NOTE

SEL/SEL2 are identical to EXC/EXC2 instructions.

Table 3-4. Standard Device Addresses

Class Code	Addresses	Option or Peripheral
00-07	01-07	Teletype or CRT device
010-017	010-013	Magnetic tape unit
	014	Fixed-head rotating memory
	015	Movable-head rotating memory
	016-017	Movable-head rotating memory
020-027	020,021	First BIC
	022,023	Second BIC
	024,025	Third BIC
	026,027	Fourth BIC
030-037	030	Card reader
	031	Card punch
	032	Digital plotter
	033	Electrostatic plotter
	034	Second paper tape system
	035,036	Line printer
	037	First paper tape system
040-047	040-043	PIM
	044	All PIM enable/disable
	045	MP/PARITY
	047	RTC
050-057	050-053	Special applications, and Digital-to-analog converter through
	054-057	Analog system
060-067	060-067	Digital I/O controller, or Buffered I/O controller
070-077	070-073	Data communications system
	074-076	Relay I/O controller, or Special applications
	077	Computer control panel

3.5 TYPE 5 INSTRUCTIONS

An assembler type 5 instruction occupies two consecutive computer words and is memory-addressing. All of these instructions have indirect addressing as an option. Most can be preindexed or postindexed.

INSTRUCTION SUMMARY

Assembler type 5 instructions are:

Extended Load/Store	LDAE	Load A register extended
	LDBE	Load B register extended
	LDXE	Load X register extended
	STAE	Store A register extended
	STBE	Store B register extended
	STXE	Store X register extended
Arithmetic	ADDE	Add memory to A register extended
	SUBE	Subtract memory from A register extended
	MULE	Multiply extended
	DIVE	Divide extended
	INRE	Increment memory extended
Logical	ANAE	AND memory and A register extended
	ORAE	Inclusive OR memory and A register extended
	ERAE	Exclusive OR memory and A register extended
Jump	IJMP	Indexed jump
	JSR	Jump and set return in index register
	SRE	Skip if register equals memory

These instructions have the following formats:

Operation	**Variable**	
xxxx	**address,*i,post***	Optional indexed addressing
xxxx*	**address,*i,post***	Indirect addressing
or		
xxxx	**(address)*,*i,post***	

where:

address is an address expression

i if present, is an index specification, described further below

post if present, is a postindex specification for all extended addressing instructions.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expression in parentheses as described for the type 1 instructions.

Preindexing is specified as described for the type 1 instructions. Note that IJMP and SRE cannot be preindexed.

Postindexing is specified by three expressions in the variable field. The first expression is the data address, the second specifies the indexing register (X register = 1, and B register = 2), and the third is logically ORed with the instruction word to set bit 7 (which specifies postindexing). The assembler does not check the validity of the third expression; thus, the value 0200 should always be used. There is no purpose to postindexing unless indirect addressing is involved.

Variations in the interpretation of the variable field entries are discussed below.

Extended Instructions

For extended instructions, the variable field may contain one operand (direct addressing), two operands (preindexing), or three operands (postindexing). The instructions may also include indirect addressing.

address	Direct addressing
or	
address,i	Preindexed addressing
or	
address,i,0200	Postindexed addressing

IJMP Instruction

The IJMP instruction may have direct, indirect, and postindexed addressing, i.e., variables of:

address	Direct addressing
or	
address,i	Postindexed addressing

IJMP cannot be preindexed.

JSR Instruction

The JSR instruction, like IJMP, is not preindexed, nor is it postindexed. A variable field of the form:

address,i

is used to specify the jump address and the index register into which the return address is to be placed.

SRE Instruction

For the SRE instruction, the first expression in the variable field is the data address, the second specifies the type of addressing, and the third is logically ORed with the instruction word to control bits 3-5 to specify the register to be compared. The format may be illustrated as:

address,t,reg

where:

address	is the memory location to be compared to the specified register

t	specifies the type of addressing and may be any of the following: = 1 index with X register = 2 index with B register = 7 not indexed
reg	is a register code of the register to be compared, as follows: = 010 A register = 020 B register = 040 X register

Examples:

`LDAE*`	`ADDR,2,0200`	Loads the A register extended, indirect and postindexed with the B register.
`IJMP`	`GO,1`	Indirect jump through location GO, postindexed by the X register.
`JSR`	`MOM,2`	Jump to location MOM and set return in B register.
`SRE`	`ADDR,7,020`	Compares the contents of the B register with the directly addressed word at ADDR, and, if equal, skips the next two locations

3.6 MULTIPLE REGISTER INSTRUCTIONS

It should be noted that from the earliest Sperry Univac 620 software, the assembler syntax uses the convention that the X register is index register 1 and the B register is index register 2. However, the V70 emulation microprograms use hardware register R1 for the B register and hardware register R2 for the X register. The VORTEX DAS Assemblers resolve this by mapping references to register R1 into references to hardware register R2 and vice versa. Thus, for V70 series instructions, references to the X register generate instructions referencing hardware register R2 (X register). Since the programmer is usually indifferent to the hardware register number assigned the X and B registers (except possibly a diagnostic programmer), this should cause no programming problems. If a diagnostic programmer does want to reference a particular hardware register, the register designation in his assembly statements should be written as follows:

a. To reference register R0 (A), write 0.

b. To reference register R1 (B), write 2.

c. To reference register R2 (X), write 1.

d. To reference registers R3 through R7, write 3 through 7, respectively.

NOTE

The multiple register instructions generally require more time for execution; therefore, the standard instruction should be used whenever possible.

3.6.1 Register-To-Memory Instructions

Assembler mnemonics for the register-to-memory instructions are:

AD Add
LD Load
SB Subtract
ST Store

Example

```
LD,0        0300,3
```

Register R0 is loaded with the contents of the memory address specified by the sum of 0300 and the contents of register R3. Thus, if R3 contains 0200, the operand for this instruction is in memory address 0500.

3.6.2 Byte Instructions

Assembler mnemonics for the byte instructions are:

LBT Load Byte
SBT Store Byte

Example

```
SBT         0200,3
```

The contents of the right byte of register R0 are stored at the address specified by the sum of 0200 and the contents of register R3 (shifted right one bit). Thus, if R3 contains 041, the operand is stored in the right byte at address 0220.

3.6.3 Jump-If Instructions

Assembler mnemonics for the jump-if instructions are:

JDNZ	Jump If Double-Precision Register Not Zero
JDZ	Jump If Double-Precision Register Zero
JN	Jump If Register Negative
JNZ	Jump If Register Not Zero
JP	Jump If Register Positive
JZ	Jump If Register Zero

Example

`JZ,3`	`ADDR`	The program jumps to the symbolic address ADDR if register R3 contains zero. If register R3 does not contain zero, the next instruction in sequence is executed.

3.6.4 Double-Precision Instructions

Assembler mnemonics for the double-precision instructions are:

DADD	Double Add
DAN	Double AND
DER	Double Exclusive OR
DLD	Double Load
DOR	Double OR
DST	Double Store
DSUB	Double Subtract

Examples

`DST,4`	`0200`	The contents of double-precision register R4-R5 are stored at the two consecutive memory locations starting at address 0200.
`DST,0`	`0200`	Same as above except register R0-R1 contents are stored.

3.6.5 Immediate Instructions

Assembler mnemonics for the immediate instructions are:

ADI	Add Immediate
LDI	Load Immediate

Example

`ADI,5`	`0642`	The immediate operand value of 0642 is added to the contents of register R5.

3.6.6 Register-To-Register Instructions

Assembler mnemonics for the register-to-register instructions are:

ADR Add Registers
SBR Subtract Registers
T Transfer Registers

Example

`T,3,4`	The contents of register R3 are transferred to register R4.

3.6.7 Single Register Instructions

Assembler mnemonics for the single register instructions are:

COM Complement
DEC Decrement
INC Increment

Example

`INC,3`	The contents of register R3 are incremented by 1.

Example

`ADI,5`	`0642`	The immediate operand value of 0642 is added to the contents of register R5.

3.6.6 Register-To-Register Instructions

Assembler mnemonics for the register-to-register instructions are:

ADR Add Registers
SBR Subtract Registers
T Transfer Registers

Example

`T,3,4`	The contents of register R3 are transferred to register R4.

3.6.7 Single Register Instructions

Assembler mnemonics for the single register instructions are:

COM Complement
DEC Decrement
INC Increment

Example

`INC,3`	The contents of register R3 are incremented by 1.

SECTION 4
ASSEMBLER DIRECTIVES

Assembler directives are requests to the assembler to perform certain operations during program assembly, just as machine instructions are used to request the computer to perform operations during program execution.

Assembler directives are divided into the following functional groups:

- Symbol definition
- Instruction definition
- Location counter control
- Data definition
- Memory reservation
- Conditional assembly
- Assembler control
- Subroutine control
- List and punch control
- Program linkage
- MOS I/O control
- VORTEX I/O control
- Macro definition

Table 4-1 lists the assembler directives by function and shows which directives are recognized by each assembler (DAS 8A and DAS MR).

Assembler directives have the same general format as the computer instructions. In the following descriptions of the individual directives, the field format:

Label Operation Variable

is used, with the optional comment field being understood to follow the variable field when used. In cases where the variable field contains more than one item or expression, these are always separated by commas. Mandatory elements of the directive are in **bold type,** and optional items, in *italic type*.

Table 4-1. Directives Recognized by DAS Assemblers

Function	Directive	DAS 8A	DAS MR
Symbol definition	EQU	Yes	Yes
	SET	Yes	Yes
	MAX	Yes	No
	MIN	Yes	No
Instruction definition	OPSY	Yes	Yes
Location counter control	ORG	Yes	Yes
	LOC	Yes	Yes
	BEGI	Yes	No
	USE	Yes	No
Data definition	DATA	Yes	Yes
	PZE	Yes	Yes
	MZE	Yes	Yes
	FORM	Yes	Yes
Memory reservation	BSS	Yes	Yes
	BES	Yes	Yes
	DUP	Yes	Yes
Conditional assembly	IFT	Yes	Yes
	IFF	Yes	Yes
	GOTO	Yes	Yes
	CONT	Yes	Yes
	NULL	Yes	Yes
Assembler control	MORE	Yes	No
	END	Yes	Yes
Subroutine control	ENTR	Yes	Yes
	RETU*	Yes	Yes
	CALL	Yes	Yes
List and punch control	LIST	Yes	No
	NLIS	Yes	No
	SMRY	Yes	Yes
	DETL	Yes	Yes
	PUNC	Yes	No
	NPUN	Yes	No
	SPAC	Yes	Yes
	EJEC	Yes	Yes
Program linkage	NAME	Yes	Yes
	EXT	Yes	Yes
	COMN	Yes	Yes

Table 4-1. Directives Recognized by DAS Assemblers *(continued)*

Function	Directive	DAS 8A	DAS MR
Macro definition	MAC EMAC	No No	Yes Yes
MOS I/O control	Applicable to DAS MR only; refer to the MOS Reference Manual.		
VORTEX I/O control	Applicable to DAS MR only; refer to the VORTEX I or VORTEX II Reference Manual.		
VORTEX EXEC requests	Applicable to DAS MR only; refer to the VORTEX I or VORTEX II Reference Manual.		

4.1 SYMBOL DEFINITION DIRECTIVES

Symbol definition directives are used to assign values, specified in the variable field, to symbols specified in the label field.

4.1.1 EQU Directive

The EQU directive assigns a value to a symbol. Once assigned by an EQU directive, the value cannot be changed elsewhere in the program.

This directive has the following format:

Label	Operation	Variable
symbol	**EQU**	**expression**

where:

symbol is a symbol which must be present.

expression is any valid expression.

The assembler places the symbol in the symbol table and assigns it the value of the expression. If the symbol has already been entered in the symbol table, DAS outputs an error message, and the expression replaces the value in the symbol table. If a symbol is used as the variable field expression, it must have been previously defined.

Examples

```
AID      EQU      076000
```
AID is assigned the value 076000.

```
X        EQU      1
```
X is assigned the value 1.

```
B         EQU     2+10/5      B is assigned the value 4.

ADDR      EQU     0500        ADDR is assigned the (absolute)
                              value 0500.

ADRS      EQU     *           ADRS is assigned the value
                              of the current location counter
                              (absolute or relocatable).

BAM       EQU     SAD-*+1     BAM is assigned the expression
                              evaluation (absolute or relocatable).

NUM       EQU     22          Double definition (*DD)--two
          .                   equate statements with the same
          .                   label should not appear in the
          .                   same program.  If they do, the
NUM       EQU     14          symbol table will contain the
                              last value used.
```

4.1.2 SET Directive

The SET directive operates the same as EQU except that a symbol may be defined without error.

This directive has the following format:

Label	Operation	Variable
symbol	SET	expression

where:

symbol is a symbol which must be present.

expression is any valid expression.

Examples

```
MOND      SET     400         Assign value of 400 to MOND;
          .                   for subsequent statements,
          .                   MOND has a value of 400.
          .
MOND      SET     500         Assign value of 500 to MOND;
          .                   for subsequent statements,
          .                   MOND has a value of 500.
          .
```

4.1.3 MAX Directive (DAS 8A Only)

The MAX directive assigns the largest (maximum) algebraic value among a string of values to a symbol.

This directive has the following format:

Label	Operation	Variable
symbol	MAX	expression,*expression(s)*

where

symbol	is a symbol which must be present
expression	is any valid expression. The field may contain multiple expressions, separated by commas.

The assembler assigns the largest algebraic value found among the expressions to the symbol. If a symbol is used as a variable field expression, it must have been previously defined. The value of the symbol may be redefined, if desired, via the SET directive.

Examples

`MOST`	`MAX`	`1,2,3,4,5`	Assigns the value 5 to MOST.
`SYM`	`MAX`	`HARRY,JOE,3`	Assigns to SYM the value of the symbol HARRY, the value of the symbol JOE, or 3, depending on which has the highest value. Both symbols must have been previously defined.

4.1.4 MIN Directive (DAS 8A Only)

The MIN directive assigns the smallest (minimum) algebraic value among a string of values to a symbol.

This directive has the following format:

Label	Operation	Variable
symbol	MIN	expression,*expression(s)*

where:

symbol	is a symbol which must be present.
expression	is any valid expression. The field may contain multiple expressions, separated by commas.

MIN is the same as MAX, except that the symbol is assigned the smallest algebraic value found among the expressions.

Examples

```
TRV      MIN      50000        Assigns the value 50000 to TRV.

IN       EQU      10
IOB      EQU      2+10/2*6
MAPN     MIN      IN,10,IOB    Assigns the value 10 to MAPN
                               (note that both label IN and
                               constant 10 have this value).
```

4.2 INSTRUCTION DEFINITION DIRECTIVE

4.2.1 OPSY Directive

The OPSY directive allows the user to optionally define his own mnemonic names for instructions.

This directive has the following format:

Label	Operation	Variable
symbol	**OPSY**	**mnemonic**

where:

symbol is a symbol which must be present.

mnemonic is any standard instruction mnemonic.

The assembler makes the symbol a mnemonic name with the same definition as the variable field mnemonic.

Examples

```
CLA      OPSY     LDA          Define CLA as equivalent to
         LDA      0300         LDA mnemonic; in subsequent
         CLA      0300         program statements, CLA and
                               LDA may be used interchangeably
                               as the "Load A register"
                               instruction mnemonic.

J123     OPSY     JIF,0700     Invalid--variable field must
                               contain only a standard instruction
                               mnemonic.
```

4.3 LOCATION COUNTER CONTROL DIRECTIVES

Location counter control directives control the program location counter(s), which control memory area assignments and always point to the next available word.

DAS 8A Location Counter Control. DAS 8A recognizes directives to modify or preset the values of any of its location counters (refer to table 2-1). In addition, up to eight other location counters can be created, thus providing the possibility of constructing complex relocation and overlay programs within a single assembly.

There are no user-created location counters at the beginning of an assembly. The assembler uses three location counters for program location assignment. Thus, IAOR (indirect pointer assignments) and LTOR (literal assignments) are always in used, as is a third counter used to assign locations to generated instructions and data. The blank location counter performs this task until the USE directive specifies another counter.

In a straightforward program using only one location counter, the ORG and LOC directives completely control the counter.

DAS MR Location Counter Control. DAS MR utilizes only one location counter. This location counter normally has a relocation bias of zero. DAS MR is most commonly used with an operating system and a relocating loader. Normally DAS MR programs are relocatable, and therefore location counter control should not be used.

The ORG directive may be used in DAS MR to change the current location counter value (relocatable or absolute). The LOC directive may be used in DAS MR for assembly of programs that are to be moved under program control. Attempts to use ORG or LOC with DAS MR programs to be run under the operating system should be done with care so as not to overlay any system tasks.

4.3.1 ORG Directive

The ORG directive is used to specify the beginning location counter value.

This directive has the following format:

Label	**Operation**	**Variable**
symbol	**ORG**	**expression**

where:

symbol is an optional user symbol.

expression is an address expression.

The assembler sets the location counter currently in use to the value of the expression. If a symbol is present in the label field, it is also set to the value of the expression (note that this is the current location counter value also).

Any symbol used as the variable field expression must have been previously defined.

For DAS MR, the address origin defaults to relocatable zero if no ORG directive is given. For DAS 8A, it defaults to absolute 04000 if no ORG directive is given.

Example

The left-hand column below shows the value of the location counter at each program statement when origined as shown.

Location Counter

```
05000           ORG     05000       Origin at 05000.
05000   STRT    LDA     A
05001           ADD     C
05002           SUB     D
05003           JMP     AID
05004
05005   A       DATA    5
05006   C       DATA    4
05007   D       DATA    3
        AID     EQU     076000
                END
```

4.3.2 LOC Directive

The LOC directive is used to assemble a block of program code that is to be relocated during program execution.

This directive has the following format:

Label	Operation	Variable
symbol	**LOC**	**expression**

where:

symbol is an optional user symbol.

expression is an address expression.

LOC is used if the data and instructions following this LOC address are to be moved to the LOC address by the object program before executing the moved block, i.e., to keep a block of data or instructions undisturbed by assembly. Data or instructions following LOC are generated as if an ORG directive had changed the current location counter value. However, this value is not actually changed.

The location counter used for coding the block is specified by the expression. If a symbol is present in the label field, it is also set to the value of the expression.

Any symbol used as a variable field expression must have been previously defined. LOC cannot be used in a relocatable program.

Example

The following program code illustrates the use of the LOC directive on the program counter values, as shown in the left-hand column.

```
Location
Counter  Contents
003000                  ORG    03000       Origin at 03000.
003000   010001   A     LDA    1           Instructions assembled
003001   120002         ADD    2           from 03000.
003002   140003         SUB    3
003003   001000         JMP    C           Last address must jump.
003004   003014
         003005   ENDA  EQU    *           ENDA = 03005.
000500            B     LOC    0500        Set assemble-origin at 0500.
000500   000001         DATA   1           These data or instructions
000501   000002         DATA   2           will be assembled for run-
000502   000003         DATA   3           ning at location 0500.  They
000503   000004         DATA   4           will be loaded into core at
000504   000005         DATA   5           locations ENDA plus.  You
000505   000006         DATA   6           must move them to location
000506   000007         DATA   7           0500 before running.
003014            C     ORG    ENDA+*-B
003014   000010         DATA   8           This is the next available
003015   000011         DATA   9           location after program B.
                        END
```

4.3.3 BEGI Directive (DAS 8A Only)

The BEGI directive may be used in DAS 8A programs to define an initial value for any of the location counters.

This directive has the following format:

Label	Operation	Variable
symbol	BEGI	expression

where:

symbol is COMN, IAOR, LTOR, or SYOR (see table 2-1); or a user symbol to create a new location counter.

expression is an address expression.

BEGI creates a new location counter, or redefines the value of any location counter before the counter has been used. Up to eight user location counters may be created. BEGI gives the new or redefined location counter the value of the expression, but has no effect on the current location counter.

BEGI is used to define initial values only. It cannot redefine the value of any location counter that has already been used for location assignment.

Any symbol used as a variable field expression must have been previously defined.

Examples

```
IAOR      BEGI      050
```
Redefine standard counter IAOR to begin at location 050.

```
LTOR      BEGI      075       Redefine standard counter
                              LTOR to begin at location
                              075.

UCNT      BEGI      06500     Create a user location counter
                              called UCNT.
```

4.3.4 USE Directive (DAS 8A Only)

The USE directive activates a specified location counter.

This directive has the following format:

Label	Operation	Variable
(none)	**USE**	**counter**

where:

counter is a blank, COMN, or SYOR (see table 2-1); PREV; or a user-created location counter label.

The USE directive causes the assembler to switch to the current value of the indicated location counter for assembly of subsequent source statements. If PREV is given, the previously used location counter is recalled, with the restriction that only the last-used counter can be so recalled.

Examples

```
USE       COMN      Switch to COMMON location counter.

USE                 Switch to standard location counter.

USE       SYOR      Switch to system location counter.
LDA*      *         (Loads a system parameter.)

USE       COMN
.
.
.
USE       SYOR
.
.
.
USE       PREV      Switch back to COMN location
                    counter.
```

4.4 DATA DEFINITION DIRECTIVES

Data definition directives allow the user to create words of data as part of his source program.

4.4.1 DATA Directive

The DATA directive generates one or more words of data that are output with the object program code.

This directive has the following format:

Label	Operation	Variable
symbol	**DATA**	**expression**,*expression(s)*

where:

symbol	if present, is assigned the value of the current location counter.
expression	is any valid expression.

DATA generates data words with the values specified by the expression(s) in the variable field. DATA assigns the symbol, if used, to the memory address of the first generated word. In the absence of a symbol, an unlabeled block of data is generated.

Examples

D	DATA	5	Creates data word of value 5 and assigns the current location counter value to the symbol D.
	DATA	FF	Creates data word of the value of symbol FF (absolute or relocatable).
	DATA	'COMMENT'	Creates 4 data words of 2 ASCII character bytes per word.
	DATA	D-5	Creates data word of the value of the expression (absolute or relocatable).
	DATA	1+2	Creates data word of value 3.
	DATA	1	Creates data word of value 1.

Figure 4-1 shows a source listing to illustrate the object code generated by the above data expressions. The first column shows the location counter (beginning at relocatable zero), and the second column shows the object code generated. Refer to section 5 for a detailed description of the source listing.

```
005000                 1        ORG     05000
005000 000005 A        2 D      DATA    5,FF,'COMMENT',D=5,1+2,1
005001 005011 A
005002 141717 A
005003 146715 A
005004 142716 A
005005 152240 A
005006 004773 A
005007 000003 A
005010 000001 A
005011 017000 I        3 FF     LDA     D
                       4        END
```

Figure 4-1. Sample DATA Directive Usage

4.4.2 PZE Directive

The PZE directive can be used to generate positive-only data words.

This directive has the following format:

Label	Operation	Variable
symbol	**PZE**	**expression,***expression(s)*

where:

symbol if present, is assigned the value of the current location counter.

expression is any valid expression.

PZE is similar to DATA except that the sign bit of the generated data word is always forced to zero (positive).

Examples

Figure 4-2 shows a source listing illustrating data words (in the second column) generated by the PZE directive. Note that the sign bit (high-order bit) is always zero, contrasted to the DATA directive generations.

```
006000                  1     ORG     06000
006000 177777 A         2     DATA    -1,-2,7,'AB',0106612
006001 177776 A
006002 000007 A
006003 140702 A
006004 106612 A
006005 077777 A         3     PZE     -1,-2,7,'AB',0106612
006006 077776 A
006007 000007 A
006010 040702 A
006011 006612 A
                        4     END
```

Figure 4-2. Sample PZE Directive Usage

4.4.3 MZE Directive

The MZE directive can be used to generate negative-only data words.

This directive has the following format:

Label	Operation	Variable
symbol	**MZE**	**expression**,*expression(s)*

where:

symbol — if present, is assigned the current location counter value.

expression — is any valid expression.

MZE is similar to DATA except that the sign bit of the generated data word is always forced to one (negative).

Examples

Figure 4-3 shows a source listing illustrating the use of MZE.

```
007000                  1     ORG     07000
007000 100001 A         2     MZE     1,,2,06612
007001 100000 A
007002 100002 A
007003 106612 A
                        3     END
```

Figure 4-3. Sample MZE Directive Usage

4.4.4 FORM Directive

The FORM directive specifies the format of a bit configuration of a data word.

This directive has the following format:

Label	Operation	Variable
symbol	**FORM**	**term,***term(s)*

where:

symbol is a user symbol.

term is an absolute expression.

The symbol is the name of the format. The terms specify the length in bits of each field in the generated data word, where the sum of their values is from one to the number of bits in the computer word.

FORM is ignored if there are any errors in the variable field, except that an error is flagged when a term cannot be represented in the number of bits specified when FORM is applied (by placing its name in the operation field of a symbolic source statement) to another statement. A FORM symbol can be redefined.

Examples

Figure 4-4 shows sample usage of the FORM directive.

a. Without error:

	Label	Operation	Variable
	1 BYTE	FORM	8,8
	2 BCD	FORM	4,4,4,4
	3 PTAB	FORM	1,2,3,4
	4 ABC	FORM	6,2,8
000000 014701 A	5	ABC	2*3,1,'A'
000001 106612 A	6	BYTE	0215,0212

b. With error:

	Label	Operation	Variable
000002 000005 A	7	PTAB	2,4,5
*SZ			
*SZ			
	8	END	

Figure 4-4. Sample FORM Directive Usage

4.5 MEMORY RESERVATION DIRECTIVES

Memory reservation directives control the reservation of memory addresses and areas.

4.5.1 BSS Directive

The BSS directive is used to reserve a block of memory locations for use by the program during its execution.

This directive has the following format:

Label	Operation	Variable
symbol	**BSS**	**expression**

where:

symbol if present, is assigned the current location counter value.

expression is an absolute expression.

BSS reserves a block of memory addresses by increasing the value of the current location counter the amount indicated by the expression. The symbol, if used, is assigned the value of the counter prior to such an increase, thus referencing the starting address of the reserved block.

If the variable field expression value is zero, the symbol is assigned the next available address (i.e., BSS 0 = BSS 1).

Examples

B	BSS	050	Reserve a block of 050 words and assign the beginning location address to B. On completion, the location counter will be at B+050. The locations can be accessed as B, B+1, B+2,..., B+047.
MO	BSS	1	These three statements reserve 3 words of storage, each separately labeled.
MP	BSS	1	
MQ	BSS	1	

4.5.2 BES Directive

The BES directive, like BSS, is used to reserve a block of memory locations.

This directive has the following format:

Label	Operation	Variable
symbol	**BES**	**expression**

where:

symbol	if present, is assigned the current location counter value.
expression	is an absolute expression.

The BES directive is similar to BSS, except that if there is a symbol it is assigned to the address one less than the incremented location counter.

If the variable field expression is zero, the symbol is assigned the last address used (i.e., BES 0 has no effect).

Example

```
B       BES       050
```

Same as BSS above, except that the label B is assigned a value of the end of the block. Thus, the locations can be accessed as B-1, B-2, B-3,..., B-047.

4.5.3 DUP Directive

The DUP directive can be used to duplicate source statements input only once.

This directive has the following format:

Label	**Operation**	**Variable**
symbol	**DUP**	**n**,*m*

where:

symbol	if present, is assigned the current location counter value.
n	is a constant that specifies the duplication count.
m	if present, is a constant that specifies the source statement count for duplication. If omitted, it defaults to one.

DUP duplicates source statements that follow the DUP directive. An n-only format duplicates the next source statement the number of times specified by n. An n,m format duplicates the next 1, 2, or 3 source statements (the number of which is specified by m) the number of times specified by n, which $m \leq 3$ and $n \leq 32,767$. If n or m is zero, it is treated as if it were a one.

A DUP statement may not appear within the range of another DUP statement. The statement(s) being duplicated should not contain any labels, as the labels will be duplicated also and a "double definition" (*DD) diagnostic will result.

Examples

```
B    DUP     3       Duplicate the next statement
     ADD     3       (the ADD instruction) three
C    EQU     *       times.

B    DUP     2,2     Duplicate the next 2 statements
     ADD     3       (the ADD instructions) two
     ADD     4       times.
C    EQU     *
```

Complete source listings for these two examples are shown in figure 4-5. Note the duplications.

Example 1

```
004000                      1      ORG    04000
        004000 A            2 A    EQU    *
                            3 B    DUP    3
004000  120003 A            4      ADD    3
004001  120003 A            4      ADD    3
004002  120003 A            4      ADD    3
        004003 A            5 C    EQU    *
                            6      END
```

Example 2

```
        000000 R            1 A    EQU    *
                            2 B    DUP    2,2
000000  120003 A            3      ADD    3
000001  120004 A            4      ADD    4
000002  120003 A            3      ADD    3
000003  120004 A            4      ADD    4
        000004 R            5 C    EQU    *
                            6      END
```

Figure 4-5. Sample DUP Directive Usage

4.6 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives assemble portions of the program according to the conditions specified in the variable fields.

4.6.1 IFT Directive

The IFT directive assembles the next source statement if the specified relationships are true.

This directive has the following format:

Label	Operation	Variable
(none)	**IFT**	**expression,***expression(s)*

where:

expression is an absolute expression

IFT assembles the next source statement only if the first expression is less than the second, and the second is less than or equal to the third, i.e.:

IFT	a	for $a \neq 0$
IFT	a,,b	for $a \neq b$
IFT	a,b,b	for $a < b$
IFT	0,a,b	for $0 < a \leq b$

IFT examples are given in section 4.6.5.

4.6.2 IFF Directive

The IFF directive assembles the next source statement if the specified relationships are false.

This directive has the following format:

Label	Operation	Variable
(none)	IFF	**expression**,*expression(s)*

where:

expression is an absolute expression

IFF is similar to IFT (IFT = true) except that IFF (IFF = false) is the logical complement of IFT, i.e.:

IFF	a	for $a = 0$
IFF	a,,b	for $a = b$
IFF	a,b,b	for $a \geq b$
IFF	0,a,b	for $0 \geq a > b$

IFF examples are given in section 4.6.5.

4.6.3 GOTO Directive

The GOTO directive can be used to skip assembly of a block of source statements.

This directive has the following format:

Label	Operation	Variable
(none)	GOTO	symbol symbol, integer integer,

where:

symbol is a user symbol

integer is any integer

a comma following the variable field entry is used to control output listing.

GOTO usually follows an IFF or IFT directive. All source statements between the GOTO and the statement containing the symbol/integer in its label field are skipped, and the instruction so labeled is assembled next. GOTO cannot return to an earlier point in the program.

If the first and third GOTO formats are used, the skipped instructions are listed. If the second and fourth formats (containing a comma after the variable field element) are used, they are not listed. This listing can also be suppressed by a SMRY directive (section 4.9.3).

GOTO examples are given in section 4.6.5.

4.6.4 CONT Directive

The CONT directive may be used in conjunction with GOTO as the destination statement.

This directive has the following format:

Label	Operation	Variable
symbol integer	CONT	(none)

where:

symbol is a user symbol

integer is any integer

CONT provides a target for a previous GOTO directive. The symbol/constant is not entered in the assembler's symbol table.

CONT examples are given in section 4.6.5.

4.6.5 NULL Directive

The NULL directive may be used in conjunction with GOTO as the destination statement.

This directive has the following format:

Label	Operation	Variable
symbol	NULL	(none)

NULL provides a target for a previous GOTO directive with the symbol entered in the symbol table. NULL has the same effect as a BSS directive with a blank variable field.

Examples

The sample program in figure 4-6 illustrates use of the conditional assembly directives.

```
       000022 A   NBIT   EQU    18
                         IFT    NBIT=16
                         GOTO   YYY       18 BITS
                  *
                  * 16 BIT INSTRUCTIONS
                         IFF    NBIT=16
                         GOTO   123       16 BITS
000000 005000 A   YYY    NOP
                  *
                  * 18 BIT INSTRUCTIONS
                  *
000001            123    NULL             ENTER INTO SYMBOL TABLE
                  345    CONT             IGNORE SYMBOL
                         END
```

Figure 4-6. Sample Conditional Assembly Directives Usage

4.7 ASSEMBLER CONTROL DIRECTIVES

Assembler control directives signal the end or continuance of an assembly.

4.7.1 MORE Directive (DAS 8A Only)

The MORE directive is used in DAS 8A assembly when the input medium does not hold all of the source statements at one time.

This directive has the following format:

Label	Operation	Variable
(none)	**MORE**	(none)

MORE halts the assembly process to allow additional source statements to be put in the input device. Assembly resumes when the RUN or START switch on the computer control panel is pressed. MORE is never listed.

4.7.2 END Directive

The END directive signals the end of the source program.

This directive has the following format:

Label	Operation	Variable
(none)	**END**	*expression*

where:

expression is an address expression

END is the last source statement in the program. The expression is the execution address of the program after it has been loaded into the computer. A blank in the variable field yields an execution address of zero.

4.8 SUBROUTINE CONTROL DIRECTIVES

Subroutine control directives create closed subroutines (i.e., internal to the main program) and control their use.

4.8.1 ENTR Directive

The ENTR directive is the first statement in a closed subroutine.

This directive has the following format:

Label	Operation	Variable
symbol	ENTR	(none)

where:

symbol is a user symbol which must be present.

The symbol is used as the name of the subroutine when called. ENTR generates a linkage word of zero in the object program.

Example

The following program listing illustrates use of the ENTR directive as the first statement of a closed subroutine.

```
000002 000000 A     2 TTYW    ENTR
000003 101101 A     3         SEN      0101,*+4
000004 000007 R
000005 001000 A     4         JMP      *-2
000006 000003 R
```

4.8.2 RETU* Directive

The RETU* directive can be used to return from a closed subroutine.

This directive has the following format:

Label	Operation	Variable
symbol	**RETU***	**expression**

where:

symbol if present, is assigned the current location counter value.

expression is an address expression

RETU* returns from a closed subroutine, generating an unconditional indirect jump to the address indicated by the value of the expression.

Example

The following program listing illustrates use of the RETU* directive to return from a closed subroutine.

```
000007 005000 A     5          NOP
000010 001000 A     6          RETU*     TTYW
000011 100002 R
                    7          END
```

4.8.3 CALL Directive

The CALL directive is used to call closed subroutines.

This directive has the following format:

Label	Operation	Variable
symbol	**CALL**	**name,***parameter(s),error(s)*

where:

symbol if present, is assigned the current location counter value.

name is the symbolic name of the subroutine being called.

parameters(s) if present, are one or more data parameters being passed to the subroutine, separated by commas.

error(s) if present, are one or more address expressions, separated by commas, that are to be used by the closed subroutine.

CALL causes the program to jump and mark to the closed subroutine specified by name. The parameter list, if present, is available to the subroutine. The error return list, if present, provides the possibility of returning to locations other than the statement following the CALL statement.

Examples

The sample program calls in figure 4-7 illustrate use of the CALL directive.

Example 1

```
000000 002000 A    1          CALL     TTYW
000001 000002 R
```

Example 2

```
004000             1          ORG      04000
004000 000000 A    2 FUNC     ENTR
                   3 *
                   4 * FUNC WILL HAVE ADDRESS OF PARAMETER X
                   5 * WHEN CALLING THIS SUBROUTINE.
                   6 *
004001 001000 A    7          RETU*    FUNC
004002 104000 A
                   8 *
                   9 *
                  10 *
004003 002000 A   11          CALL     FUNC,X,Y+1,(ERR),(GOOF)*
004004 004000 A
004005 004011 A
004006 004013 A
004007 004013 A
004010 104014 A
                  12 *
                  13 * MAIN BODY OF PROGRAM
                  14 *
004011 000005 A   15 X        DATA     5
004012 000006 A   16 Y        DATA     6
004013 000747 A   17 ERR      DATA     0747
004014 000727 A   18 GOOF     DATA     0727
                  19          END
```

Figure 4-7. Sample CALL Directive Usage

4.9 LIST AND PUNCH CONTROL DIRECTIVES

List and punch control directives control listing and punching during program assembly. They are operative only during the second pass of the assembler, when the object program and listings are produced.

4.9.1 LIST Directive

The LIST directive is used to resume generating a source listing after a list-inhibiting directive has been given.

This directive has the following format:

Label	Operation	Variable
(none)	LIST	(none)

LIST causes the assembler to start or resume output of a source program listing. The assembler normally outputs a list of the source statements. The LIST directive is used to bring the assembler back to this condition when the NLIS directive (section 4.9.2) has been issued to change the listing status.

4.9.2 NLIS Directive

The NLIS directive is used to inhibit the program listing.

This directive has the following format:

Label	Operation	Variable
(none)	NLIS	(none)

NLIS suppresses further listing of the program.

4.9.3 SMRY Directive

The SMRY directive may be used to inhibit listing of conditionally-skipped source statements.

This directive has the following format:

Label	Operation	Variable
(none)	SMRY	(none)

SMRY suppresses the listing of source statements that have been skipped under control of the conditional assembly directives.

4.9.4 DETL Directive

The DETL directive is used to cancel the effect of the SMRY directive.

This directive has the following format:

Label	Operation	Variable
(none)	DETL	(none)

DETL removes the effect of SMRY, i.e., causes listing of all source statements, including those skipped by conditional assembly directives.

4.9.5 PUNC Directive (DAS 8A Only)

The PUNC directive is used in DAS 8A programs to cancel the effect of the NPUN directive.

This directive has the following format:

Label	Operation	Variable
(none)	PUNC	(none)

PUNC causes the assembler to produce a paper tape punched with the object program. The assembler normally outputs such a tape. PUNC returns the assembler to this condition when the NPUN directive (section 4.9.6) changes the punching status.

4.9.6 NPUN Directive (DAS 8A Only)

The NPUN directive may be used to inhibit further punching of the object program to paper tape.

This directive has the following format:

Label	Operation	Variable
(none)	NPUN	(none)

NPUN suppresses further production of paper tape punched with the object program.

4.9.7 SPAC Directive

The SPAC directive can be used to insert blank lines in the source listing.

This directive has the following format:

Label	Operation	Variable
(none)	SPAC	(none)

SPAC causes the listing device to skip a line. The SPAC directive itself is not listed.

4.9.8 EJEC Directive

The EJEC directive causes a page eject.

This directive has the following format:

Label	Operation	Variable
(none)	EJEC	(none)

EJEC causes the listing device to move to the next top of form. The EJEC directive itself is not listed.

4.10 PROGRAM LINKAGE DIRECTIVES

Program linkage directives establish and control links among programs that have been assembled separately but are to be loaded and executed together.

4.10.1 NAME Directive

The NAME directive establishes linkage definition points among separately assembled programs.

This directive has the following format:

Label	Operation	Variable
(none)	NAME	symbol,*symbol*(s)

where:

symbol is any symbolic expression

With the NAME directive, each symbol can then be referenced by other programs. Each symbol also appears in the label field of a symbolic source statement in the body of the program to give it a value. Undefined NAME symbols cause error messages to be output.

Examples

NAME	A	Provide value of symbol A to other programs.
NAME	A,B	Provide values of symbols A and B to other programs.
NAME	EX,WHY,ZEE	Provide values of symbols EX, WHY, and ZEE to other programs.

4.10.2 EXT Directive

The EXT directive allows separately assembled programs to obtain the values of symbols defined in other program NAME directives.

This directive has the following format:

Label	Operation	Variable
label	**EXT**	*symbol(s)*

where:

symbol is a value to be obtained from other programs.

In linking separately assembled programs, EXT declares each symbol not defined within the current program. Each symbol, in both the label and variable fields, is output to the relocatable loader with the address of the last reference to the symbol for the loader to supply the value to the program when the value is known.

If a symbol is not defined within the current program and is not declared in an EXT directive, it is considered undefined and causes an error message output. If a symbol is declared in EXT but not referenced within the current program, it is output to the loader for loading, but no linkage to this program is established. If a symbol is both defined in the program and declared to be external, the EXT declaration is ignored.

Examples

	`EXT`	`AY`	Declare AY to be external.
`BEG`	`EXT`	`BE,SEE`	Declare BE and SEE to be external; the value of BEG is passed to the loader.
	`EXT`	`DEE,EE,FF,GEE`	Declare the indicated symbols to be external.

4.10.3 COMN Directive

The COMN directive defines an area in blank common for use at execution time.

This directive has the following format:

Label	Operation	Variable
symbol	**COMN**	**expression**

where:

symbol if present, is assigned the current location counter value

expression is an absolute expression

COMN allows an assembler program to reference the same blank common area as a FORTRAN program. The common area is cumulative for each use of COMN, i.e., the first COMN defines the base area of the blank common, the second COMN defines an area to be added to the already established base, etc.

Examples

AAA	COMN	3	Allocate 3 words of common, the first word addressable by AAA.
	COMN	6*2	Allocate 12 words of common; if following the above statement, this would be the fourth through sixteenth common locations.
BBB	COMN	9	Allocate 9 words of common, the first word addressable by BBB; if following the above 2 statements, this would be the seventeenth through twenty-fifth locations of common.

4.11 MACRO DEFINITION DIRECTIVES (DAS MR ONLY)

The V70 series macro language is an extension of the V70 assembler language. It provides a convenient way to generate a desired sequence of assembly language statements many times in one or more programs. The macro definition is written only once, and a single macro call statement used each time a programmer wants to generate the desired sequence of statements. This method simplifies the coding of programs, reduces the chance of programming errors, and ensures that standard sequences of statements are used to accomplish desired functions.

Every defined macro is associated with a four- or six-character symbolic name. The defined macro is called when this name appears in the operation field of an assembler source statement.

A Macro Definition is a set of statements that provides the assembler with the symbolic name of the macro and the sequence of statements that is to be generated when the macro is called. Macro definitions start with the MAC directive and are ended with the EMAC directive.

The macro is the assembly equivalent of the execution subroutine. It is defined once and can then be "called" from the program. The macro is an algorithmic statement of a process that can vary according to the arguments supplied. It is assembled with the resultant data inserted into the program at each point of reference, whereas the subroutine executed during execution time appears but once in a program.

4.11.1 MAC Directive (DAS MR Only)

The MAC directive is used to mark the beginning of a macro definition and specify the name of the macro.

This directive has the following format:

Label	Operation	Variable
symbol	MAC	(none)

MAC introduces a macro definition. The symbol is the name of the macro.

The use of the MAC directive is shown in the program example given in section 4.11.3.

4.11.2 EMAC Directive (DAS MR Only)

The EMAC directive is used to signal the end of a macro.

This directive has the following format:

Label	Operation	Variable
(none)	EMAC	(none)

EMAC terminates the definition of a macro.

The use of the EMAC directive is shown in the program example given in section 4.11.3.

4.11.3 Macro Calls

A Macro Call statement is a source program statement with the symbolic name of a defined macro written in the operation field. The assembler generates a sequence of assembly language statements for each occurrence of the same macro call statement. The generated statements are then processed like any other assembly langauge statement.

A macro is called by the appearance of its name in the operation field of a source statement. The variable field of this statement contains expression(s) P(1), P(2).,,,P(n), which are then processed with the values in the table being substituted for the respective values of the expressions in the source statement variable field. For example, if the variable field of the symbolic source statement contains:

2,B,9 + 8, = 63

then within the generated macro P(1) = 2, P(2) = the value of B, P(3) = 17, and P(4) is the address of the value 63. All terms and expressions within the macro-referencing symbolic source statement parameter list are evaluated prior to calling the macro.

If the label field of such a source statement contains a symbol, the symbol is assigned the value and relocatability of the location counter at the time the macro is called but before data generation.

A macro definition can contain references to machine instruction mnemonics or to assembler directives other than DUP. Macros can be nested within macros to a depth limited only by the available memory at assembly time.

Figure 4-8 illustrates the use of macros.

```
                          1 SENSE   MAC                      } Macro
                          2         SEN      P(1),*+4        } Definition
                          3         JMP      *-2             }
                          4         EMAC                     }
                          5         SENSE    0201            - Macro Call
000000 101201 A                                              } Macro
000001 000004 R                                              } Expansion
000002 001000 A                                              }
000003 000000 R                                              }
000004 102501 A           6         CIA      01
                          7         SENSE    0101
000005 101101 A
000006 000011 R
000007 001000 A
000010 000005 R
000011 103101 A           8         OAR      01
                          9         END
```

Figure 4-8. Sample Macro Usage

P(0) can also be accessed by a normal call. P(0) is the first entry in the table formed by the assembler and contains the number of entries in that table. Figure 4-9 shows the output listing obtained by calling P(0).

```
                          1      A    MAC
                          2           DATA    P(0)
                          3           EMAC
000001      000000A       4           A
000002      000001A       5           A       1
000003      000002A       6           A       1,2
000004      000003A       7           A       1,2,3
000005      000004A       8           A       1,2,3,4
000006      000005A       9           A       1,2,3,4,5
                          10          END
```

Figure 4-9. Output Listing Obtained by Calling P(0)

SECTION 5
OPERATING THE ASSEMBLER

DAS MR and DAS 8A are two-pass assemblers that may be scheduled by job central directives. Assembler processing during the two passes is described in section 5.1. Operation of DAS MR under VORTEX I/VORTEX II is described in section 5.2, followed by operation descriptions of DAS MR under MOS, as stand-alone, and of DAS 8A (also stand-alone).

5.1 ASSEMBLER PROCESSING

This section describes the general features of DAS assembler processing. Specific operating procedures and output listing examples for various DAS/operating system combinations are given in section 5.2.

5.1.1 Assembler Input Media

The source program may be input to the assembler on punched cards, paper tape, or any other source input medium. Details regarding source statement field placement are given below.

Fixed Format. Fixed format, normally used with punched cards, used as input to the DAS assemblers contains four fields corresponding to the instruction and directive fields:

a. The label field is in columns 1 through 6. Its use is governed by the requirements of the instruction or directive.

b. The operation field is in columns 8 through 14. It contains the instruction or directive mnemonic. Indirect addressing is specified by an asterisk following the mnemonic.

c. The variable field begins in column 16 and ends with the first blank that is not part of a character string. Its use depends on the instruction or directive. If two or more subfields are present, they are separated by commas.

d. The comment field fills the remainder of the card. If the variable field is blank, the comment field begins in column 17.

An asterisk in column 1 indicates that the entire card contains a comment.

The fixed format is shown in figure 5-1. Note that columns 7 and 15 are always unpunched (blank).

Free Format. Free format (normally used with paper tape) used as input to the DAS assemblers contains source statements of up to 80 characters each (not incuding the carriage return and line feed characters). Each punched statement contains four fields corresponding to the instruction and directive fields. The label, operation and variable fields are separated by commas, and the comment field starts after the first variable field blank that is not part of

A. FIXED FORMAT (STANDARD COLUMNS)

<u>**LABEL**</u>	<u>**OPERATION**</u>	<u>**VARIABLE**</u>	<u>**COMMENT***</u>	<u>**IDENTIFICATION**</u>
1 6	8 14	16 28	30 72	73 80

B. FIXED FORMAT (MINIMUM SPACING)

LABEL OPERATION VARIABLE COMMENT

7 spaces or less

LABEL OPERATION COMMENT

8 spaces or more

C. FREE FORMAT (COMMAS FOR SEPARATORS)

LABEL ,OPERATION ,VARIABLE COMMENT*

,OPERATION ,COMMENT *FORMAT FOR NO LABEL OR VARIABLE FIELD.*

* The comment can start anywhere after a blank following the variable field.

Figure 5-1. Field Placement Summary

a character string. Each statement is terminated by a carriage return (CR) followed by a line feed (LF).

The four fields used when free format input to the DAS assembler is selected are:

a. Label field use is governed by the requirements of the instruction or directive. It is terminated with a comma. If this field is not used, a comma appears as the first character of the source statement.

b. The operation field contains the instruction or directive mnemonic. An asterisk following the mnemonic specifies indirect addressing. This field begins immediately following the label field terminator and is terminated by a comma.

c. The variable field can be blank, or contain one or more subfields separated by commas. It must immediately follow the instruction field terminator (,). Subfields can be voided by using adjacent commas. This field is terminated by a blank that is not part of a character string, or with a CR or LF.

d. The comment field fills the remainder of the statement (from the terminating blank of the variable field to the next CR or LF).

If the first nonblank character of a source statement is an asterisk, the entire statement is a comment.

The free format where commas are used as separators is shown in figure 5-1. Note that any source input may use either free or fixed format.

5.1.2 Pass 1 - Symbol Table

During pass 1, the DAS assembler reads the source program and constructs a symbol table of all symbols appearing in the source program. For each symbol in the table, there is a corresponding value, usually an address in memory. Symbol table capacities are summarized in table 5-1.

Table 5-1. DAS Symbol Table Capacities

Assembler	8K Memory	Greater than 8K Memory
DAS 8A	440	440 + n (800)
DAS MR	20	20 + n (800)
where n = number of 4K memory increments above 8K.		

5.1.3 Pass 2 - Assembler Output

DAS produces a source/object listing of the assembled program, as well as an object program in reloadable format. The object program may be output to any BO device supported by the operating system.

The listing can be obtained in whole or in part as the program is being assembled. The source (symbolic) program and the object (absolute) program are listed side by side on the listing device. This device can be any LO device supported by the operating system.

The listing is output according to the specifications given by the list and punch control directives in the assembly (DAS 8A, DAS MR).

Error analysis during assembly causes error messages (section 5.1.4) to be output on the line following the point of detection.

Figure 5-2 illustrates the format of the output listing. The columns are further described below:

Address	This column shows the current location counter value in octal. It is incremented for each word of object code.
Code	Most entries in this column are words of object code (in octal). The values of symbols assigned via symbol definition directives (EQU, SET, etc.) are also shown in this column but are not part of the object code.
Mode	An indication of the addressing mode, as follows: A Absolute value C Common E Externally defined I Indirect Pointer L Literal Pointer R Relative address value
Line Count (DAS MR only)	The assembler assigns a unique ascending integer number to each non-blank input statement in order of sequence in the input source deck, starting with 1. This statement number is listed in the fourth column, and is used to cross reference error messages to the statements which caused the errors. Statements generated by macro expansions are not assigned a statement number. All statements generated by a DUP directive have the same line number.

Symbolic Source Statement	Reproduces the source statements as input, with additional lines showing directive-duplicated statements and macro expansion space.

Address	Code	Mode	Line Count	Symbolic Source Statement		
014000			1		ORG	014000
014000	000000		2	ABS	ENTR	
014001	001002		3		JAP*	ABS
014002	114000	R				
014003	005211		4		CPA	
014004	001000		5		JMP*	ABS
014005	114000	R				
	000000		6		END	

Figure 5-2. Output Listing Format

5.1.4 Error Messages

The assembler checks source statement syntax during both pass 1 and 2. Detectable errors are listed during pass 2.

The error message appears in the listing line following the statement found to be in error. Each line can hold up to four error messages.

The DAS error codes and their meanings are listed in table 5-2.

Table 5-2. DAS Error Codes

Code	Meaning
*AD	Error in an address expression
*DC	Decimal character in an octal constant
*DD	Illegal redefinition of a symbol or the location counter
*E	Incorrectly formed statement
*EX	Illegally constructed expression
*FA	Floating-point number contains a format error
*IL	First nonblank character of a source statement is invalid (the statement is not processed)

Table 5-2. DAS Error Codes *(continued)*

Code	Meaning
*MA	Inconsistent use of indexing and indirect addressing
*MQ	Missing right quotation mark in character string
*NR	No memory space available for additional entries in assembler tables
*NS	No symbol in the label field of a SET, EQU, MAC, or FORM directive or no symbol in the label or variable field of an OPSY directive, or no symbol in the variable field of a NAME directive.
*OP	Undefined operation field (two No Operation (NOP) instructions are generated in the object program; the remainder of the statement is not processed), or illegal nesting of DUP or MAC directives or DUP of a macro call
*QQ	Illegal use of prime (')
*R	Relocatable item where an absolute item should be defined
*SE	Synchronization error: symbol value in pass 2 is different from that found in pass 1
*SY	Undefined symbol in an expression
*SZ	Expression value too large for a subfield, or a DUP directive specifies that more than three statements are to be assembled (m parameter)
*TF	Undefined or illegal indexing specification
*UC	Undefined character in an arithmetic expression
*UD	Undefined symbol in the variable field of a USE directive

Table 5-2. DAS Error Codes *(continued)*

Code	Meaning
*VF	Instruction contains variable subfields either missing or inconsistent with the instruction type
*XR	Address out of range for an indexing specification
*=	Invalid use of literal
*II	Implicit indirect reference when I parameter is present on the /DASMR directive.

5.2 ASSEMBLER OPERATING PROCEDURES

Since DAS MR operates under MOS or VORTEX and uses the MOS or VORTEX I/O control system, the I/O devices can be defined as required.

DAS MR uses the secondary storage device unit for pass 1 output. It inputs the symbolic source statements from the processor input (PI) logical unit in alphanumeric mode, and outputs them in the same mode on the processor output (PO) logical unit. When DAS MR detects the END directive, it terminates pass 1, returns to the beginning of the source program, and begins pass 2. During pass 2, the source statements are the input from the system scratch (SS) logical unit, a listing is output on the LO unit, and the binary object program is output on the BO unit.

Sections 5.2.1, 5.2.2, and 5.2.3 describe DAS MR operations in different environments. DAS 8A operation is described in section 5.2.4.

5.2.1 DAS MR Operation (VORTEX I/VORTEX II)

The /DASMR directive schedules the DAS MR assembler with the specified options for background operation on priority level 1. It has the general form:

/DASMR,*p(1)*,*p(2)*...,*p(n)*

where:

each *p(n)* — if any, is a single character specifying one of the options shown in table 5-3. The /DASMR directive can contain up to six such parameters in any order.

Table 5-3. DAS MR Options for Background Operation

Parameter	Presence	Absence
B	Suppresses binary object	Output binary object
L	Outputs binary object on GO file	Suppresses output of binary object on GO file
M	Suppresses symbol-table listing	Output symbol-table listing
N	Suppresses source listing	Outputs source listing
E	Assembles multiple register instructions	Flags multiple register instructions with '*OP error'.
I	Flags implicit indirect instructions with '*II error'.	Assembles implicit indirect instructions.

The DAS MR assembler reads source records from the VORTEX PI logical unit on the first pass. The PI unit must be set to the beginning of the source file before the /DASMR directive is executed. This can be done with an /ASSIGN, /SFILE, /REW, or /PFILE directives. A load-and-go operation requires, in addition, an /EXEC directive. Details of the preceding directives are given in the V70 VORTEX I or VORTEX II Operating System Reference Manual.

Shown below is an example for scheduling the DAS MR with no source listing but with the binary object output on the VORTEX logical unit GO file:

```
/JOB,EXAMPLE
/DASMR,N,L,B
```

/JOB (as well as /ENDJOB or /FINI) initializes the GO file to start of file. If BO is assigned to a rotating memory partition, a /PFILE,BO,,BO must precede the /DASMR directive to initialize the file (unless the assembly is part of a stacked job).

DAS MR uses the secondary storage device unit for pass 1 output. It reads a source module from the PI logical unit and outputs it on the PO unit. The source input for pass 2 is entered from the SS logical unit.

When an END statement is encountered, the SS unit is repositioned and reread. During pass 2, the output can be directed to the BO and/or GO units for the object module and the LO unit for the assembly listing. The SS or PO file, which contains a copy of the source module, can be used as input to a subsequent assembly.

DAS MR has a symbol-table area for 175 symbols at five words per symbol. To increase this area, input before the /DASMR directive a /MEM directive where each 512-word block enlarges the capacity of the table by 100 symbols.

A VORTEX II physical record on an RMD is 120 words. Source records on RMD are blocked three 40-word records per VORTEX II physical record, and object modules on RMD are blocked two 60-word modules per record. However, in the case where SI = PI = RMD, records are not blocked but assumed to be one per VORTEX II physical record. When an input file contains more than one source module each new source module must start at a physical record boundary. Unused portions of the last physical record of the previous source modules should be padded with blank records. Proper blocking may be ensured by following the END statement of the previous source module with two blank records.

Figure 5-3 shows the listing output resulting from assembling and executing a sample DAS MR program under VORTEX II.

```
13:26:43   /JOB,SWITCH
13:26:49   /KPMODE,0
13:26:52   /DASMR,L,B
```

Figure 5-3. Example of Assembled and Executed DAS MR Program Under VORTEX Control

```
PAGE     1  08-16-76  SWITCH    VORTEX  DASMR              1326 HOURS

                          1         NAME     SWITCH
          000000 R        2 SWITCH  EQU      *
                          3         EXT      PIFCB,LOFCB
          000001 A        4 X       EQU      1
          000002 A        5 B       EQU      2
          000024 A        6 COUNT   EQU      20             SWITCH COUNT
          000050 A        7 RECL    EQU      COUNT+COUNT    RECORD LENGTH (IN WORDS)
          000004 A        8 PI      EQU      4              PROCESSOR INPUT
          000005 A        9 LO      EQU      5              LISTING OUTPUT
          000000 A       10 WAIT    EQU      0              WAIT FOR IO
          000001 A       11 NOWAIT  EQU      1              IMMEDIATE RETURN
          000001 A       12 ASCII   EQU      1
          000000 R       13 START   EQU      *
                         14         IOLINK   PI,BUFF,RECL
000000    006505 A
000001    000000 E
000002    001404 A
000003    000075 R
000004    000050 A
                         15         IOLINK   LO,CNTRL,RECL+1
000005    006505 A
000006    000001 E
000007    001405 A
000010    000074 R
000011    000051 A
                         16 READ    READ     PIFCB,PI,WAIT,ASCII
000012    006505 A
000013    000000 E
000014    100000 A
000015    010004 A
000016    000000 E
000017    000000 A
000020    000000 A
                         17 READCR  STAT     READ,END,END,END,READCR
```

Figure 5-3. Example of Assembled and Executed DAS MR Program Under VORTEX Control *(continued)*

```
000021 006505 A
000022 000000 E
000023 000012 R
000024 000071 R
000025 000071 R
000026 000071 R
000027 000021 R
000030 006030 A    18          LDXI     COUNT

PAGE     2  08-16-76  SWITCH     VORTEX  DASMR              1326 HOURS

000031 000024 A
       000032 R    19 DOIT     EQU      *
000032 006015 A    20          LDAE     BUFF-1,X          GET A WORD
000033 000074 R
000034 004250 A    21          LRLA     8                 SWITCH BYTES
000035 005244 A    22          CPX                        INVERT POINTER
000036 006025 A    23          LDBE     BUFF+RECL+1,X GET INVERSE WORD
000037 000146 R
000040 006055 A    24          STAE     BUFF+RECL+1,X SAVE ORIGINAL SWITCHED WORD
000041 000146 R
000042 004050 A    25          LRLB     8                 SWITCH BYTES OF INVERSE WORD
000043 005244 A    26          CPX                        RESTORE POINTER
000044 006065 A    27          STBE     BUFF-1,X          SAVE INVERTED INVERSE WORD
000045 000074 R
000046 005344 A    28          DXR                        COUNT DOWN
000047 001046 A    29          JXNZ     DOIT              REPEAT IF MORE
000050 000032 R
                   30 WRITE    WRITE    LOFCB,LO,WAIT,ASCII
000051 006505 A
000052 000013 E
000053 100000 A
000054 010405 A
000055 000000 E
000056 000000 A
```

Figure 5-3. Example of Assembled and Executed DAS MR Program Under VORTEX Control *(continued)*

```
000057 000000 A
                     31 BUSY    STAT     WRITE,END,END,END,BUSY
000060 006505 A
000061 000022 E
000062 000051 R
000063 000071 R
000064 000071 R
000065 000071 R
000066 000060 R
000067 001000 A      32         JMP      READ           READ SOME MORE
000070 000012 R
                     33 END     EXIT
000071 006505 A
000072 000006 E
000073 000200 A
000074 120240 A      34 CNTRL   DATA     '  '           PRINT CONTROL
000075               35 BUFF    BSS      RECL
       000000 R      36         END      START


PAGE    3  08-16-76  SWITCH    VORTEX  DASMR              1326 HOURS

ENTRY NAMES
000000 R SWITCH
EXTERNAL NAMES
000055 E LOFCB  000016 E PIFCB  000072 E VSEXEC 000052 E VSIOC
000061 E VSIOST
SYMBOLS
000001 A ASCII  000002 A B      000075 R BUFF   000060 R BUSY
000074 R CNTRL  000024 A COUNT  000032 R DOIT   000071 R END
000005 A LO     000055 E LOFCB  000001 A NOWAIT 000004 A PI
000016 E PIFCB  000012 R READ   000021 R READCR 000050 A RECL
000000 R START  000000 R SWITCH 000072 E VSEXEC 000052 E VSIOC
000061 E VSIOST 000000 A WAIT   000051 R WRITE  000001 A X
  0 ERRORS ASSEMBLY COMPLETE
```

Figure 5-3. Example of Assembled and Executed DAS MR Program Under VORTEX Control *(continued)*

```
13127122  /EXEC

PAGE    1  08-16-76  SWITCH    VORTEX  LMGEN

  VSFMER A 71345   VSFMCB A 71335    VSBIC A 71244   VSERR A 70632
  VSFNRM A 70307   IOFOOD A 70213    VSFNR A 70013  VSTBSR A 67062
  VSALTB A 67002   VSSERV A 65646   VSFNIS A 65254   VSEMP A 65205
   VSSAL A 63323   VSEROR A 63071    IFLAG A 62770  VSPFDN A 62675
  VSPFUP A 62552    VSFPP A 62352   VSMPER A 62267  VSMPJP A 62166
  VSCLOK A 62160    BIFCB A 75516    SIFCB A 75460  VSGFCB A 75460
  VSJPBF A 75407   VSOPBF A 75336     VSTB A 75303  TIDSL2 A 75303
  TIDBSL A 75251   TIDBER A 75217   TBINTH A 75165  VSIOST A 71154
   VSIOC A 67262   VSEXEC A 65646    PIFCB A 75472   LOFCB A 75504
  SWITCH A 01000   [SIAP] A 00500   [SLIT] A 00777  [SPED] A 01145
                                                          HCTIWS    EMAN
                                                               *     UQE HCTIWS
                                                    BCFOL,BCFIP     TXE
                                                               1     UQE      X
                                                               2     UQE      B
                               TNUOC HCTIWS                   02     UQE  TNUOC
                  )SDROW NI( HTGNEL DROCER           TNUOC+TNUOC     UQE   LCER
                         TUPNI ROSSECORP                       4     UQE     IP
                          TUPTUO GNITSIL                       5     UQE     OL
                              OI ROF TIAW                      0     UQE   TIAW
                         NRUTER ETAIDEMMI                      1     UQE TIAWON
                                                               1     UQE  IICSA
                                                               *     UQE  TRATS
                                                  LCER,FFUB,IP   KNILOI
                                                1+LCER,LRTNC,OL  KNILOI
                                             IICSA,TIAW,IP,BCFIP  DAER   DAER
                                        RCDAER,DNE,DNE,DNE,DAER  TATS RCDAER
                                                           TNUOC  IXDL
                                                               *   UQE   TIOD
                             DROW A TEG            X,1-FFUB   EADL
```

Figure 5-3. Example of Assembled and Executed DAS MR Program Under VORTEX Control (*continued*)

```
                   SETYB HCTIWS                         8     ALRL
                 RETNIOP TREVNI                               XPC
              DROW ESREVNI TEG X,1+LCER+FFUB                  EBDL
DROW DEHCTIWS LANIGIRO EVAS X,1+LCER+FFUB                     EATS
DROW ESREVNI FO SETYB HCTIWS                            8     BLRL
                RETNIOP EROTSER                               XPC
   DROW ESREVNI DETREVNI EVAS           X,1-FFUB              EBTS
                     NWOD TNUOC                               RXD
                 EROM FI TAEPER             TIOD              ZNXJ
                            IICSA,TIAW,OL,BCFOL              ETIRW   ETIRW
                         YSUB,DNE,DNE,DNE,ETIRW               TATS    YSUB
                 EROM EMOS DAER             DAER               PMJ
                                                              TIXE     DNE
                  LORTNOC TNIRP             '  '              ATAD   LRTNC
                                            LCER               SSB    FFUB
                                           TRATS               DNE
```

Figure 5-3. Example of Assembled and Executed DAS MR Program Under VORTEX Control *(continued)*

5.2.2 DAS MR Operation (MOS)

The DAS MR assembler may be loaded and executed under the Master Operating System (MOS) using the following directives:

```
/ASSEMBLE
/A,p(1),p(2),...,p(n)
```

This control directive directs the executive to load the assembler. The parameter string specifies optional tasks for the assembler or executive to perform after the assembly is completed. These tasks are:

Parameter	Definition	Default Assignment
N	No source listing	Source listing
B	No binary object	Binary object program output
MAP	Memory map on load-and-go	No memory map on load-and-go
L	Load-and-go after assembly	No load-and-go after assembly
M	No symbol table listing	Symbol table listing

To read the same physical symbolic source statements for both assembly passes, input:

```
/ASSIGN   PO=DUM,SI=PI
/ASSEMBLE
```

The processor output listing serves as a copy of the program; it can be input for another assembly.

During a DAS MR assembly operation, if logical unit SS is not a magnetic tape unit, a flag bit is set in the peripheral control word PCW. When the end of pass 1 is detected, this bit is interrogated. If it is set, DAS MR does a status check on logical unit PO, prints the message RELOAD SOURCE on the Teletype, and halts. When the computer is placed in the run mode, DAS MR rewinds logical unit SS and begins pass 2 of the assembly. If the flag bit is not set (SS not equal to magnetic tape), no status check is done on PO and DAS MR immediately rewinds logical unit SS and begins pass 2.

Figure 5-4 illustrates a sample program assembly under MOS.

```
/JOB,EXAMPLE
/DATE,08-17-76
/ASSEMBLE,B,L
```

Figure 5-4. Example of Assembled and Executed DAS MR Program Under MOS Control

```
PAGE     1  EXAMPLE   08-17-76

                          1          NAME     STRT
          106612 A        2 CRLF     EQU      0106612
000000                    3 STRT     BSS      0
000000    002000 A        4          WALF     5,36,NAME
000001    000000 E
000002    001005 A
000003    000044 A
000004    000016 R
000005    002000 A        5          STAT     5,*+4,*+3,*+2,*-6
000006    000001 E
000007    000005 A
000010    000014 R
000011    000014 R
000012    000014 R
000013    000005 R
000014    002000 A        6          CALL     EXIT
000015    000000 E
000016    106612 A        7 NAME     DATA     CRLF,'ODEAN J. GASTON'
000017    147704 A
000020    142701 A
000021    147240 A
000022    145256 A
000023    120307 A
000024    140723 A
000025    152317 A
000026    147240 A
000027    106612 A        8          DATA     CRLF,'975 N. GRAND'
000030    134667 A
000031    132640 A
000032    147256 A
000033    120307 A
000034    151301 A
000035    147304 A
000036    106612 A        9          DATA     CRLF,'ORANGE    ORANGE'
```

Figure 5-4. Example of Assembled and Executed DAS MR Program Under MOS Control (continued)

```
000037 147722 A
000040 140716 A
000041 143705 A
000042 120240 A
000043 120240 A
000044 147722 A
000045 140716 A
000046 143705 A
000047 106612 A    10          DATA    CRLF,'CALIF     92667',CRLF,0


PAGE     2  EXAMPLE   08-17-76

000050 141701 A
000051 146311 A
000052 143240 A
000053 120240 A
000054 120240 A
000055 134662 A
000056 133266 A
000057 133640 A
000060 106612 A
000061 000000 A
000061             11 LAST    BES      0
                   12 EXIT    EXT
       000000 R    13         END      STRT
ENTRY NAMES
 000000 R STRT
EXTERNAL NAMES
 000015 E EXIT   000006 E IOCS
SYMBOLS
 106612 A CRLF   000015 E EXIT   000006 E IOCS   000061 R LAST
 000016 R NAME   000000 R STRT
   0 ERRORS ASSEMBLY COMPLETE
```

Figure 5-4. Example of Assembled and Executed DAS MR Program Under MOS Control *(continued)*

ODEAN J. GASTON
975 N. GRAND
ORANGE ORANGE
CALIF 92667

Figure 5-4. Example of Assembled and Executed DAS MR Program Under MOS Control *(continued)*

5.2.3 DAS MR Operation (Stand-Alone)

DAS MR may be loaded and executed under control of the stand-alone FORTRAN IV loader. The operating procedure is as follows:

a. Load the stand-alone loader using the binary load/dump program (BLD II). Set A register to zero before loading to prevent execution of the stand-alone loader. At completion of loading, the execution address of the stand-alone loader will be in the X register (013260).

b. Make the following modifications to memory:

Location	**New Contents**
5	0210
6	0210
7	0210

c. Execute the stand-alone loader by setting the P register to the execution address determined in step a and pressing RUN.

d. When executed, the stand-alone laoder will print "LN" on the Teletype. At this time, peripheral device assignments may be altered by entering the one-digit number of the old logical unit followed by the two-digit number of the substitute unit. DAS MR uses the following logical units:

Logical Unit Number	**Logical Unit Name**	**Default Device Assignment**
3	PI	Card reader
4	LO	Line printer
2	BO	Paper tape punch
6	GO	Dummy
8	SS	Magnetic tape* 00
9	PO	Magnetic tape** 10

* Device Address 010
** Device Address 011

As an example of device reassignment:

```
LN
300400201806900
```

Would reassign:

PI = Teletype Keyboard
LO = Teletype Printer
BO = Teletype Paper Tape Punch
SS = Teletype Keyboard
PO = Dummy

For a complete list of peripheral assignments, see table 5-4.

Table 5-4. List of Peripheral Assignments for Stand-Alone DAS MR

Logical Unit Number	Assignment
0	Teletype keyboard and printer
1	Teletype paper tape reader and punch
2	High-speed paper tape reader/punch
3	Card reader
4	Line printer
5	Dummy
6	Dummy
7	Card punch
8	Magnetic tape unit 0
9	Magnetic tape unit 1
10	Magnetic tape unit 2
11	Magnetic tape unit 3
12	Unformatted paper tape I/O (HSPT)

e. Following device reassignments, the stand-alone loader will print "IN" on the Teletype. At this time, the operator should ready the DAS MR object on the input device and respond by typing the proper designation on the Teletype:

P = Paper Tape Reader
T = Teletype Paper Tape Reader
0, 1, 2, 3 = Magnetic Tape Controller
0, 1, 2, or 3 respectively

To enable print out of a load map, the operator must type "M" immediately following the device designator. Following the typed characters, the operator must type a CR (carriage return) to initiate loading of the DAS MR object.

If an error is detected, the loader types a 2-character error message code and halts. To continue, the operator should remove the cause of the error (refer to error messages), ready the input device to read from the beginning of the object material, reload the loader program, and repeat the above procedure.

Error Messages

The following 2-character error messages are output to the Teletype whenever the corresponding error condition is detected:

Messages	Meaning
PS	Program Size Error. Program memory requirements exceed available program/common storage.
LS	Literal Size Error. Program literal requirements exceed available literal storage.
CM	Common Error. The program contains conflicting size definitions for a common block.
DA	Data Error. The program attempted to overlay the loader, loader tables, or resident programs.
TX	Text Error. The program object text contains an illegal or erroneous loader code.
RD	Read Error. The loader encountered a read error while attempting input of object text.
RC	Record Error. The loader inputs an invalid record type.
SQ	Sequence Error. The loader inputs an object text record with an invalid sequence number.
CK	Check-Sum Error. The loader inputs an object text record with an invalid check-sum.

f. After DAS MR is loaded, peripheral devices for logical units 3, 4, 2, 6, 8, and 9 must be loaded from the Run-Time I/O tape. This is accomplished by placing the Run-Time I/O tape on the input device and repeating step e.

g. After the Run-Time I/O is loaded, the I/O control program must be loaded from the Run-Time utility tape. This is accomplished by placing the Run-Time utility tape on the input device and repeating step e.

h. When all externals have been satisfied the loader will halt with the P register = 3. To execute DAS MR, the operator should press RUN.

Upon execution, DAS MR will input source statements from logical unit (PI), output source for pass to logical unit (PO), input pass source from logical unit (SS), output binary object to logical unit (BO), and output listing to logical unit (LO).

Source input to DAS MR terminates upon input of either an EOF or a source record containing a slash (/) as the first character. A slash record will cause an end-of-file to be output to the BO device.

5.2.4 DAS 8A Operation

The DAS 8A assembler may be loaded and executed by the stand-alone procedure described in the following paragraphs.

Loading the Assembler. Load the assembler program into memory using the binary load/dump program (BLD II). Execute it by entering a positive, nonzero value in the A register during loading, or by clearing all registers, pressing (SYSTEM) RESET and entering the RUN state. (Set RUN indicator on and press START).

During execution, the program first determines the amount of memory required. It then stores in address 000003 a value one less than the lower limit of BLD II. This is the highest address that the assembler can use without destroying part of BLD II.

DAS 8A comprises two sections: The I/O section allows the specification of I/O devices for assembler input and output. The second section is the assembler itself.

I/O Section Operation. The I/O section of DAS 8A, using the Teletype printer, makes three requests for definitions of I/O devices:

```
ENTER DEVICE NAME FOR xx
```

where xx is one of the I/O function names: SI (source input), LO (list output), or BO (binary output), respectively.

I/O Device Assignment. Assignment of I/O devices is accomplished by responding to each request in turn by means of a Teletype keyboard input which names the desired device, followed by a carriage return (CR). The acceptable device names for each request are listed in table 5-5. If the default assignment is desired, press CR only.

If an incorrect device name is type, the message:

```
DEVICE NAME NOT VALID
```

is output and the request repeated.

To terminate the output of any line to the Teletype, press RUBOUT. The error correction feature can be used any time during I/O device specification.

When I/O assignments are complete, the I/O section uses BLD II to load the assembler section into memory.

To restart the I/O section before the assembler section is loaded, set STEP indicator on, clear all registers, press (SYSTEM) RESET, set RUN indicator on and press START.

Table 5-5. Acceptable I/O Devices

Assembly Function	Device	Description	Default Assignment
SI (source input)	TR	Teletype paper tape read	TR
	TY	Teletype keyboard	
	PR	High-speed paper tape reader	
	CR	Card reader (026 code)	
	CR1	Card reader (029 code)	
	MTnn	Magnetic tape	
LO (list output)	TY	Teletype printer	TY
	LP2	Line printer (70-6701)	
BO (binary output)	TP	Teletype paper tape punch	TP
	PP	High-speed paper tape punch	
	CP	Card punch	
	MTnn	Magnetic tape	

Assembler Section Operation. When BLD II relinquishes control to the assembler section, the computer halts with 000001 in the program counter (P register). For an assembler pass 1, set SENSE switch 1; for pass 2, reset SENSE switch 1 and set SENSE switches 2 and 3.

If pass 1 is selected, ready the SI device with the source input media and set RUN indicator on and press START.

For pass 2, ready the SI device with the source input media, ready the BO and LO devices, set RUN indicator on and press START.

The END directive terminates both passes 1 and 2. Pass 1 terminates with 000001 in the P register and 0177777 in the A register. Pass 2 produces the binary object loader text and program listing and terminates when END is encountered with the same register values as pass 1. A MORE directive causes the computer to stop and wait until the SI unit prepared with the additional source input media, and the RUN state is entered. MORE is indicated by 0170017 in the A register.

The program listing can be suppressed during pass 2 by resetting SENSE switch 2, and the binary output, resetting SENSE switch 3. Error messages cannot be suppressed and are output on the LO device as the error is detected during pass 2.

Synchronization errors halt the assembly with 000777 in the A register. To continue the assembly, set RUN indicator and press START. The assembler resets the location counter value to that assigned on pass 1, prints error message *SE, and continues the assembly.

Pass 2 can be restarted or repeated for extra copies of the assembled program without repeating pass 1.

At the completion of pass 2, the assembler can accept another assembly using the same I/O devices. For other I/O devices, reload the assembler program, starting with the I/O section.

To restart the assembler, set STEP indicator on, clear all registers, press (SYSTEM) RESET, set RUN indicator on and press START. The assembler halts with 000001 in the P register and is ready to accept another assembly.

Using Magnetic Tape. The DAS 8A assembler can communicate with any of the magnetic tape transports on a controller. Up to four transports may be connected to each of the tape controllers. A configuration may have one to four magnetic tape controllers.

The magnetic tape transport number and controller device address is specified in the device name specification of the I/O Control Section. A listing of magnetic tape transport device names with their corresponding tape transport number and address is given in table 5-6.

Table 5-6. Device Names for Magnetic Tape Transports

Device Name		Transport Number
MT00	010	1
MT01	010	2
MT02	010	3
MT03	010	4
MT10	011	1
MT11	011	2
MT12	011	3
MT13	011	4
MT20	012	1
MT21	012	2
MT22	012	3
MT23	012	4
MT30	013	1
MT31	013	2
MT32	013	3
MT33	013	4

A coding example of a DAS 8A program is shown in figure 5-5. An example of an assembled DAS 8A program is shown in figure 5-6. An example of an assembled DAS 8A program with errors is shown in figure 5-7.

DAS CODING FORM

PROGRAMMER: * EXAMPLE   PROGRAM: SQUARE RØØT PRØGRAM   PAGE 1 OF 3

```
LABEL   OPERATION      VARIABLE AND COMMENT FIELD
*
* THIS IS A RØUTINE TØ CALL THE SQUARE RØØT (XSQT) SUBRØUTINE.
*    ERRØR RETURN FØR SQUARE RØØT ØF NEGATIVE NUMBERS IS IN CALL
*    +2 (n+2) NORMAL RETURN FRØM SQUARE RØØT IS AT CALL + 3 (n+3).
*    THIS RØUTINE IS DESIGNED TØ TAKE THE SQUARE RØØT
*    ØF 40 ØCTAL NUMBERS AND STØRE THE ANSWER IN 40 ØCTAL LØC.
*
        ØRG     0500            STARTING ADDRESS
        LDXI    037             XR = CØUNT - 1
NEXT    LDB     LØC,1           BR = (LØC + XR)
        CALL    XSQT,0777       SUBR CALL WITH ERRØR RETURN
        STB     SQRT,1          NØRMAL RETURN STØRE RESULT
*
* NØTE THAT THE DATA IS RETRIEVED AND STØRED FRØM
* BØTTØM TØ TØP
*
        JXZ     HALT            XR = 0 END ØF RØUTINE
        DXR                     INDEX - 1 = INDEX
        JMP     NEXT            RETURN FØR NEXT NUMBER
HALT    HLT                     NØRMAL HALT
LØC     DATA    25,30,36,050,-1,100,0100,0,4,200
        DATA    1000,0700,-40,50,60,70,80,90,110,120
        DATA    0,02000,2,9,3000,03000,15,17,130,0140
        DATA    0204,300,310,320,330,340,350,400,500,-10
```

Figure 5-5. Coding Example

DAS CODING FORM

2 3

```
SQRT   BSS   040            RESERVE 40 ØCTAL LØCATIØNS
*
*  INTEGER SQUARE RØØT SUBRØUTINE CALCULATED BY THE APPRØXIMATION
*
*  1/2 (Xi + A/Xi) = Xi+1 (DO NOT KEYPUNCH)
*
*  ENTER WITH NUMBER FØR SQUARE RØØT IN THE B REGISTER. THE
*  X REGISTER IS SAVED AND REPLACED ØN EXIT. ERRØR RETURN FØR
*  SQUARE RØØT OF NEGATIVE NUMBERS AT n+2 FRØM CALL.
*  NØRMAL RETURN AT n+3 FROM CALL WITH SQUARE RØØT ØF NUMBER
*  IN THE B REGISTER
*
XSQT   ENTR                 PLACE WHERE RETURN ADDR IS SAVED
       JBZ   EXIT+1         SQ RT. ØF 0=0
       TBA                  NUMBER = BR = AR
       JAN*  XSQT           ERRØR RETURN TØ N+2
       STB   NMBR           SAVE NUMBER
       STB   APRX           NUMBER = 1ST APPRØXIMATIØN
       STX   SAVE           SAVE XR
       LDXI  7              INITIALIZE XR FØR APPR.
AGN    TZA                  ZERØ AR FØR DIVIDE
       LDB   NMBR           NUMBER = BR
       DIV   APRX           NUMBER / APPRØXIMATIØN
       TBA                  A/X = BR = AR
```

Figure 5-5. Coding Example (continued)

DAS CODING FORM

PROGRAMMER | PROGRAM | 3 3

LABEL	OPERATION	VARIABLE AND COMMENT FIELD		IDENTIFICATION
*				
	ADD	APRX	A/X+X =AR	
	TAB		A/X+X =AR •BR	
	ASRB	1	(A/X+X)1/2 =BR	
	STB	APRX	NEXT APPRØXIMATIØN	
	DXR		XR- 1 =XR	
	JXZ	EXIT	SQ RT. =BR	
	JMP	AGN	CØMPLETE APPRØXIMATION	
EXIT	LDX	SAVE	RESTØRE XR	
	INR	XSQT	UPDATE ENTRY TO n+3	
	RETU*	XSQT	GØ BACK TØ MAIN PRØGRAM	
NMBR	BSS	1		
APRX	BSS	1		
SAVE	BSS	1		
	END		NØ EXECUTIØN ADDRESS	

Figure 5-5. Coding Example (continued)

```
PAGE     000001

                        *EXAMPLE                            SQUARE ROOT PROGRAM
                        *
                        *  THIS A ROUTINE TO CALL THE SQUARE ROOT (XSQT) SUBROUTINE,
                        *   ERROR RETURN FOR SQUARE ROOT OF NEGATIVE NUMBERS IS IN CALL
                        *   +2 (N+2) NORMAL RETURN FROM SQUARE ROOT IS AT CALL + 3 (N+3)
                        *   THIS ROUTINE IS DESIGNED TO TAKE THE SQUARE ROOT
                        *   OF 40 OCTAL NUMBERS AND STORE THE ANSWER IN 40 OCTAL LOC,
                        *
000500                        ,ORG     ,0500            STARTING ADDRESS
000500  006030                ,LDXI    ,037             XR = COUNT - 1
000501  000037
000502  025515          NEXT  ,LDB     ,LOC,1           BR = (LOC + XR)
000503  002000                ,CALL    ,XSQT,0777       SUBR CALL WITH ERROR RETURN
000504  000626 R
000505  000777
000506  065566                ,STB     ,SQRT,1          NORMAL RETURN STORE RESULT
                        *
                        *  NOTE THAT THE DATA IS RETRIEVED AND STORED FROM
                        * BOTTOM TO TOP
                        *
000507  001040                ,JXZ     ,HALT            XR = 0 END OF ROUTINE
000510  000514 R
000511  005344                ,DXR     ,                INDEX - 1 = INDEX
000512  001000                ,JMP     ,NEXT            RETURN FOR NEXT NUMBER
000513  000502 R
000514  000000          HALT  ,HLT     ,                NORMAL HALT
000515  000031          LOC   ,DATA    ,25,30,36,050,-1,100,01,00,0,4,200
000516  000036
000517  000044
000520  000050
000521  177777
000522  000144
000523  000001
000524  000000
000525  000000
000526  000004
000527  000310
000530  001750                ,DATA    ,1000,0700,-40,50,60,70,80,90,110,120
000531  000700
000532  177730
000533  000062
000534  000074
```

VTII-1171

Figure 5-6. Example of an Assembled DAS 8A Program

```
PAGE     000002

  000535  000106
  000536  000120
  000537  000132
  000540  000156
  000541  000170
  000542  000000          ,DATA    ,0,02000,2,9,3000,03000,15,17,130,01 40
  000543  002000
  000544  000002
  000545  000011
  000546  005670
  000547  003000
  000550  000017
  000551  000021
  000552  000202
  000553  000001
  000554  000204          ,DATA    ,0204,300,310,320,330,340,350,400,500,-10
  000555  000454
  000556  000466
  000557  000500
  000560  000512
  000561  000524
  000562  000536
  000563  000620
  000564  000764
  000565  177766
  000566             SQRT  ,BSS     ,040           RESERVE 40 OCTAL LOCATIONS
                     *
                     *  INTEGER SQUARE ROOT SUBROUTINE CALCULATED BY THE APPROXIMATION
                     *
                     *        1/2 (X + A/Xi) = Xi + 1
                     *
                     *  ENTER WITH NUMBER FOR SQUARE ROOT IN THE B REGISTER. THE
                     *  X REGISTER IS SAVED AND REPLACED ON EXIT.  ERROR RETURN FOR
                     * SQUARE ROOT OF NEGATIVE NUMBERS AT N+2 FROM CALL.
                     *  NORMAL RETURN AT N+3 FROM CALL WITH SQUARE ROOT OF NUMBER
                     *  IN THE B REGISTER
                     *
  000626  000000     XSQT  ,ENTR    ,              PLACE WHERE RETURN ADDR IS SAVED
  000627  001020           ,JBZ     ,EXIT+1        SQ RT. OF 0=0
  000630  000657 R
  000631  005021           ,TBA     ,              NUMBER = BR = AR
  000632  001004           ,JAN*    ,XSQT          ERROR RETURN TO N+2
```

VTII-1172

Figure 5-6. Example of an Assembled DAS 8A Program (continued)

```
PAGE      000003

  000633   100626 R
  000634   060662          ,STB     ,NMBR          SAVE NUMBER
  000635   060663          ,STB     ,APRX          NUMBER = 1ST APPROXIMATION
  000636   070664          ,STX     ,SAVE          SAVE XR
  000637   006030          ,LDXI    ,7             INITIALIZE XR FOR APPR.
  000640   000007
  000641   005001    AGN   ,TZA     ,              ZERO AR FOR DIVIDE
  000642   020662          ,LDB     ,NMBR          NUMBER = BR
  000643   170663          ,DIV     ,APRX          NUMBER / APPROXIMATION
  000644   005021          ,TBA     ,              A/X =BR =AR
  000645   120663          ,ADD     ,APRX          A/X+X =AR
  000646   005012          ,TAB     ,              A/X+X =AR =BR
  000647   004101          ,ASRB    ,1             (A/X+X)1/2 =BR
  000650   060663          ,STB     ,APRX          NEXT APPROXIMATION
  000651   005344          ,DXR     ,              XR- 1 =XR
  000652   001040          ,JXZ     ,EXIT          SQ RT. =BR
  000653   000656 R
  000654   001000          ,JMP     ,AGN           COMPLETE APPROXIMATION
  000655   000641 R
  000656   030664    EXIT  ,LDX     ,SAVE          RESTORE XR
  000657   040626          ,INR     ,XSQT          UPDATE ENTRY TO N+2
  000660   001000          ,RETU*   ,XSQT          GO BACK TO MAIN PROGRAM
  000661   100626 R
  000662             NMBR  ,BSS     ,1
  000663             APRX  ,BSS     ,1
  000664             SAVE  ,BSS     ,1
           000000          ,END     ,              NO EXECUTION ADDRESS

LITERALS

POINTERS

SYMBOLS

1  000664 R  SAVE
1  000663 R  APRX
1  000662 R  NMBR
1  000656 R  EXIT
1  000641 R  AGN
1  000626 R  XSQT
1  000566 R  SQRT
```

VTII-1173

```
PAGE     000004

1  000515 R  LOC
1  000514 R  HALT
1  000502 R  NEXT
```

VTII-1174

Figure 5-6. Example of an Assembled DAS 8A Program (continued)

```
PAGE     000001

                            *EXAMPLE L                      EXAMPLE WITH ERRORS
  015000                         ,ORG     ,015000
  015000  005011                 ,TZA     ,010              CANNOT HAVE A VAR. FIELD
*SZ
  015001  005001        SEC      ,TZA     ,
*DD
  015002  001411                 ,HLT     ,777              VARIABLE FIELD TO LARGE
*SZ
  015003  000777                 ,HLT     ,0777
  015004  015036                 ,LDA     ,ALFA,1           EXP 1 TO LARGE
*AD
  015005  006015                 ,LDAE    ,ALFA,1
  015006  015036 R
  015007  006030                 ,LDXI    ,ALFA
  015010  015036 R
  015011  015000        SEC      ,LDA     ,0,1              DOUBLE DEFINITION
*DD
  015012  000004                 ,LDA     ,0,4              EXP 2 HAS TO BE A 1 OR 2
*TF
  015013  015000                 ,LDA     ,0,1
  015014  016000                 ,LDA     ,0,2
  015015  014020                 ,LDA     ,ALFA             CREATE A REL ADDRESS
  015016  006010                 ,LDAI    ,77777            VAR FIELD TO LARGE
*SZ
  015017  027721
  015020  006010                 ,LDAI    ,077777
  015021  077777
  015022  006010                 ,LDAI    ,32767
  015023  077777
  015024  006010                 ,LDAI    ,-32768
  015025  100000
                                 ,JZZ     ,ALFA             ILLEGAL OPERATION CODE
*OP
  015030  001040                 ,JXZ     ,ALFA
  015031  015036 R
  015032  001000                 ,JMP     ,BRA              BRA UNDEFINED
*SY
  015033  000000
  015034  001000                 ,JMP     ,BRAV
  015035  015037 R
  015036  000005        ALFA     ,DATA    ,5
  015037  014045        BRAV     ,DATA    ,014045
```

VTII-1177

```
PAGE     000002

  015040                STR      ,BSS     ,1
          000000                 ,END     ,

LITERALS

POINTERS

SYMBOLS

0   015040 R   STP
1   015037 R   BRAV
1   015036 R   ALFA
0   015001 R   SEC
```

VTII-1178

Figure 5-7. Example of an Assembled DAS 8A Program with Errors

SECTION 6

STAND-ALONE FORTRAN/DAS MR LIBRARIES

There are eight libraries for the stand-alone FORTRAN/DAS MR system.

6.1 COMPLEX MATH FUNCTIONS (FORTRAN CODED)

This library consists of programs collected, without modification, from the MOS. In order, they are:

$9E	$AC
CCOS	CMPLX
CSIN	$8K
CLOG	$8L
CEXP	$8M
CSQRT	$8N
CABS	$ZD
CONJG	AIMAG
$AK	$OC
$AL	REAL
$AM	$8F
$AN	$8S

6.2 DOUBLE PRECISION MATH FUNCTIONS (FORTRAN CODED)

This library consists of programs collected, without modification, from the MOS. In order, they are:

$XE	DMINI
$YE	DSIGN
$ZE	$YK
DATAN2	$YL
DLOGIO	$YM
DMOD	$YN
DINT	DBLE
DABS	$XC
DMAXI	

6.3 SINGLE PRECISION MATH FUNCTIONS (FORTRAN CODED)

This library consists of programs collected, without modification, from the MOS. In order, they are:

TANH	SNGL
ATAN2	MAX0
ALOG10	MAX1

AMOD	MIN0
AINT	MIN1
AMAX0	MOD
AMAX1	INT
AMIN0	IDIM
AMIN	IFIX
DIM	$JC
FLOAT	

6.4 DOUBLE PRECISION ARITHMETIC (DAS CODED)

This library consists of programs collected from the MOS. The only modifications made were the deleting or adding of control cards to define the object code for 16- or 18-bit machines. In order, they are:

DSINCOS	DMULT
DATAN	DDIVIDE
DEXP	DADDSUB
DLOG	DNORMAL
IF	DLOADAC
POLY	DSTOREAC
CHEB	RLOADAC
DSQRT	SINGLE
$DFR	DOUBLE
IDINT	DBLECOMP

6.5 SINGLE PRECISION ARITHMETIC (DAS CODED)

6.5.1 Hardware Multiply/Divide

This library consists of programs collected from the MOS. The only modifications made were the deleting or adding of control cards to define the object code for 16- or 18-bit machines. In order, they are:

$HE	XDADD
$PE	XDSUB
$QE	XECOMP
ALOG	$FLOAT
EXP	$IFIX
ATAN	IABS
SQRT-H	ABS
SINCOS	ISIGN
FMULDIV	SIGN
FADDSUB	$HN-H
SEPMANTI	$HM-H
FNORMAL	XMUL
XDDIV-H	XDIV
XDMULT-H	I$FA

6.5.2 SOFTWARE MULTIPLY/DIVIDE

This library consists of programs collected from the MOS. The only modifications made were the deleting or adding of control cards to define the object code for 16- or 18-bit machines. In order, they are:

$HE	XDADD
$PE	XDSUB
$QE	XDCOMP
ALOG	$FLOAT
EXP	$IFIX
ATAN	IABS
SQRT-S	ABS
SINCO	ISIGN
FMULDIV	SIGN
FADDSUB	$HN-S
SEPMANTI	$HM-S
FNORMAL	$XMUL
XDDIV-S	XDIV
XDMULT-S	I$FA

6.6 RUN-TIME I/O (DAS CODED)

This library consists of programs collected from the MOS. Control cards were added or deleted to define the object code for 16- or 18-bit machines.

Two additional modifications were made to the MOS routines: the Teletype paper tape reader and punch drivers were merged into a single driver, $OH/$01; and the entry name of the driver for the line printer was changed to $OR. In order, they are:

FORTIO	MT$3
$00	MTAE
$04	KNT$
$08	RDC$
$0C	WRT$
$0G	STR$
$0H/$01	SWR$
$00	BL$P
$0M	FCH$
CRIE	TCK$
$0Q($OR)	$TC01
$0Q	$HC37
$0P	HCK$
$0S	DIM$
CPAE	LAS$
MT$0	IOA$
MT$1	100K
MT$2	$BICD

6.7 RUN-TIME UTILITIES (DAS CODED)

This library, except for $BUF consists of MOS programs, some modified and some not. In the following list, an asterisk (*) flags the programs which have more extensive modifications than selecting the 16- or 18-bit word size. In order, they are:

$DO	$EE
$CG	RSCB3*
$3S	RSCBIMTB*
$SE	$BUF
FORTUTIL	

APPENDIX A
INDEX OF INSTRUCTIONS

Mnemonic	Octal Code	Description
AD	0072xx	Add
ADD	12xxxx	Add memory to A register
ADDE	00612x	Add extended
ADI	00745x	Add immediate
ADDI	006120	Add immediate
ADR	0075xx	Add register
ANA	15xxxx	AND memory and A register
ANAE	00615x	AND extended
ANAI	006150	AND immediate
AOFA	005511	Add overflow to A register
AOFB	005522	Add overflow to B register
AOFX	005544	Add overflow to X register
ASLA	004200 + n	Arithmetic shift left A register
ASLB	004000 + n	Arithmetic shift left B register
ASRA	004300 + n	Arithmetic shift right A register
ASRB	004100 + n	Arithmetic shift right B register
BT	0064xx	Bit test
CIA	1025xx	Clear and input to A register
CIAB	1027xx	Clear and input to A and B registers
CIB	1026xx	Clear and input to B register
COM	00743x	Complement register
COMP	005xxx	Complement source to destination registers

INDEX OF INSTRUCTIONS

Mnemonic	Octal Code	Description
CPA	005211	Complement A register
CPB	005222	Complement B register
CPX	005244	Complement X register
DADD	004x2x	Double add
DAN	004x4x	Double AND
DAR	005311	Decrement A register
DBR	005322	Decrement B register
DEC	00742x	Decrement register
DECR	0053xx	Decrement source to destination registers
DER	004x6x	Double Exclusive OR
DVI	17xxxx	Divide
DIVE	00617x	Divide extended
DIVI	006170	Divide immediate
DLD	004x0x	Double load
DOR	004x5x	Double OR
DST	004x1x	Double store
DSBU	004x3x	Double subtract
DXR	005344	Decrement X register
ERA	13xxxx	Exclusive OR memory and A register
ERAE	00613x	Exclusive OR extended
ERAI	006130	Exclusive OR immediate
EXC	100xxx	External control
EXC2	104xxx	Auxiliary external control
FAD	105410	Add single precision memory to floating point accumulator

Mnemonic	Octal Code	Description
FADD	105503	Add double precision memory to floating point accumulator
FDV	105401	Single precision floating point divide
FDVD	105535	Double precision floating point divide
FIX	105621	Reformat floating point accumulator and store integer in memory
FLD	105420	Load floating point accumulator with single precision number
FLDD	105522	Load floating point accumulator with double precision number
FLT	105425	Reformat single precision integer and load into floating point accumulator
FMU	105416	Single precision floating point multiply
FMUD	105506	Double precision floating point multiply
FSB	105450	Single precision floating point subtraction
FSBD	105543	Double precision floating point subtraction
FST	105600	Store floating point accumulator in memory in single precision format
FSTD	105710	Store floating point accumulator in memory in double precision format
HLT	000000	Halt
IAR	005111	Increment A register
IBR	005122	Increment B register
IJMP	0067xx	Indexed jump

INDEX OF INSTRUCTIONS

Mnemonic	Octal Code	Description
IME	1020xx	Input to memory
INA	1021xx	Input to A register
INAB	1023xx	Input to A and B registers
INB	1022xx	Input to B register
INC	00741x	Increment register
INCR	0051xx	Increment source to destination registers
INR	04xxxx	Increment memory and replace
INRE	00604x	Increment memory and replace extended
INRI	006040	Increment memory and replace immediate
IXR	005144	Increment X register
JAN	001004	Jump if A register negative
JANM	002004	Jump and mark if A register negative
JANZ	001016	Jump if A register not zero
JANZM	002016	Jump and mark if A register not zero
JAP	001002	Jump if A register positive
JAPM	002002	Jump and mark if A register positive
JAZ	001010	Jump if A register zero
JAZM	002010	Jump and mark if A register zero
JBNZ	001026	Jump if B register not zero
JBNZM	002026	Jump and mark if B register not zero
JBZ	001020	Jump if B register zero
JBZM	002020	Jump and mark if B register zero
JDNZ	00677x	Jump if double precision register not zero

Mnemonic	Octal Code	Description
JDZ	00676x	Jump if double precision register zero
JIF	001xxx	Jump if conditions met
JIFM	002xxx	Jump and mark if conditions met
JMP	001000	Jump unconditionally
JMPM	002000	Jump and mark unconditionally
JN	00674x	Jump if register negative
JNZ	00673x	Jump if register not zero
JOF	001001	Jump if overflow indicator set
JOFN	001007	Jump if overflow indicator not set
JOFM	002001	Jump and mark if overflow indicator set
JOFNM	002007	Jump and mark if overflow indicator not set
JP	00675x	Jump if register positive
JSR	0065xx	Jump unconditionally and set return in X register
JS1M	002100	Jump and mark if SENSE switch 1 set
JS2M	002200	Jump and mark if SENSE switch 2 set
JS3M	002400	Jump and mark if SENSE switch 3 set
JS1N	001106	Jump if SENSE switch 1 not set
JS2N	001206	Jump if SENSE switch 2 not set
JS3N	001406	Jump if SENSE switch 3 not set
JS1NM	002106	Jump and mark if SENSE switch 1 not set

INDEX OF INSTRUCTIONS

Mnemonic	Octal Code	Description
JS2NM	002206	Jump and mark if SENSE switch 2 not set
JS3NM	002406	Jump and mark if SENSE switch 3 not set
JSS1	001100	Jump if SENSE switch 1 set
JSS2	001200	Jump if SENSE switch 2 set
JSS3	001400	Jump if SENSE switch 3 set
JXNZ	001046	Jump if X register not zero
JXNZM	002046	Jump and mark if X register not zero
JXZ	001040	Jump if X register zero
JXZM	002040	Jump and mark if X register zero
JZ	00672x	Jump if register zero
LASL	004400 + n	Long arithmetic shift left
LASR	004500 + n	Long arithmetic shift right
LBT	00746x	Load byte
LD	0070xx	Load
LDA	01xxxx	Load A register
LDAE	00601x	Load A register extended
LDAI	006010	Load A register immediate
LDB	02xxxx	Load B register
LDBE	00602x	Load B register extended
LDBI	006020	Load B register immediate
LDI	00744x	Load immediate
LDX	03xxxx	Load X register
LDXE	00603x	Load X register extended

Mnemonic	Octal Code	Description
LDXI	006030	Load X register immediate
LLRL	004440 + n	Long logical rotation left
LLSR	004540 + n	Long logical rotation right
LRLA	004240 + n	Logical rotation left A register
LRLB	004040 + n	Logical rotation left B register
LSRA	004340 + n	Logical shift right A register
LSRB	004140 + n	Logical shift right B register
MERG	0050xx	Merge source to destination registers
MUL	16xxxx	Multiply
MULE	00616x	Multiply extended
MULI	006160	Multiply immediate
NOP	005000	No operation
OAB	1033xx	Output OR of A and B registers
OAR	1031xx	Output from A register
OBR	1032xx	Output from B register
OME	1030xx	Output from memory
ORA	11xxxx	OR memory and A register
ORAE	00611x	OR extended
ORAI	006110	OR immediate
ROF	007400	Reset overflow indicator
SB	0073xx	Subtract
SBR	0076xx	Subtract register
SBT	00747x	Store byte
SEN	101xxx	Program sense

INDEX OF INSTRUCTIONS

Mnemonic	Octal Code	Description
SOF	007401	Set overflow indicator
SOFA	005711	Subtract overflow from A register
SOFB	005722	Subtract overflow from B register
SOFX	005744	Subtract overflow from X register
SRE	0066xx	Skip if register equal
ST	0071xx	Store
STA	05xxxx	Store A register
STAE	00605x	Store A register extended
STAI	006050	Store A register immediate
STB	06xxxx	Store B register
STBE	00606x	Store B register extended
STBI	006060	Store B register immediate
STX	07xxxx	Store X register
STXE	00607x	Store X register extended
STXI	006070	Store X register immediate
SUB	14xxxx	Subtract memory from A register
SUBE	00614x	Subtract extended
SUBI	006140	Subtract immediate
T	0077xx	Transfer
TAB	005012	Transfer A register to B register
TAX	005014	Transfer A register to X register
TBA	005021	Transfer B register to A register
TBX	005024	Transfer B register to X register
TSA	007402	Transfer switches to A register
TXA	005041	Transfer X register to A register

Mnemonic	Octal Code	Description
TXB	005042	Transfer X register to B register
TZA	005001	Transfer zero to A register
TZB	005002	Transfer zero to B register
TZX	005004	Transfer zero to X register
XAN	003004	Execute if A register negative
XANZ	003016	Execute if A register not zero
XAP	003002	Execute if A register positive
XAZ	003010	Execute if A register zero
XBNZ	003026	Execute if B register not zero
XBZ	003020	Execute if B register zero
XEC	003000	Execute unconditionally
XIF	003xxx	Execute if conditions met
XOF	003001	Execute if overflow indicator set
XOFN	003007	Execute if overflow indicator not set
XS1	003100	Execute if SENSE switch 1 set
XS2	003200	Execute if SENSE switch 2 set
XS3	003400	Execute if SENSE switch 3 set
XS1N	003106	Execute if SENSE switch 1 not set,
XS2N	003206	Execute if SENSE switch 2 not set
XS3N	003406	Execute if SENSE switch 3 not set
XXNZ	003046	Execute if X register not zero
XXZ	003040	Execute if X register zero
ZERO	00500X	Zero (clear) registers

NOTE: n = shift count

APPENDIX B
V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
200	128	NUL			Null
201	129	SOH			Start of Heading
202	130	STX			Start of Text
203	131	ETX			End of Text
204	132	EOT			End of Transmission
205	133	ENQ			Enquiry
206	134	ACK			Acknowledge
207	135	BEL			Bell
210	136	BS			Backspace
211	137	HT			Horizontal Tab
212	138	LF			Line Feed
213	139	VT			Vertical Tab
214	140	FF			Form Feed
215	141	CR			Carriage Return
216	142	SO			Shift Out
217	143	SI			Shift In
220	144	DLE			Data Link Escape
221	145	DC1			Device Control 1
222	146	DC2			Device Control 2
223	147	DC3			Device Control 3
224	148	DC4			Device Control 4
225	149	NAK			Negative Acknowledge
226	150	SYN			Synchronous File

V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
227	151	ETB			End of Transmission Block
230	152	CAN			Cancel
231	153	EM			End of Medium
232	154	SUB			Substitute
233	155	ESC			Escape
234	156	FS			File Separator
235	157	GS			Group Separator
236	158	RS			Record Separator
237	159	US			Unit Separator
240	160	SP	(blank)	(blank)	Space
241	161	!	11/2/8	11/2/8	Exclamation Point
242	162	"	7/8	0/5/8	Quotation Mark
243	163	#	3/8	0/7/8	Pound Sign
244	164	$	11/3/8	11/3/8	Dollar Sign
245	165	%	0/4/8	11/7/8	Percent Sign
246	166	&	12	12/7/8	Ampersand
247	167	'	5/8	4/8	Apostrophe (prime)
250	168	(	12/5/8	0/4/8	Left Paren
251	169	)	11/5/8	12/4/8	Right Paren
252	170	*	11/4/8	11/4/8	Asterisk
253	171	+	12/6/8	12	Plus Sign
254	172	,	0/3/8	0/3/8	Comma
255	173	–	11	11	Minus Sign
256	174	.	12/3/8	12/3/8	Period
257	175	/	0/1	0/1	Slash

Octal	Decimal	Character	029	026	Description
260	176	0	0	0	
261	177	1	1	1	
262	178	2	2	2	
263	179	3	3	3	
264	180	4	4	4	
265	181	5	5	5	
266	182	6	6	6	
267	183	7	7	7	
270	184	8	8	8	
271	185	9	9	9	
272	186	:	2/8	5/8	Colon
273	187	;	11/6/8	11/66/8	Semi-Colon
274	188	<	12/4/8	12/6/8	Less Than
275	189	=	6/8	3/8	Equal Sign
276	190	>	0/6/8	6/8	Greater Than
277	191	?	0/7/8	12/2/8	Question Mark
300	192	@	4/8	0/2/8	At
301	193	A	12/1	12/1	
302	194	B	12/2	12/2	
303	195	C	12/3	12/3	
304	196	D	12/4	12/4	
305	197	E	12/5	12/5	
306	198	F	12/6	12/6	
307	199	G	12/7	12/7	
310	200	H	12/8	12/8	
311	201	I	12/9	12/9	

V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
312	202	J	11/1	11/1	
313	203	K	11/2	11/2	
314	204	L	11/3	11/3	
315	205	M	11/4	11/4	
316	206	N	11/5	11/5	
317	207	O	11/6	11/6	
320	208	P	11/7	11/7	
321	209	Q	11/8	11/8	
322	210	R	11/9	11/9	
323	211	S	0/2	0/2	
324	212	T	0/3	0/3	
325	213	U	0/4	0/4	
326	214	V	0/5	0/5	
327	215	W	0/6	0/6	
330	216	X	0/7	0/7	
331	217	Y	0/8	0/8	
332	218	Z	0/9	0/9	
333	219	[	12/2/8	12/5/8	Left Bracket
334	220	\	11/7/8	0/6/8	Backslash
335	221	]	0/2/8	11/5/8	Right Bracket
336	222	↑ or ∧	12/7/8	7/8	Vertical Arrow
337	223	← or –	0/5/8	2/8	Horizontal Arrow
340	224				Accent Grave
341	225	a			
342	226	b			