Honeywell

DAP-16 AND DAP-16 MOD 2 ASSEMBLY LANGUAGE

SERIES 16

SUBJECT:

DAP-16 Assembly Language and Its Extension for the 316 and 516 Computers, DAP-16 Mod 2.

SPECIAL INSTRUCTIONS:

This manual completely supersedes the edition dated August 1970. Changes specified by ECO 9246 update this manual to comply with Revision C of the Assembler and provide improved examples to assist programmers in more efficient application of the DAP-16 Assembly Language. The order number has been changed to be consistent with the overall Honeywell publications numbering system.

DATE:

June 1971

ORDER NUMBER:

BY09, Rev. 0 (Formerly M-1756)

DOCUMENT NUMBER:

70130072442B

PREFACE

This document is organized as a reference manual. The DAP-16 and DAP-16 Mod 2 Assembly Languages and Assemblers used on Series 16 general purpose computer systems are described. Subject areas include pseudo-operations (instructions to the assembler rather than instructions to be assembled into the program), the mixing of FORTRAN and DAP-16 programs in a memory load, performing an assembly, and generating an assembler system.

Users of this manual should have some familiarity with Series 16 computers but need no assembly language experience. The 316/516 Programmers' Reference Manual (Order No. BX47, Doc. No. 70130072156 - M-490) and the 316/516 Operators' Guide (Order No. BX48, Doc. No. 70130072165 - M-491) are companion volumes.

Series 16 DAP-16 and DAP-16 Mod 2 Assembly Language is a coded program designed to extend the power of Series 16 in the area of program preparation and maintenance. It is supported by comprehensive documentation and training; periodic program maintenance is furnished for the current version of the program in accordance with established Honeywell specifications, provided it is not modified by the user.

File No.: 1A23

BY09

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

ILLUSTRATIONS

ILLUSTRATIONS (Cont)

TABLES

SECTION I
INTRODUCTION

SCOPE OF MANUAL

This manual describes the DAP-16 and DAP-16 Mod 2 Assembly Languages and Assemblers for use on Honeywell Series 16 general purpose computer systems. DAP-16 Mod 2 is an extension of the DAP-16 Assembly Language which is supported only on the DDP-516 and H316 computers. All existing source programs for these computer systems will assemble correctly using the DAP-16 Mod 2 Assembler.

SUPPORTING PROGRAMS

Source programs written in DAP-16 language may be processed by several supporting programs. Each provides the programmer with a specific tool helping him toward the goal of producing an efficient, error-free object program.

The DAP-16 Assembler is the primary program for processing the DAP-16 Language. This program produces object text for eventual loading into the computer along with a listing of the source program and the assembler's action on each statement. This program is discussed in Section II of this Manual.

The Macro Preprocessing Program permits processing of a DAP-16 source program with several additional statement types. These statements allow predefined blocks of source text to be modified and inserted in a copy of the source program. The term "Macro" implies that one statement produces several instruction blocks. These blocks, called macro-expansions, may be defined within the program or may come from a macro library. These macro-expansions are also modified to include appropriate symbols for each instance of use. Through use of the Macro Preprocessing Program the programmer can significantly reduce the number of statements to be written. With this program, the user can also define a new language which suits his needs more closely than DAP-16. Macros also aid "installation standard" code for system interfacing where the macro library contains the critical code for connecting user programs with the operating system and/or I/O equipment. The output of the Macro Preprocessing Program is a DAP-16 source text suitable for use by any of the programs discussed in this Manual. However, the Macro Preprocessing Program is discussed in a separate Manual, titled MAC Macro Preprocessor Programmers Reference Manual.

The Concordance Program operates upon a DAP-16 source program in a manner similar to the operation of the DAP-16 Assembler. Its Output is a cross-reference table listing each symbolic name and literal and the source locations of every reference to them. This program is discussed in a separate Manual, titled XREF Concordance Program Programmers Reference Manual.

The Update Program allows manipulation of a source program within the computer. This program is discussed in a separate manual titled 016-XREF, SSUP and MAC Source Language Processors.

The discussion so far has concerned the assembly process prior to loading. However, a loading program is logically inseparable from an associated assembler, because the path from assembly language code to loaded program must pass through both the assembler and the loader. The loading programs used with either DAP-16 Assembler are described in Section II of this manual.

Figure 1-1 illustrates the processing of a DAP-16 source program by these supporting programs. Note that Figure 1-1 references another useful program, namely, the Write and Load Program. This type of program provides a core dump which is easily reloaded without the use of a loader, providing a handy method of storing completed programs between use.

REFERENCE DOCUMENTS

Document	Doc. No.	Order No.
DAP-16	70180275000	M-1052
DAP-16M2	70181446000	M-1727
DECCL	70180455000	M-236
DECCS	70180458000	M-186
DUMY-X16	70180095000	M-861
IOS-OAAA	70182615000	M-1732
IOS-ORAA	70182603000	M-1726
IOS-ORPA	70182601000	M-1723
IOS-O16D	70181507000	M-1810
IOS-5AAA	70180323000	M-1053
IOS-5CAA	70180618000	M-535
IOS-5CPA	70180594000	M-534
IOS-5RAA	70180592000	M-538
IOS-5RPA	70180573000	M-354
IOS-516D	70180278000	M-567
IOS-516X	70180324000	M-1054
LDR-APM	70180005000	M-569
LDR-C	70180582000	M-860
MEMSIZ	70180606000	M-363
MINILOAD	70180580000	M-372
O16-DECL	70181506000	M-1801
O16-DECS	70181505000	M-1703
SETSIZ	70180457000	

Document	Doc. No.	Order No.
SLDR-A	70180341000	M-237
SLDR-C	70180583000	M-368
SLDR-P	70180342000	M-76
SYMLIST	70181445000	M-1821
TABLESIZ	70181497000	M-1728

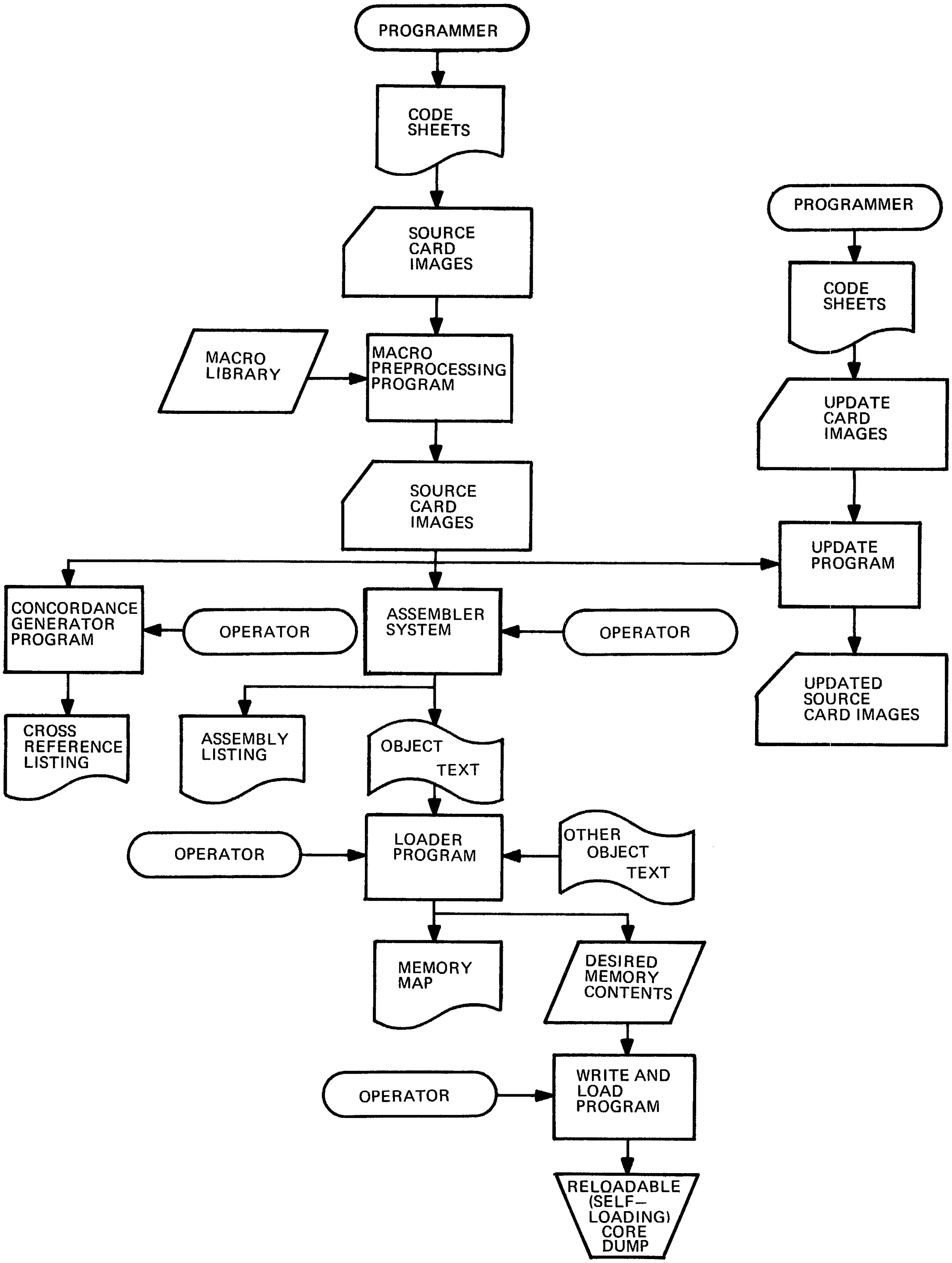


Figure 1-1. General Program Flow

SECTION II
DAP-16 ASSEMBLER

The DAP-16 Assembler provides the programmer with the means for generating linkages between a source program and others which are assembled or compiled separately. The linkage is actually performed by the Loader. Each point in a program to be linked is assigned an external symbolic name which is then referenced by any other program requesting use of that link point. The Loader will not complete its job until all references to external names in the program being loaded have been satisfied.

The Assembler produces two independent outputs. The first is the object text which is further processed by the Loader, and the second is the assembly listing. The listing serves to inform the programmer of the actions taken by the Assembler so he can eliminate errors and make other changes. The assembly listing also carries programmer comments and other documentation.

The assembly listing is printed during the final pass. Thus the listing from a two-pass assembly contains more information than that from a one-pass assembly, namely the definition of all symbols encountered anywhere in the program. Object tapes from the two types of assembly may be loaded by the same Loader.

The DAP-16 Assembler must be linked to a number of support programs which permit it to operate independent of associated input/output devices and to operate either alone or under an operating system. The input/output system can use a general supervisor, allowing successive assemblies to be conducted with different devices, or can be formed from one of several dedicated supervisors which use a preselected combination of input/output devices. Such a dedicated supervisor is useful for systems where standard devices are always used or the available memory is limited. Note that the DAP-16 Assembler is referred to as an Assembler System in Figure 1-1. The specific programs comprising this system are described in Section V and VI.

The Assembler may make either one or two passes over a source text depending on how the assembly is initiated.

LOCATION COUNTER

The DAP-16 Assembler maintains a Location Counter which points to the memory location for which a word is currently being assembled. This counter is relocatable or absolute depending on the mode of assembly and is used in defining symbols appearing in the Location Field and in establishing a value for asterisks appearing in the Address Field.

After each word (instruction) is assembled, the Location Counter is normally incremented by one.

TWO-PASS ASSEMBLY

In this mode of assembly, the DAP-16 Assembler reads the source program twice, first to develop a dictionary of symbols, and a second time to assemble the object program by referencing the Symbol Table (Dictionary). Each entry in the Symbol Table is three words in length. Therefore, the maximum number of symbols that may be handled is one-third of the number of locations available (usually all of the locations between the highest location used by the assembler and the highest location of memory). During pass two, DAP-16 assembles and outputs the Object Text and Assembly Listing. Each source line is processed before the next line is read. Figure 2-1 illustrates the processing of each line.

During the processing of a line, the operation mnemonic is first examined. If a standard machine operation is being conducted, the proper code is inserted in the object text. If a pseudo operation is specified (calling for some action by the assembler rather than specifying an operation code) the proper action is taken. The address field is then processed and the proper value inserted in the object text. The assembly listing image is formed and any errors detected in the line are flagged at the left end.

ONE-PASS ASSEMBLY

The development of the Symbol Table and the assembly of the Object Program are accomplished simultaneously in a one-pass assembly. Any symbols not defined when encountered are assigned an internal symbol number. The printed output shows two asterisks in the field which would contain the symbol value. When the Assembler determines the assigned value of a symbol this information is included in the object text. The Loader then uses this information to finish assembling the instruction words in core.

LOADERS

A Loader processes object text to form a core image and places this image in memory. Memory references within the program are resolved and indirect links generated as required. References to external names (which are assembled without an address) are also resolved. The Loader operates in the mode specified by the programmer in the source text. Loaders, which are large and complicated programs, are as important to the process of generating an executable core content as Assemblers and Compilers.

There are two kinds of Loaders available, namely linking and non-linking. LDR-APM, SLDR-A, SLDR-C, and SLDR-P are the linking Loaders; and MINILOAD is the non-linking Loader.

LDR-APM is the full Loader, and with proper support programs can load object text from any medium or mix of media. Object Text from either one or two-pass assemblies can be loaded as well as FORTRAN Object Texts with all external references correctly linked.

SLDR-A and SLDR-P are smaller linking loaders for paper tape Object Texts loaded through an ASR teletypewriter and the high-speed tape reader respectively. SLDR-C is the small linking loader for punch card object text. These Loaders can load object text from

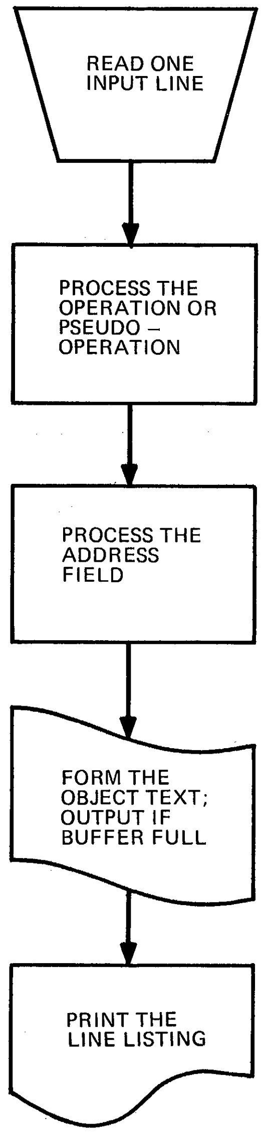


Figure 2-1. DAP-16 Processing of a Line

two-pass assemblies and FORTRAN compilations, but not from one-pass assemblies. Again, all external references are correctly linked.

MINILOAD is the smallest of the Loaders, and loads object text from any medium in conjunction with appropriate support programs. The object text must be derived from two-pass assemblies. One-pass assemblies and FORTRAN compilations cannot be loaded. Furthermore, only one mode of loading must be used in any one program. Since no linkages are made to external names, these must be handled by the programmer as absolute references.

MODES OF OPERATION

There are three assembly and loading modes which may be specified to and through the DAP-16 Assembler by the programmer. These are illustrated in Figure 2-2. The descriptions of the pseudo-operations which implement the three operating modes are located in Section III.

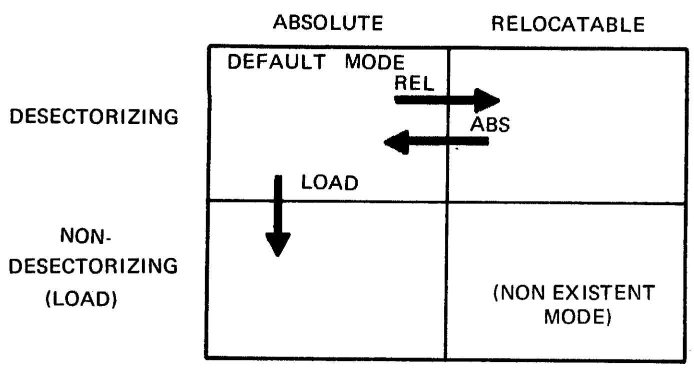


Figure 2-2. Assembler and Loader Operating Modes

Desectorizing Modes

In the two Desectorizing Modes, the Loader handles all intersector references by generating indirect address links (vectors) when necessary. These links are located in sector zero unless the programmer has specified location elsewhere by the use of a SETB pseudo-operation. Because in general the programmer may not be aware of which instructions will have indirect bits set by the Loader, he must be careful in modifying the address of instructions during program execution.

The Loader may handle intersector links for either normal addressing or extended addressing. The EXD pseudo-operation causes the Loader to form 15-bit indirect address links, while the LXD pseudo-operation returns the Loader to the normal 14-bit mode. These pseudo-operations should be used in conjunction with the EXA and DXA machine operations. The effect of EXD and LXD may also be forced by the operator at load time.

Desectorizing and Absolute Mode. -- This mode is the Assembler default mode for program loading unless one of the other modes is specified. The location at which the program is loaded is fixed by the ORG pseudo-operation, which must be assembled before any locations are assigned. This location cannot be changed at load time.

Desectorizing and Relocatable Mode. -- This mode differs from the Desectorizing and Absolute Mode in that addresses may be relocated at load time. The REL pseudo-operation initiates entrance into this mode. The ABS pseudo-operation may be used to return to Desectorizing and Absolute mode.

Any symbolic names assigned in the relocatable portion of a program are considered relocatable. Such symbols may not be treated in ways which the Loader cannot handle, (e.g., being added together).

Load Mode

In this mode all intersector links are assumed to be handled by the object program. Warning flags are posted whenever a link is required. The Loader will generate the link if this program is loaded. This feature provides a useful tool for debugging, timing, or loading a program when the programmer must give cross sector linkages special treatment. Addresses are absolute (there is no relocatable load mode). The Load Mode is entered with the LOAD pseudo-operation and continues for the duration of the assembly.

CODING DAP-16 PROGRAMS

Symbolic Names

DAP-16 uses Symbolic Names to identify numerical values computed by the Assembler. These values are normally the addresses of instructions or data. The assembler maintains a Symbol Table that permits substitution of the proper value for any reference to a Symbolic Name.

The most common method of assigning values to Symbolic Names is to enter the symbol to be named in the location field of the DAP-16 coding form. The assembler will assign the value of the Location Counter to that symbol when that line is processed. Multiple definition is an error. Symbols may also be assigned values by the EQU and SET pseudo-operations.

Allowable symbols consist of from one to four characters from the 37-character set A-Z, 0-9, and $, with at least one of the characters in a symbol being alphabetic. The dollar sign can not be the first character, and generally should be used with care since it usually signifies system programs. Six-character symbols may be used for referenced external names in the address field.

The following symbols are legitimate:

LOOP
ST2P
A$
CENTER (an external name)

DAP-16 Coding Form

The DAP-16 Assembler's input support programs accept input in either of two formats, namely, fixed-field and tab-field (paper tape input only). In the fixed-field format each source line is an 80-character field (a punched card image). Each data field within this 80-character field has a specified location. The input drivers convert a tab-field format to this fixed-field format. Each data field may be terminated by a backslash character (\, '334), and the source line may be terminated by a carriage return.

Figure 2-3 shows a DAP-16 Assembler Coding Form. The five fields that appear on this form are: Location, Operation, Operand, Comments, and Identification. The circled t's in columns 5, 11, and 29 signify that a backslash to the left of that column will be interpreted as a tab to the column following the marked column. Similarly, the circled CR in column 72 indicates that the comments field may be terminated by a carriage return. Furthermore, Table 2-1 shows in detail how the assembler defines and interprets these fields in both the fixed-field and tab-field formats. Notice that each field, with the exception of the Comments and Identification fields, is terminated by blanks. Therefore, their contents must be left-justified and cannot contain embedded blanks. If, for example, the statement X1 LDA X2+7 were written as X1 LDA X2 + 7, the assembler would interpret this statement as X1 LDA X2 and assume that + 7 was a comment:

```
0103                        * INTENDED STATEMENT
0104 01030    0 02 01427        LDA    X2+7
0105                        * CAUSES INCORRECT ACTION IF WRITTEN AS
0106 01031    0 02 01420        LDA    X2 + 7
```

Text Examples

The examples in this manual are shown in the form of assembly listings which are described in detail at the end of this section. The first few examples present both the coding form and the assembly listing to show the correspondence of the fields. See DAP-16 ASSEMBLY LISTINGS near the end of this section for a description of the fields generated to the left of what the programmer has written.

Location Field

Each time a symbolic name is encountered in the location field it is entered into the symbol table along with the value of the location counter at the time the name was encountered. Thus, the location field is used to name instructions or data for later reference. In the second pass of the assembler (or the first pass for one-pass assemblies), the symbolic name is replaced by its value as found in the symbol table. In addition, the location field can sometimes be used in other ways by pseudo-operations. References to multiply defined symbols are arbitrarily assigned to the first definition.

As asterisk in column 1 of the location field signifies that the entire line is a comment, which is printed on the output listing but otherwise ignored. The first line in the

Honeywell

DAP CODING FORM

F574-1070

PROGRAMMER	DATE	PAGE OF
PROGRAM		CHARGE

LOCATION (1–4)	(5)	OPERATION (6–10)	(11)	OPERAND FIELD (12–29)	COMMENTS (30–44)	EXTENDED COMMENTS (45–72)	IDENTIFICATION (73–80)
* IN	T	ENDED		STATEMENT			
		LDA		X2+7			
* CA	U	SES I	N	CORRECT ACTION IF	WRITTEN AS		
		LDA		X2 + 7			

Figure 2-3. DAP-16 Coding Form

assembly (whether it is a comment or not) is used as a header for all pages in the assembly listing.

Operation Field

This field contains the abbreviation (mnemonic) of an operation or pseudo-operation. If a given abbreviation is not recognized or is not legal on the object machine, an error is flagged.

TABLE 2-1. DAP-16 ASSEMBLER FORMATS

Field	Fixed-Field Format	Tab-Field Format	How Assembler Handles Field
Location	Column 1 to first blank column following	First column to first blank or backslash following	Symbolic name for address of this operation or data
Operation	Column 6 to first blank column following	End of location field to next blank or back-slash	Abbreviation for operation or pseudo-operation
Operand	Column 12 to first blank column follow-ing	End of operation field to next blank or back-slash	Variables or data
Comments	First blank column following column 12 to column 44	First 15 characters between end of operand field and carriage return character	Printed on listing, otherwise ignored
Extended Comments	Columns 45 to 72	Any remaining characters before carriage return character	Printed on listing, except overprints last character on ASR
Identification	Columns 73 to 80	Part of comments field	Printed on listing, otherwise ignored

An asterisk (*) used in the operation field of a memory reference line (immediately following the operation code) signifies that the indirect bit is to be set. For example, to store the contents of the A-register indirectly through the location at symbolic name XNA (i.e., to store at the location pointed to by XNA), the following code would be written:

LOCATION	①	OPERATION	①	OPERAND FIELD	①	COMMENTS
		STA*		XNA		INDIRECT STORE

The assembly listing of this line would appear as follows:

```
0010 03407  -0 04 03244      STA*   XMA                   INDIRECT STORE
```

The assembly listing always shows a minus sign for indirect references as shown above.

Address Field

The address field is used in several ways, but generic operations do not use this field at all. As an example of the use of this field, consider two's complementing the current value of the A-register:

LOCATION (1–4)	5	OPERATION (6–10)	11	OPERAND FIELD (12–29)	COMMENTS (30–)
		TCA			2S COMPLEMENT

or:

```
0011 06654   140407          TCA                          2S COMPLEMENT
```

Shift instructions use the address field to specify the number of bit positions to shift. For instance, a 4-bit logical right shift is coded as:

LOCATION (1–4)	5	OPERATION (6–10)	11	OPERAND FIELD (12–29)	COMMENTS (30–)
		LGR		4	THIS IS A SHIFT

```
0018 02763    0404 74          LGR    4                THIS IS A SHIFT
```

In input/output instruction lines, both the value of the function and the controller address may be coded as a single variable or an expression to be evaluated. This is often an octal number coded with an apostrophe. However, it is recommended that a symbol be used (see SET and EQU Pseudo-Operations in Section III). For instance, before using the ASR (controller address = '04), it must be enabled in the proper mode. The function code for enabling in the output mode (applicable to ASR) is '01. Therefore, the instruction for enabling the ASR in the output mode may be coded numerically as:

LOCATION	OPERATION	OPERAND FIELD	COMMENTS
	OCP	'0104	ASR OUTPUT
*	OR		
	OCP	'104	
*	OR		
	OCP	'100+'4	

The first line is listed as:

```
0014 07667    14 0104          OCP    '0104            ASR OUTPUT
```

The address field for memory reference instructions contains two subfields. The first subfield specifies the address to be used in the instruction. For example, loading the A-register with the contents of the memory cell at symbolic location CEX would be coded as follows:

LOCATION	Ⓣ	OPERATION	Ⓣ	OPERAND FIELD Ⓣ	COMMENTS
1 2 3 4	5	6 7 8 9 10	11	12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
		LDA		CEX	

and assembled as:

```
0020 04202   0 02 04221      LDA   CEX
```

In the example above, the second subfield is null. However, when used, the second subfield usually specifies that the index bit is to be set in the assembled line. A value of 0 or null designates no indexing; a value of 1 designates indexing. All other values are errors. Also, the two subfields are separated by commas. For example, storing the contents of the A-register in the memory cell at the address which is the sum of the symbolic value CEX and the contents of the X-register at the time this STA instruction is executed would generate the following line on the listing:

```
0022 04207   1 04 04221      STA   CEX,1
```

The assembly listing shows the index bit as 0 or 1 for memory reference lines. Pseudo-operations can use the address field in a number of ways, some of which allow division into many subfields separated by commas.

Expressions. -- The address field generally contains a symbolic algebraic expression to be evaluated, with the result of the evaluation being passed to the loader through the object text. Within the object text, such an expression may be either absolute or relocatable.

Only plus and minus operators are allowed. Furthermore, all elements of the expression must be constants or symbols present in the symbol table by the end of the final pass. Arithmetic may not be performed on external symbols. No indication of overflow is given. The following examples show both addition and subtraction. In the third line, indexing is also specified.

```
0101 06072   0 02 06252      LDA   DATA+5
0102 06073   0 06 06244      ADD   DATA-1
0103 06074   1 04 06154      STA   RSLT+40,1
0104 06075  -0 01 06373      JMP*  NEXT+20
0105 06076   0 000170        DAC   DATA-RSLT+23
```

Absolute and Relocatable Symbols. -- Symbols defined within relocatable program segments are relocatable. Other symbols and all constants are absolute. In the following example the retrieval of the contents of core location 0002 is implemented irrespective of where the instruction resides or is relocated in core.

```
0018 03717    0 02 00002        LDA    2
```

Special Elements. -- The asterisk is used as an element by itself, and three other symbols — the apostrophe ('), the dollar sign ($), and the equals sign (=) — modify the elements they precede.

The DEC and DBP pseudo-operations allow the letters B and E to be used in the address field to specify the position of binary and decimal points (these pseudo-operations are discussed in Section III).

Asterisk. -- The single asterisk is a variable which always has the value of the location counter. For example:

```
0026 04615    0 01 04614        JMP    *-1
```

means jump to the previous instruction. The two following examples have the same effect, a jump to symbolic location CONT:

```
0030 00462    0 01 00464        JMP    *+2
0031 00463    101000            NOP
0032 00464    0 01 00501        JMP    CONT
```

```
0077 00462    0 01 00464        JMP    X3
0078 00463    101000            NOP
0079 00464    0 01 00501  X3    JMP    CONT
```

Double Asterisk. -- The double asterisk is assembled as zero. Normally the program will set the address during execution.

```
0030 01347    0 01 00000        JMP    **
```

The example above might be used in a program in which the location to be jumped to was unknown before assembly. The loader places zero in the 9-bit address and 1-bit sector fields and handles the index and indirect bits normally. However, if this instruction were assembled in sector 0 rather than sector 1, the sector bit would be one, because the referenced location, location 0, is in the same sector as the instruction.

Apostrophe (Octal Numbers). -- An Apostrophe preceding a number signifies that the number is to be evaluated as an octal number. The following examples yield the same result:

```
0037 05164    0 02 00200      LDA    '200

0022 05164    0 02 00200      LDA    128
```

The minus sign for negative numbers should follow the apostrophe, e.g., '-60 = -48, and the minus operator in expressions should precede the Apostrophe; A-'60 is valid but A+'-60 is not.

Dollar Sign (Hexadecimal Numbers).[a] -- A Dollar Sign preceding a number signifies that the number is to be evaluated as a hexadecimal number. The following examples yield the same result:

```
0034 00213    0 02 00017      LDA    $F

0041 00213    0 02 00017      LDA    15

0026 00213    0 02 00017      LDA    '17
```

The minus sign for negative numbers should follow the Dollar Sign, e.g., $-30 = -48. The minus operator in expressions should precede the Dollar Sign; A-$30 is valid but A+$-30 is not.

Equal Sign (Literals). -- The use of constants in calculations is done conventionally by storing a constant as data and writing the data name in the Address field. When reading the listing, the value of the constant is not apparent from its name. However, by using a literal (expressed as the value of the constant preceded by an equal sign), the same result is achieved except that the name of the constant now gives its actual value. There are two additional advantages to use of literals. First, the storage location of the literal becomes the concern of the Assembler and Loader rather than the object program (i.e., a literal is self-defining). And second, all references to a literal of the same value refer to the same location, even though the programmer may not remember that he had made more than one use of that value or even that the form of the literal is different.

Evaluated literals are stored in the Symbol Table along with other symbols.

---

[a]DAP-16 Mod 2 only.

The following examples all achieve the same effect, namely loading of a word composed of all ones (-1 in twos complement notation) into the A Register. The programmer controls the location of the -1 word in the first case, but the Assembler controls location in all other cases. In any case, the address in the assembled instruction is the address of a word containing -1.

```
0039 01306    0 02 01323          LDA    M1
                                   .
                                   .
                                   .
0043 01323    177777        M1    DEC    -1

0047 01306    0 02 01344          LDA    =-1

0051 01306    0 02 01344          LDA    ='-1

0055 01306    0 02 01344          LDA    =$-1

0059 01306    0 02 01344          LDA    ='177777
```

The DEC pseudo-operation, as used above, assembles a word with the indicated decimal value (-1 in this case).

USASCII Literals. -- To specify a USASCII literal the form =A is used. The following example implements loading of a 16-bit word containing C and $ ('141644) into the A-Register:

```
0045 00456    0 02 00563          LDA    =AC$
```

DAP-16 ASSEMBLY LISTINGS

The printed output of DAP-16 Assembler System is an Assembly Listing containing the source program as it was read along with the action taken by the assembler. Figure 2-4 illustrates a sample listing.

The first column contains the line record number of the source statement. The next column contains the value of the Assembler Location Counter (octal). The third column shows, in octal, the binary word assigned to the location. The parts of the word are broken up differently for different categories of instruction. Fifteen bits of address information are included in memory reference instructions and the Loader uses these fifteen bits to determine the ten bits of address information to be loaded into the instruction. The three modes of loading cause the Loader to modify these fifteen bits in three different ways.

Note the following features of Figure 2-4.

a. Line 1 contains an asterisk in the location field, causing DAP-16 to treat the entire line as remarks.

b. Line 2 contains a pseudo-operation (ORG) which sets the DAP-16 location counter to octal 1000, the starting address of sector one.

c. The expression in the variable field in line 3 means the current value of the location counter, plus one. Consequently, DAP-16 has written octal 1001 into the address field of the instruction word assigned to this location.

d. The symbol in the left margin of line 5 is a diagnostic signifying that a memory reference instruction (LDA) has an empty address field. Diagnostics are covered in more detail in Section III.

e. Indirect addressing is specified in line 5, and indexing is specified in line 8.

f. In line 10 the programmer has entered the number of shifts desired in an LGL instruction. DAP-16 has generated the necessary TWOs complement form in the object program.

g. The literal pool starts in line 11 and continues until all three literals called for have been satisfied.

```
     0001                          * SAMPLE ASSEMPLY LISTING
     0002                                ORG   512
     0003 01000   0 02 01001 STRT  LDA   *+1
     0004 01001   0 04 01000       STA   *-1
A    0005 01002  -0 02 00000       LDA*
     0006 01003   0 06 01012       ADD   =15
     0007 01004   0 06 01011       ADD   ='15
     0008 01005   1 04 00700       STA   STRT-64,1
     0009 01006   0 02 01010       LDA   ='-5
     0010 01007   0414 76          LGL   2
     0011 01010   177773           END
          01011   000015
          01012   000017
```

(Performs no useful function. See text for discussion of handling of fields.)

Figure 2-4. Assembly Listing

SECTION III
PSEUDO-OPERATIONS

DAP-16 pseudo-operations are instructions to the Assembler rather than instructions to be assembled into the program. Table 3-1 lists the abbreviations (mnemonics) for these instructions in the order of discussion. The most basic pseudo-operations are preceded by a checkmark.

TABLE 3-1. PSEUDO-OPERATIONS

Abbreviation	Meaning
	ASSEMBLY-CONTROLLING PSEUDO-OPERATIONS
CFx	Computer Configuration
√REL	Relocatable mode
ABS	Absolute mode
LOAD	Load mode
√ORG	Origin
FIN	Assemble Literals
MOR	Operator Action Required
√END	End of Source Program
	LIST-CONTROLLING PSEUDO-OPERATIONS
EJCT	Start at top of page
LIST	Generate assembly listing
NLST	Generate no assembly listing
	LOADER-CONTROLLING PSEUDO-OPERATIONS
EXD	Enter extended desectorizing
LXD	Leave extended desectorizing
SETB	Set base sector
	SYMBOL-DEFINING PSEUDO-OPERATIONS
√EQU	Give a symbol a permanent value
SET	Give a symbol a temporary value
	DATA-DEFINING PSEUDO-OPERATIONS
√DAC	Address constant
√DEC	Decimal constant
DBP	Double precision constant
√OCT	Octal constant
HEX	Hexadecimal constant
√BCI	Binary (ASCII) coded information
VFD	Variable field constant
	STORAGE ALLOCATION PSEUDO-OPERATIONS
BSS	Block starting with symbol
BES	Block ending with symbol
√BSZ	Block storage of zeros
COMN	Common storage
SETC	Set common base
	PROGRAM-LINKING PSEUDO-OPERATIONS
ENT	Entry point
√SUBR	Entry point
EXT	External name
√XAC	External address constant
√CALL	Call subroutine
	CONDITIONAL ASSEMBLY PSEUDO-OPERATIONS
IFP	Assemble only if plus
IFM	Assemble only if minus
IFZ	Assemble only if zero
IFN	Assemble only if not zero
ENDC	End of conditional assembly
ELSE	Combined IF and ENDC
FAIL	Identifies statement which should never be assembled
	SPECIAL SYMBOLS
***	Op Code Zero
PZE	Op Code Zero

In the discussion that follows, the diagram under the title of each pseudo-operation illustrates what the Assembler expects to find in the location, operation and operand fields. The comments and identification fields are used normally for all pseudo-operations. The words "previously defined" mean "already in the symbol table even in the first pass." The pseudo-operations that apply only to DAP-16 Mod 2 are footnoted.

ASSEMBLY-CONTROLLING PSEUDO-OPERATIONS

CFx, Computer Configuration

LOCATION	OPERATION	OPERAND
Ignored	CF1 for DDP-116 CF3 for H316 CF4 for DDP-416 CF5 for DDP-516	Ignored

The pseudo-operation CFx defines the computer on which the program is to run and if used, must precede the executable instructions to be tested. If the configuration is not specified with CFx, the DAP-16 Mod 2 Assembler assumes that the program will be run on an H316 or DDP-516. The DAP-16 Assembler assumes that the source computer is the object type. The only effect of this pseudo-operation is to print O flags on the listing for illegal operations. The object text is unaffected.

✓REL, Relocatable Mode

LOCATION	OPERATION	OPERAND
Ignored	REL	Ignored

The pseudo-operation REL specifies the desectorizing and relocatable mode for assembly and loading (see Section II, Modes of Operation). The action of the REL is reversibly terminated by an ABS pseudo-operation and irreversibly terminated by a LOAD pseudo-operation. REL may not follow LOAD.

ABS, Absolute Mode

LOCATION	OPERATION	OPERAND
Ignored	ABS	Ignored

This pseudo-operation specifies the desectorizing and absolute mode for assembly and loading (see Section II, Modes of Operation). The assembler assumes this as the operating mode in the absence of a REL, ABS, or LOAD pseudo-operation. The action of the ABS is reversibly terminated by a REL pseudo-operation and irreversibly terminated by a LOAD pseudo-operation. ABS may not follow LOAD.

LOAD, Load Mode

LOCATION	OPERATION	OPERAND
Ignored	LOAD	Ignored

The pseudo-operation LOAD informs the assembler that the source program from this point on is to be assembled in load mode (see Section II, Modes of Operation). All references to addresses not present in either the current sector or sector zero are flagged as errors on the assembly listing but do not affect the object text. Load mode continues in effect for the duration of the assembly.

✓ORG, Origin

LOCATION	OPERATION	OPERAND
Normal	ORG	Any previously-defined symbol or expression

The assembler's location counter is given the value of the expression in the address field. In the desectorizing and relocatable mode, the program will be loaded at the location specified by the ORG plus the relocation factor, which is not normally useful. In the absolute mode (either desectorizing or load) the ORG specifies the exact location at which the program will be loaded. Any number of ORGs may be used in a program.

Any symbol in the location field will be assigned the value of the location counter before the ORG is processed.

In the following example, a relocatable program temporarily reverts to absolute and stores two pointers to relocatable locations. The program then returns to the relocatable mode giving the location counter the value it would have had if the excursion into absolute had not been made.

```
0034                            REL            RELOCATABLE PROGRAM
                                 .
                                 .
                                 .
0037 01050   0 02 00334         LDA    '334    ...REFERENCING SECTOR
0038 01051   0 04 01573         STA    X47        ZERO
0039                       701  ORG    '334    START AT LOCATION
0040                            ABS               ABSOLUTE '334,
0041 00334   0 004465           DAC    X       PUT IN POINTERS,
0042 00335   0 004502           DAC    Y          AND
0043                            ORG    Z01     RETURN TO MAIN SEQUENCE
0044                            REL               (RELOCATABLE)
0045 01052  -0 06 00335         ADD*   '335
0046 01053   0 04 01574         STA    X48
```

In the example below, the next instruction must be in an odd location. The DBP pseudo-operation (described below) forces the assembler to locate its first word in an even memory location. Therefore, ODD in the example below is forced to be in an odd location.

```
0003 03260    0 01 03263        JMP    ODD
0004 03262    000000            DBP    0                  DUMMY VALUE; USED FOR
     03263    000000
0005                        *                               ALIGNMENT
0006                            ORG    *-1                FORCE ODD LOCATION
0007 03263    0 02 03244 ODD    LDA    XNA                PROGRAM EXECUTION RESUMES
```

FIN, Assemble Literals

LOCATION	OPERATION	OPERAND
Ignored	FIN	Ignored

Whenever the pseudo-operation FIN is encountered, DAP-16 starts at the present setting of the location counter and assembles all literals accumulated since the beginning of the program or since the last FIN. When the next statement is processed, the location counter points to the first location following the literals. The same function is performed by the END pseudo-operation; however, END also terminates the assembly. FIN allows the programmer to distribute literals throughout his program, thereby possibly reducing the indirect address links that the loader must supply. The program must not be allowed to jump to a location within the literal pool.

MOR, Operator Action Required

LOCATION	OPERATION	OPERAND
Ignored	MOR	Ignored

This pseudo-operation is used when additional material must be added to the assembly. When MOR is encountered the computer halts (unless the source input is on magnetic tape, in which case MOR is ignored). The computer resumes processing when the START button is pushed. MOR causes a halt on both the first and second passes.

✔END, End Of Source Program

LOCATION	OPERATION	OPERAND
Ignored	END	Blank of any defined symbol or expression. If blank, loader will start execution of program at its first location. Otherwise, execution will start at address specified.

An END pseudo-operation must be the last statement in a source program; no statements are processed following an END statement. All accumulated literals are assembled as with a FIN statement. If this is the final pass, the value in the address field is entered into the object text. The loader can be directed to start execution of the program at that address. If the address field is blank, the first address in the program will be entered into the object text as the starting address.

In a two-pass assembly from cards or paper tape, the computer halts when the END statement is reached on the first pass. The operator must then reposition the source text to its start and push the START pushbutton to initiate pass two. The second pass may be repeated with the same parameters or with other parameters to gain additional outputs.

LIST-CONTROLLING PSEUDO-OPERATIONS

EJCT, Start At Top Of Page

LOCATION	OPERATION	OPERAND
Ignored	EJCT	Ignored

The pseudo-operation EJCT causes the next source line on the assembly listing to be printed at the top of the next page following the heading. It has no effect if the NLST pseudo-operation is in effect. The EJCT pseudo-operation is effective only when the line printer is being used for the assembly listing or the ASR is being used with Input/Output Supervisor O16-OAAA (see Section V, Input/Output Supervisors). The line containing EJCT is printed.

LIST, Generate Assembly Listing; NLST, Generate No Assembly Listing

LOCATION	OPERATION	OPERAND
Ignored	LIST or NLST	Ignored

The LIST pseudo-operation causes the assembly listing to be printed. The assembler is ordinarily in the LIST mode. NLST inhibits printing of the assembly listing. LIST and NLST may be used throughout a program in order to list selected sections. The line containing NLST is printed if printing is on.

LOADER-CONTROLLING PSEUDO-OPERATIONS

EXD, Enter Extended Desectorizing; LXD, Leave Extended Desectorizing

LOCATION	OPERATION	OPERAND
Ignored	EXD or LXD	Ignored

The loader forms 14-bit indirect address words (each having an indirect bit and an index bit) unless an EXD pseudo-operation is performed or the operator forces extended loading at load time. EXD causes the loader to form 15-bit indirect address words (each having an indirect bit but no index bit). EXD, normally used in conjunction with the EXA operation, implies that the program is to be operated in EXTEND addressing mode. LXD, used in conjunction with the DXA operation, implies that the program is in the normal addressing mode.

SETB, Set Base Sector

LOCATION	OPERATION	OPERAND
Normal	SETB	Normal. For one-pass assemblies, any symbol used in this field must be previously defined.

The pseudo-operation SETB is used for programmer control of the location of the address constants. SETB causes the loader to place the address constants starting at the address derived from the address field of SETB. This statement may be used to ensure that the loader-generated address vectors are in the same sector as the instructions that use them. In this case, the program must reserve a block of memory locations for their storage. The following example shows this use of SETB.

```
0067                                ORG    '3000          START AT BEGINNING
0068                                SETB   *+1              OF SECTOR 3
0069  03000     0 01 03013          JMP    *+11           JUMP OVER ADDRESS
0070                          *                             CONSTANTS
0071  03001                         BSS    10             UP TO 10 CONSTANTS
0072  03013     0 02 03763          LDA    RTOP           CONTINUE HERE
```

SETB pseudo-operations and loader B-register settings may be used freely to move the base during the course of loading a program and its subroutines. The loader allows only one contiguous block of base locations to be in any one sector. Thus, if the base is ever returned to a sector it has been directed to before (e.g., back to sector zero) address constants will continue to be loaded immediately following the previous block of address constants loaded in that sector. For example, if the next address constant were to be loaded into location '134 when the loader encountered a SETB to another sector, a following

SETB to any location in sector zero (e.g., SETB 0, SETB '134, or SETB '100) would return the base to '134.

SETB may also be used with the base-setting operation SMK '1320 (Memory Lockout Option). The programmer must be sure that the relocation register is properly loaded when the program starts executing and that storage is allocated for the address constants.

SYMBOL-DEFINING PSEUDO-OPERATIONS

√EQU, Give a Symbol a Permanent Value

LOCATION	OPERATION	OPERAND
Normal. Must contain a symbol.	EQU	Normal. Any symbol used in this field must be previously defined.

The EQU pseudo-operation allows a symbol to be defined without being used in a location field, thereby permitting more than one symbol to refer to the same value. EQU also allows a symbol to be given a value outside the range of locations in the program. Once a symbol has been defined with EQU it may not be redefined.

SET, Give a Symbol a Temporary Value[a]

LOCATION	OPERATION	OPERAND
Normal. Must contain a symbol.	SET	Normal. Any symbol used in this field must be previously defined.

The SET pseudo-operation is identical to the EQU pseudo-operation, except that the symbol may be redefined any number of times with further SET pseudo-operations. An example of the use of EQU and SET pseudo-operations is shown below. At the start, EQU is used to set STRT = A, S1 = B, and S2 = C. SET is used to set TOP = A = STRT. Later, TOP is reset to '4223.

```
0053          001121      STRT EQU   *
0054          001122      S1   EQU   *+1
0055          001123      S2   EQU   *+2
0056          001121      TOP  SET   *
0057                      *                     START INSTRUCTIONS
0058 01121    0 000000    A    DAC   **
0059 01122    0 02 01162  B    LDA   CNT
0060 01123    141206      C    AOA
0061 01124    0 04 01162       STA   CNT
0062 01125   -0 01 01121       JMP*  TOP
0063                      *                     RETURN THROUGH
                                .                 TOP (=A)
                                .
                                .
                                .
0067          004223      TOP  SET   '4223
```

[a]DAP-16 Mod 2 only.

EQU is particularly useful in making the address field of I/O instructions more readable. For example, if the ASR teletypewriter is to be programmed, the following memory aid symbols might be chosen:

```
0009    000004    TIN  EQU   '4       SET INPUT MODE
0010    000104    TOUT EQU   '104     SET OUTPUT MODE
0011    000004    TRDY EQU   '4       SKIP IF READY
0012    000104    TNBS EQU   '104     SKIP IF NOT BUSY
0013    001004    TINA EQU   '1004    CLEAR A AND INPUT ASCII
0014    000004    TOTA EQU   '4       OUTPUT ASCII
```

DATA-DEFINING PSEUDO-OPERATIONS

✓DAC, Address Constant

LOCATION	OPERATION	OPERAND
Normal	DAC or DAC*	Normal. Indexing may be specified.

The low-order 14 bits of address generated from the address field of a DAC pseudo-operation is combined with the indirect bit (if specified by an asterisk after DAC) and index bit (if specified by , 1 after the address). Relocatable addresses are relocated during loading. If extended desectorizing has been specified with EXD, the loader will form 15-bit instead of 14-bit addresses (without regard to the index bit). Thus, the programmer must be careful in using address constants with the index bit set. A 14-bit number with indirect and index bits, or a 15-bit number with indirect bits, is generated by the loader for any positive expression or negative relocatable expression. A 16-bit negative number is generated for negative absolute expressions.

There is no provision for literal address constants. Thus, a DAC must be used and given a symbolic value for each indirect reference. For example, to transfer the address of location FIND to location PUT, the following statement must be written:

```
0110 03617   0 02 03045        LDA   ADDR
0111 03620   0 04 03300        STA   PUT
                                 •
                                 •
                                 •
0115 03045   0 003307    ADDR  DAC   FIND
```

The following example shows address constants used in several ways. This sequence works properly only for programs operating in the normal addressing mode, because the desired post indexing is specified in the address constants. The example moves 10 words from a buffer specified by the calling sequence to a buffer in the example program.

```
                                   *     SAMPLE CALLING SEQUENCE FOR TRANSFER SUBROUTINE
                                   *       (NORMAL ADDRESSING)
     03355    0 10 05375                 JST    TRNS               CALL TRANSFER SUBROUTINE
     03356    1 003372                   DAC    BUF1+10,1          INDEXED POINTER
                                   *                                 TO FIRST BUFFER
     03357    0 01 03374                 JMP    CONT               CONTINUE AT CONT
     03360                         BUF1  BSS    10                 FIRST BUFFER
                                           .
                                           .
                                           .
                                   *     TRANSFER SUBROUTINE
     05375   -0 000000             TRNS  DAC*   **                 TRANSFER SUBROUTINE
                                   *                                 ENTRY POINT.  HAS
                                   *                                 INDIRECT FLAG SET.
     05376    0 35 05424                 LDX    =-10               TEN TRANSFERS WILL BE MADE
     05377   -0 02 05375           LOOP  LDA*   TRNS               PICK UP WORD USING IN-
                                   *                                 DIRECT AND INDEXED DAC
     05400   -0 04 05410                 STA*   AC1                STORE IN BUFFER, USING
                                   *                                 ANOTHER INDIRECT,
                                   *                                 INDEXED DAC
     05401    0 12 00000                 IRS    0                  UPDATE INDEX USED FOR
                                   *                                 BOTH BUFFERS
     05402    0 01 05377                 JMP    LOOP               CONTINUE IF NOT DONE
     05403    0 02 05375                 LDA    TRNS               PICK UP RETURN POINTER
     05404    140100                     SSP                       REMOVE INDIRECT FLAG
     05405    141206                     AOA                       INCREMENT TO POINT TO
                                   *                                 RETURN POINT
     05406    0 04 05423                 STA    TEMP               STORE IT
     05407   -0 01 05423                 JMP*   TEMP               RETURN TO RETURN POINT
                                   *
     05410    1 005423             AC1   DAC    BUF2+10,1          INDEXED POINTER
     05411                         BUF2  BSS    10                 SECOND BUFFER
     05423    000000               TEMP  BSZ    1                  TEMPORARY POINTER LOCATION
```

The following example shows this same subroutine rewritten for operation in extended addressing. Notice that indexing must now be specified in the instruction rather than the address constant.

```
                                   *     SAMPLE CALLING SEQUENCE FOR TRANSFER SUBROUTINE
                                   *       (EXTENDED ADDRESSING)
     03355    0 10 05375                 JST    TRNS               CALL TRANSFER SUBROUTINE
     03356    0 003372                   DAC    BUF1+10            POINTER TO FIRST BUFFER
     03357    0 01 03374                 JMP    CONT               CONTINUE AT CONT
     03360                         BUF1  BSS    10                 FIRST BUFFER
                                           .
                                           .
                                           .
                                   *     TRANSFER SUBROUTINE
     05375   -0 000000             TRNS  DAC*   **                 TRANSFER SUBROUTINE
                                   *                                 ENTRY POINT.  HAS
                                   *                                 INDIRECT FLAG SET.
     05376    0 35 05424                 LDX    =-10               TEN TRANSFERS WILL BE MADE
     05377   -1 02 05375           LOOP  LDA*   TRNS;1             PICK UP WORD USING IN-
                                   *                                 DIRECT DAC WITH
                                   *                                 POST-INDEX
     05400   -1 04 05410                 STA*   AC1,1              STORE IN BUFFER, USING
                                   *                                 ANOTHER INDIRECT
                                   *                                 DAC WITH POST-INDEX
     05401    0 12 00000                 IRS    0                  UPDATE INDEX USED FOR
                                   *                                 BOTH BUFFERS
     05402    0 01 05377                 JMP    LOOP               CONTINUE IF NOT DONE
     05403    0 02 05375                 LDA    TRNS               PICK UP RETURN POINTER
     05404    140100                     SSP                       REMOVE INDIRECT FLAG
     05405    141206                     AOA                       INCREMENT TO POINT TO
                                   *                                 RETURN POINT
     05406    0 04 05423                 STA    TEMP               STORE IT
     05407   -0 01 05423                 JMP*   TEMP               RETURN TO RETURN POINT
                                   *
     05410    0 005423             AC1   DAC    BUF2+10            POINTER
     05411                         BUF2  BSS    10                 SECOND BUFFER
     05423    000000               TEMP  BSZ    1                  TEMPORARY POINTER LOCATION
```

Address constants may also be used to define ranges by subtraction. In this case, the only restriction is that the result must be a positive number less than 16,384 (or 32,768 if the program is being loaded with extended addressing). In the following example, the assembler calculates the length of the buffer and enters it as the first word.

```
0054            000100     LNGT EQU    '100
                                 .
                                 .
                                 .
0057 01341      0 000100   BUFF DAC    LAST-BUFF+1
0058 01342      000000          BSZ    LNGT-2
0059 01440      000000     LAST BSZ    1
```

(The BSZ pseudo-operation is described in this section.) Notice that the length of the buffer has been specified to the assembler by LNGT earlier (using EQU or SET).

✓DEC, Decimal Constant;
DBP, Double Precision Constant

LOCATION	OPERATION	OPERAND
Normal	DEC or DBP	One or more subfields, each containing a decimal data item. As many subfields can be used as can fit in columns 12-72, but no more than 29 words can be generated.

These pseudo-operations, DEC and DBP, cause DAP-16 to convert each subfield to one, two, or three words of binary data with the desired value in either fixed-point or floating-point format. As each subfield is encountered, the next successive memory location is used. Subfields are separated by commas.

The addition and subtraction operations may be used in DEC and DBP address subfields, for example:

```
0119 00633   002010        DEC    1024+8
```

The DBP pseudo-operation is identical to the DEC pseudo-operation, except that in all cases two words are generated and the first word is always in an even memory location. This allows constants generated by DBP to be loaded and stored using DLD and DST of the High-Speed Arithmetic Option. The loader maintains the double-word boundary alignment.

Figure 3-1 shows the general format of numerical values for DEC and DBP. Table 3-2 summarizes subfield conversions for DEC or DBP. Further details on writing subfields for either DEC or DBP follow Table 3-2.

(±) n...n [.n...n] [E / EE (±) mm] [B / BB (±) PP]

mantissa or fraction — characteristic or exponent — scale factor

Figure 3-1. General Format for Numerical Values

TABLE 3-2 SUBFIELD CONVERSIONS FOR DEC AND DBP PSEUDO-OPERATIONS

Condition	DEC Pseudo-Op	DBP Pseudo-Op
✓ 1. No decimal point, B, or E (B15 assumed) or B (with or without decimal point, E, or EE)	Fixed, 1 word	Fixed, 2 words[1]
2. BB (with or without decimal point, E, or EE)	Fixed, 2 words	Fixed, 2 words
3. Decimal point, no B or E or E, no B (with or without decimal point)	Floating, 2 words	Floating 2 words
4. EE, no B (with or without decimal point)	Floating, 3 words	Floating, 2 words[2]

[1]The second word is always '000000.

[2]No third word is generated when EE is used with DBP.

Use of Plus and Minus Signs. -- A plus or minus sign (unary operator) may be used before any number in a DEC or DBP subfield (including the numbers which follow B or E). The plus sign is always optional.

Use of B (Binary Point Position). -- The letter B followed by a number is used to specify the location of the binary point in evaluating fixed-point data. The number following the B is the number of positions the binary point is shifted from the standard assumed location between bits 1 and 2. For example, 3B5 means assemble a word with the value of 3 if the binary point is considered to be 5 bits to the right of the standard position (i.e., between bits 6 and 7, see Figure 3-2).

The hardware binary point location between bits 1 and 2 is important only for multiplication and division. The Assembler therefore assumes a binary point following bit 16 (B15) when the B is not specified.

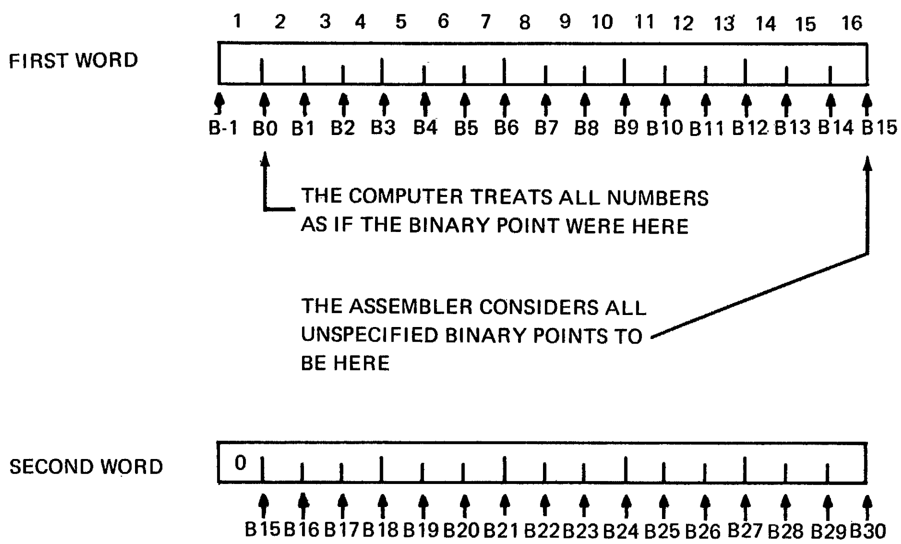


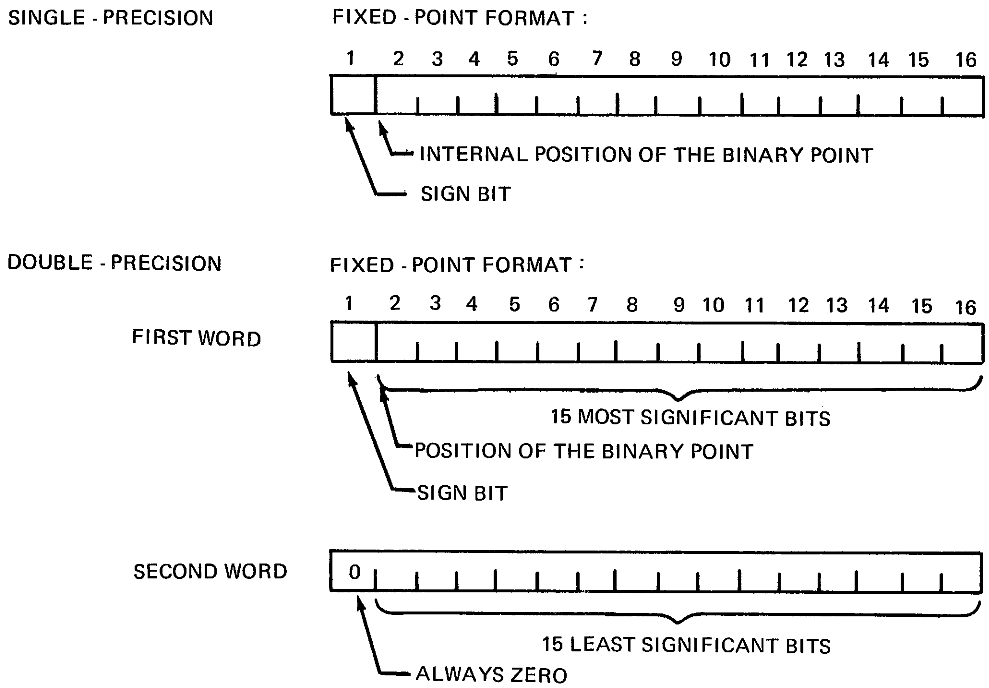
Figure 3-2. Binary Point Position

Use of E (Decimal Point Position). -- The letter E followed by a number is used to specify the position of the decimal point in either fixed-point or floating-point data. The E should be read as "times ten to the...". For example 3E5 means assemble a floating point word with the value of 3×10^5 (300,000). The number following the E is known as the exponent or characteristic, and the value before the E is known as the fraction or mantissa.

Use of the Decimal Point. -- A decimal point may be specified in any floating-point number and some fixed-point numbers. However, it may not be used in the number specifying the exponent or the position of the binary point (that is, following E or B).

✓Fixed-Point Word Formats. -- Figure 3-3 shows the word format for single and double-precision fixed-point words. The central processor always treats fixed-point words as if the binary point were between bits 1 and 2. Negative numbers are in twos-complement form. All bits of a double-precision word except bit 1 of the second word are twos complemented. Bit 1 of the second word is always 0.

Specifying Fixed-Point Data. -- Fixed-point data is specified either by no modifier at all (e.g., 349) or by a B or BB with or without an E or a decimal point (e.g., 349.3B13). B signifies single precision, and BB signifies double precision.



(Negative numbers are represented by two's complement of absolute value. Bit 0 of second word in double-precision is always 0 for both positive and negative numbers.

Figure 3-3. Fixed-Point Word Formats

The effect of B and BB is to move the actual point to an assumed position. B or BB is referred to as a scale factor since it allows the programmer to scale his number to a value more easily handled. The relationship is:

$$N_1 = N_0(2^{-P})$$

where N_1 is the value of the generated word, with the binary point between bits 1 and 2; N_0 is the original value of the number in the DEC, DBP, or literal address field; and P is the value following B or BB. Any low-order bits beyond 15 (or 30) bits of significance are truncated without rounding.

E may also be used in fixed-point numbers if B is present. The formula above is then modified to:

$$N_1 = N_0(2^{-P})(10^X)$$

where N_0, N_1, and P have their former significance and X is the value following E. The DAP-16 Assembler flags an error for any value of N_1 not between -1 and +1.

The following example delineates fixed-point conversions and serves to point out errors. The last four conversions show that there is no rounding in the conversion. The binary approximation to 1/10 (which often appears in conversions) is also shown.

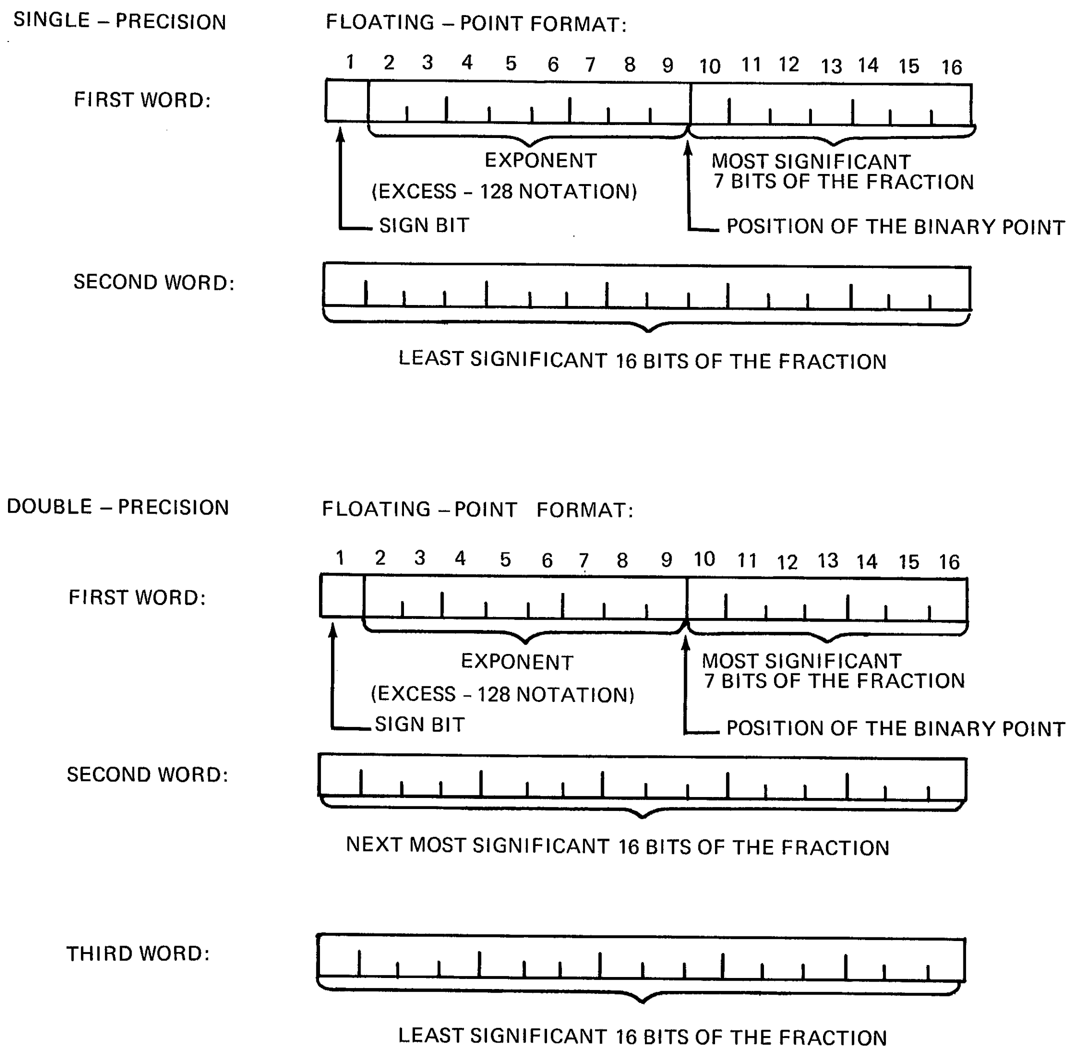
```
     0023 00346   000017        DEC   15           DECIMAL 15 = OCTAL 17
     0024 00347   177761        DEC   -15          NEGATIVE OF FIRST EXAMPLE
     0025 00350   041170        DEC   150E-1       ERROR--RESULT IS FLOATING
          00351   000000
     0026                   *                        POINT (NO B)
     0027 00352   177610        DEC   -15B+12      SECOND EXAMPLE TIMES 8
     0028 00353   000170        DEC   15B12        NEGATIVE OF PREVIOUS
     0029                   *                        EXAMPLE
C    0030 00354   074000        DEC   15B1         ERROR--TOO LARGE
     0031 00355   000170        DEC   150E-1B+12   USE OF BOTH E AND B
     0032 00356   001700        DEC   +0.15E2B9    PREVIOUS EXAMPLE TIMES 8
     0033 00357   000000        DEC   1.5E+1BB21   DOUBLE PRECISION USING DEC
          00360   017000
     0034 00362   000000        DBP   1.5E1BB24    DOUBLE PRECISION USING DBP
          00363   001700
     0035 00364   000000        DBP   15000E-3BB24 SAME AS PREVIOUS EXAMPLE
          00365   001700
C    0036 00366   074000        DBP   15BB+1       ERROR--TOO LARGE
          00367   000000
     0037 00370   000001        DBP   +15BB18      BIT 17 ALWAYS = 0
          00371   070000
VC   0038 00372   000000        DEC   '17B15       ERROR--CANNOT USE B
     0039                   *                        OR E WITH APOSTROPHE
     0040 00373   020000        DEC   0.125B-1     USE OF NEGATIVE B
     0041 00374   001717        DEC   15+.0015E4B9 USE OF ADDITION
     0042                   *   THE FOLLOWING CONVERSIONS SHOW
     0043                   *   TRUNCATION AND THE BINARY VALUE OF 1/10
     0044 00375   000001        DEC   1,1B15
     0045 00376   000001        DEC   1.1BB15
          00377   006314
     0046 00400   000001        DEC   1,99999B15
     0047 00401   000001        DEC   1.99999BB15
          00402   077777
```

Floating-Point Word Formats. --Figure 3-4 presents the format for single- and double-precision floating-point words. Negative numbers are constructed by assembling a positive number and taking the twos complement of the entire two- or three-word number including the exponent.

The exponent is a power-of-two expressed in excess-128 notation. This gives a range between 2^{-127} and 2^{+127} (about 10^{-38} to 10^{+38}). The number zero is represented by using a number of all zero digits.

Specifying Floating-Point Data. -- Floating-point data is specified by an E without a B, an EE without a B, or a decimal point without a B. One E specifies single-precision (two words); two Es specify double-precision (three words).

The DAP-16 Assembler automatically generates the floating-point number with the largest possible (normalized) fraction (< 1). An error is flagged if an exponent with an absolute value greater than 127 is required. Zero is converted to two or three words of all zeros, and excess bits are truncated.



VALUE OF NUMBER IS FRACTION X 2 RAISED TO EXPONENT.
NEGATIVE NUMBERS ARE REPRESENTED BY TWO'S COMPLEMENT
OF ENTIRE POSITIVE NUMBER INCLUDING EXPONENT.

Figure 3-4. Floating-Point Word Formats

The following example illustrates floating-point decimal conversions and serves to point out errors:

```
0003                     *    FLOATING POINT EXAMPLES
0004                     *                                 EXPONENT  FRACTION
0005                     * NO. 1  1/2 TIMES 2 TO THE 0
0006 00223   040100          DEC   0.5                     200       .400...
     00224   000000
0007                     * NO. 2  SLIGHTLY LESS THAN NO. 1
0008 00225   037777          DEC   0.49999999              177       .777...
     00226   177777
0009                     * NO. 3  2S COMPLEMENT OF NO. 1
0010 00227   137700          DEC   -0.5
     00230   000000
0011                     * NO. 4  NO. 1 TIMES 2 TO THE 11
0012 00231   042700          DEC   1.024E3                 213       .400...
     00232   000000
0013                     * NO. 5  NO. 1 TIMES 2 TO THE -2
0014 00233   037500          DEC   125E-3                  176       .400...
     00234   000000
0015                     * NO. 6  PI (IN DOUBLE PRECISION)
0016 00235   040544          DEC   3.1415926535898EE0      202       .62207+
     00236   103755
     00237   050420
0017                     * NO. 7  ERROR--FIXED POINT, NOT FLOATING POINT
0018 00240   040000          DEC   16E0B5              (B IS PRESENT)
```

✔OCT, Octal Constant;
HEX, Hexadecimal Constant[a]

LOCATION	OPERATION	OPERAND
Normal	OCT or HEX	One or more subfields, each containing an octal or hexadecimal data item. As many subfields can be used as can fit in columns 12-72, but no more than 29 words can be generated.

These pseudo-operations, OCT or HEX, cause DAP-16 to convert each subfield to one word of binary data with the desired value. As each subfield is encountered the next successive memory location is used. Subfields are separated by commas.

Octal numbers use the characters 0 through 7, plus, minus, and apostrophe. The apostrophe is redundant but acceptable. Hexadecimal numbers use the character 0 through 9, A through F, plus, minus, and dollar sign. A through F represent decimal numbers 10 through 15 and are contiguous to 0 through 9. The dollar sign is redundant but acceptable. Hexadecimal and octal data may not be mixed in these pseudo-operations.

---

[a]DAP-16 Mod 2 only

The binary point is fixed following bit 16 with both OCT and HEX. However, there is no provision for moving the point with B or E as there is with DEC and DBP. The following example illustrates binary conversions using OCT and HEX:

```
    0018 01021   000015        OCT   15        DECIMAL 13
    0019 01022   000015        OCT   +15       SAME AS FIRST EXAMPLE
    0020 01023   177763        OCT   -15       NEGATIVE OF FIRST EXAMPLE
CVC 0021 01024   000000        OCT   15B2      ERROR--B AND E CANNOT
    0022                   *                     BE USED IN OCT AND HEX
    0023 01025   177763        OCT   177763    SAME AS THIRD EXAMPLE
C   0024 01026   000000        OCT   200000    ERROR--TOO LARGE
    0025 01027   000025        HEX   15        DECIMAL 21
    0026 01030   177753        HEX   -15       NEGATIVE OF PREVIOUS
    0027                   *                     EXAMPLE
    0028 01031   177777        HEX   FFFF      -1
C   0029 01032   073543        HEX   177763    ERROR--TOO LARGE
```

✓BCI, Binary (ASCII) Coded Information

LOCATION	OPERATION	OPERAND
Normal	BCI	A decimal number, N, followed by a comma and 2N alpha-numeric characters. N specifies number of words to be formed and cannot exceed 29.

The BCI pseudo-operation causes DAP-16 to convert each group of two characters to a binary word in USASCII code. A symbol in the location field is assigned to the location of the first word. The words generated are stored in successively higher storage locations as the address field is scanned from the left. The first character of a pair is stored in the most significant bits. Blanks are acceptable characters and do not terminate the address field. The comments field follows the 2Nth character.

The following example shows a conversion of eight words to USASCII. Note that the last two and one-half words contain USASCII blanks ('240). The symbol FINI is assigned to the first word.

```
0056 00027   151305    FINI BCI   8,RELOAD TAPE
     00030   146317
     00031   140704
     00032   120324
     00033   140720
     00034   142640
     00035   120240
     00036   120240
```

VFD, Variable Field Constant[a]

LOCATION	OPERATION	OPERAND
Normal	VFD	Up to 16 pairs of subfields. Each subfield must contain a symbol or expression composed of symbols defined in object program.

[a]DAP-16 Mod 2 only.

The VFD pseudo-operation allows a 16-bit word to be formed, with the programmer having complete control over each bit. The first subfield of a pair specifies the number of bits to be controlled by the next subfield (starting with the most significant end of the word). The second subfield of a pair provides the value to be inserted. This value will be truncated to the number of bits given in the first subfield with no error indication. Each pair of subfields defines one or more bits from the most-significant to the least-significant bits of the word. Unspecified bits at the least-significant portion of the word are filled with zeros. An error indication is given if more than 16 bits are specified. The following examples show data conversions using VFD:

```
    0003 01277   177777          VFD   16,'177777         -1
    0004 01300   106612          VFD   8,'215,8,'212      CARRIAGE RETURN,
    0005                   *                                LINE FEED
    0006 01301   006412          VFD   1,0,7,'215,1,0,7,'212
    0007                   *                              SAME, WITH MSB = 0
    0008                   *                                FOR EACH CHARACTER
    0009 01302   040000          VFD   2,1                BIT 2 ONLY
C   0010 01303   006060          VFD   6,3,6,3,6,3        ERROR--18 BITS
    0011                   *                                SPECIFIED
    0012 01304   100063          VFD   1,1,15,'63         SAME AS DAC* '63
```

STORAGE ALLOCATION PSEUDO-OPERATIONS

BSS, Block Starting With Symbol;
BES, Block Ending With Symbol

LOCATION	OPERATION	OPERAND
Normal	BSS or BES	Normal. Only one subfield allowed. Any symbol used must be previously defined.

These two pseudo-operations, BSS and BES, effectively reserve a block of storage without defining its contents by advancing the location counter. The value in the address field specifies the size of the block in words. If there is a symbolic name in the location field, BSS causes that symbolic name to be assigned to the first location in the block, while BES causes it to be assigned to the first location following the block. In the following two examples a block of storage is defined from '1000 to '1027 inclusive. The symbol BUF is assigned the value '1000 by BSS and '1030 by BES.

```
0073                             ORG    '1000
0074 01000                 BUF   BSS    '30
0075 01030   0 001000            DAC    BUF

0071                             ORG    '1000
0072 01030                 BUF   BES    '30
0073 01030   0 001030            DAC    BUF
```

✓BSZ, Block Storage of Zeros

LOCATION	OPERATION	OPERAND
Normal	BSZ	Normal. Only one subfield allowed. Any symbol used must have been previously defined.

The pseudo-operation BSZ reserves a storage block which is initialized to zeros when the object program is loaded. The first zero location is shown on a DAP-16 Mod 2 Assembly Listing. All zero locations are shown on a DAP-16 Assembly Listing.

COMN, Common Storage

LOCATION	OPERATION	OPERAND
Normal	COMN	Normal. Only one subfield allowed. Any symbol used must be previously defined.

The loader establishes a pool of common values in upper memory using the pseudo-operation COMN. The top of this pool is initialized by the loader but may be moved using SETC (DAP-16 Mod 2 only). The block resulting from each COMN encountered in a program is placed lower in memory than the previous one. (See COMMON Storage below for discussion of DAP-16 and FORTRAN COMMON.)

SETC, Set Common Base[a]

LOCATION	OPERATION	OPERAND
Ignored	SETC	Normal. Only one subfield allowed. Any symbol used must be previously defined.

The loader initializes the COMMON base (the highest location in common) to a location near the top of memory (or the present memory bank in systems with over 16K locations). The SETC pseudo-operation allows another location to be specified. All programs referencing this block of COMMON must use the same value in the address field of SETC.

COMMON Storage

DAP-16 Convention. -- The absolute address assignments are made at the time of assembly. The assembler maintains an internal COMMON base, which is initially set to 'XX600 (where XX is the last sector of memory). It may be reset at any time by the DAP-16 Mod 2 Assembler by the SETC pseudo-operation. When a symbol is defined by a COMN pseudo-operation, the number of locations specified in the address field is subtracted from the current COMMON base. The result is both the address assigned to the symbol and the new COMMON base. Figure 3-5 presents an example of this procedure.

[a]DAP-16 Mod 2

DAP-16 Coding

```
*                                    RESULTING       SYMBOL
* STATEMENT                            BASE         ASSIGNED
*-----------                         --------   ----------------
*                                     '27600    (ORIGINAL VALUE)
C      COMN   2                       '27576      C = '27576
I      COMN   1                       '27575      I = '27575
A      COMN   2                       '27573      A = '27573
```

Storage Allocation Diagram:

'27600	
'27577	C+1
'27576	C
'27575	I
'27574	A+1
'27573	A

Figure 3-5. COMMON Allocation in DAP-16

In the following examples, two programs reference the same COMMON location at the top of sector 6 (location '6776). The first program refers to this location as LBUF, the second as PASS:

```
0066                              SETC   '6777             SET COMMON BASE
0067           006776        LBUF COMN   1                 ONE VALUE NAMED LBUF
0068 00567   0 006776        LDAC DAC    LBUF              POINTER TO LBUF
                                   .
                                   .
0072 00634   0 02 00344           LDA    =1                STORE 1
0073 00635  -0 04 00567           STA*   LDAC                IN LBUF



0079                              SETC   '6777             SAME COMMON BASE
0080           006776        PASS COMN   1                 NOW CALLED PASS
0081 05501   0 006776        PDAC DAC    PASS              POINTER TO PASS
                                   .
                                   .
0085 05525  -0 02 05501           LDA*   PDAC              PICK UP VALUE IN PASS
0086                         *    (=1 IF PREVIOUS PROGRAM WAS THE LAST
0087                         *    TO ACCESS THIS LOCATION)
```

FORTRAN Convention. -- The FORTRAN compiler passes a displacement rather than an absolute address to the loader for each variable in COMMON. The loader determines the address by subtracting the displacement from the COMMON base. This base may be altered when the program is loaded. The displacements assigned by FORTRAN are such that the first variable mentioned has the largest displacement (and is lowest in memory) and the last variable mentioned has the smallest displacement (and is highest in memory). The address assignment may be altered at run time by changing the loader's COMMON base (relative location '2000 in LDR-APM). If the two COMMON statements below are the last COMMON statements in a FORTRAN program, and if the loader COMMON base is set to '27600, these statements will reference the same locations shown in Figure 3-5.[a]

COMMON A, I

COMMON C

Note that variables in COMMON must be named in the opposite order in DAP-16 and FORTRAN.

PROGRAM-LINKING PSEUDO-OPERATIONS

ENT, Entry Point;[b]
✓SUBR, Entry Point

LOCATION	OPERATION	OPERAND
Ignored	ENT or SUBR	One or two subfields containing a name of one to six characters.

ENT and SUBR are two names for the same pseudo-operation. This pseudo-operation usually precedes executable instructions; however, it may be used anywhere. These pseudo-operations cause the assembler to output the symbolic name from the address field in the object text. Its value at load time can then be saved by the loader for use by other programs (via EXT, XAC, or CALL). The loader starts loading a CALLed subroutine from the point where the programmer placed the ENT or SUBR. Thus, it is possible to bypass the beginning of a subroutine. If there are two names in the address field these names are considered synonyms within the assembler. DAP-16 looks for the value of the second name in the symbol table and assigns that value to the first name for use by other programs. Although only four characters are used for names within a program, up to six characters may be communicated between programs. The extra one or two characters are ignored when searching the symbol table for a value.

The following is an example routine with three entry points. Other programs may call the first entry using either SINE or SINF. The second entry may only be called COSINE. The third entry may only be called ARCTAN. This entry point has been placed following the SINE and COSINE entry points, because the ARCTAN routine uses none of the instructions above its entry point.

---

[a] A and C are FORTRAN Real Variables occupying two words; I is an Integer Variable occupying only one word.

[b] ENT is supported only in DAP-16 Mod 2.

```
0077                               SUBR  SINE          NAME FOR SINE ROUTINE
0078                               ENT   SINF,SINE     ALTERNATE NAME FOR
0079                          *                          SINE ROUTINF
0080                               ENT   COSINE        NAME FOR COSINE ROUTINE
0081 00543   0 000000   SINE DAC   **                  START OF SINE ROUTINE
                                   .
                                   .
0085 00630  -0 01 00543       JMP*  SINE               EXIT FROM SINF ROUTINE
0086 00631   0 000000   COSI  DAC   **                 START OF COSINE ROUTINE
                                   .
                                   .
0090 00662  -0 01 00631       JMP*  COSI               EXIT FROM COSINF ROUTINF
0091                          SUBR  ARCTAN,ATAN        NAME FOR ARCTAN ROUTINE
0092 00663   0 000000   ATAN  DAC   **                 START OF ARCTAN ROUTINE
                                   .
                                   .
0096 00705  -0 01 00663       JMP*  ATAN               EXIT FROM ARCTAN ROUTINF
```

EXT, External Name[a]

LOCATION	OPERATION	OPERAND
Ignored	EXT	A name of one to six characters.

The EXT pseudo-operation signals the loader that the name in the address field is not defined in this program. An error is flagged if executable instructions preceed EXT, but this error may have no effect on the object text. If the name is referenced later in the program the loader will make the proper linkage. Loading will not be complete until a subroutine using the name in an ENT or SUBR pseudo-operation has been loaded. In the example below, the loader is informed that a program defining SRTE as an accessible location via ENT or SUBR must be linked to this one:

```
0002                       EXT   SRTE
                            .
                            .
0006 00070   0 02 00000    LDA   SRTE
```

✓XAC, External Address Constant

LOCATION	OPERATION	OPERAND
Normal	XAC or XAC*	Any external subroutine name. Indexing may be specified.

The XAC pseudo-operation is the same as the DAC pseudo-operation, except that the loader fills the low-order 14 bits (15 if extended desectorizing has been specified) with the address of an external name specified by another program.

EXT allows the programmer to treat an external name as if it were part of the current program. XAC performs the same function but, in addition, allows the programmer to control the location of the indirect link.

[a]DAP-16 Mod 2 only.

✓CALL, Call Subroutine

LOCATION	OPERATION	OPERAND
Normal	CALL	Any external subroutine name.

The CALL pseudo-operation simultaneously specifies a JST operation and EXT pseudo-operation (which is effective, however, only for the processing of that one statement).

The following examples link two programs. A JST is inserted in location ARC linking (indirectly if necessary) to the entry point ARCTAN of another subroutine. In the second example, the name ARCTAN is valid throughout the program, but in the remaining examples it is valid only in the statement shown.

```
0091 01672    0 10 00000 ARC   CALL   ARCTAN


0002                           EXT    ARCTAN
                                :
0006 01672    0 10 00000 ARC   JST    ARCTAN


0083 01672   -0 10 01715 ARC   JST*   ARCT
                                :
0087 01715    0 000000   ARCT  XAC    ARCTAN
```

CONDITIONAL ASSEMBLY PSEUDO-OPERATIONS[a]

IFP, Assemble Only if Plus;
IFM, Assemble Only if Minus;
IFZ, Assemble Only if Zero;
IFN, Assemble Only if Not Zero

LOCATION	OPERATION	OPERAND
Ignored	IFP, IFM, IFZ, or IFN	Normal. Only one subfield allowed. Any symbol used must be previously defined.

The address field is evaluated at assembly time. If the condition specified by the operation field is not met, assembly is inhibited until an ELSE or ENDC is encountered. Otherwise, assembly continues uninterrupted. In the following example assembly would always be inhibited:

```
0092                 IFZ   1
```

Assembly would be inhibited in the following example if symbolic name NAM2 has a smaller value than symbolic name NAM1.

```
0097                 IFM   NAM1-NAM2
```

See Using Conditional Assembly on the following page for further details.

[a]Conditional assembly is supported only in DAP-16 Mod 2.

ENDC, End of Conditional Assembly

LOCATION	OPERATION	OPERAND
Ignored	ENDC	Ignored

The ENDC pseudo-operation removes the effect of a preceeding IF statement with which it is paired. When conditions are nested this fact may not restore inhibited assembly. A Z-error is flagged if the END statement is reached before all IFs have been matched by ENDCs.

ELSE, Combined IF and ENDC

LOCATION	OPERATION	OPERAND
Ignored	ELSE	Ignored

The ELSE pseudo-operation is used as a switch between inhibited and uninhibited assembly and has the following effects.

a. Between any IF and an ENDC when assembly is not inhibited, ELSE acts as

```
0111          ENDC
0112          IFN    0
```

That is, it matches the previous IF statement and generates a new statement that inhibits assembly.

b. Between any IF and an ENDC when assembly is inhibited, ELSE acts as

```
0096          ENDC
0097          IFZ    0
```

That is, it removes the inhibition unless this IF/ENDC pair is nested within another statement that is causing the inhibition.

c. A Z-error is flagged if ELSE is used anywhere other than between an IF and an ENDC.

FAIL, Identifies Statement Which Should Never Be Assembled

LOCATION	OPERATION	OPERAND
Ignored	FAIL	Ignored

The FAIL pseudo-operation causes an O-error and is used in conditional assemblies to ensure that the conditions are logically consistent.

Using Conditional Assembly

Conditional assembly allows a comprehensive source program to be written covering many conditions. Parameters are passed using SET or EQU pseudo-operations at the beginning of the program to effect different assemblies for different objects. These statements can control the variables used by Conditional Assembly statements and consequently cause assembly of only those parts of the program necessary to this objective.

The following four examples show the same program assembled in four ways. Four parameters, V1, V2, V3, and V4 control the assembly. Note that nothing is assembled if V1 = V2. If V1 if greater than V2, only the FAIL pseudo-operation is assembled, otherwise some combination of instructions is assembled.

In the routine below V1 = 1, V2 = 3, V3 = 1, and V4 = 0. First is a listing showing both assembled and skipped lines listed (see Performing an Assembly).

```
                                *     EXAMPLE OF CONDITIONAL ASSEMBLY
              000001            V1    EQU     1
              000003            V2    EQU     3
              000001            V3    EQU     1
              000000            V4    EQU     0
                                      IFN     V1-V2
                                      IFP     V1-V2
                                      FAIL
                                      ELSE
      00337   0 10 00375              JST     A3X
                                      IFZ     V1-V3+V4
      00340   0 10 00452              JST     B3X
      00341   0 04 00665              STA     TEMP
                                      ELSE
                                      JST     C3X
                                      IFM     V2-V4
                                      JST     D3X
                                      ADD     TEMP
                                      STA     TEMP
                                      ELSE
                                      LDA     =-1
                                      ADD     TEMP
                                      STA     TEMP
                                      ENDC
                                      ERA     TTST
                                      SNZ
                                      ELSE
      00342   100400                  SPL
                                      ENDC
      00343   0 01 00301              JMP     X1
                                      ENDC
                                      ENDC
```

The following example shows the same routine assembled without listing the skipped statements.

```
0112                           *     EXAMPLE  OF  CONDITIONAL  ASSEMBLY
0113             000001        V1    EQU     1
0114             000003        V2    EQU     3
0115             000001        V3    EQU     1
0116             000000        V4    EQU     0
0121  00337      0 10  00375         JST     A3X
0123  00340      0 10  00452         JST     B3X
0124  00341      0 04  00665         STA     TEMP
0139  00342      100400              SPL
0141  00343      0 01  00301         JMP     X1
```

The following example shows the same routine assembled using a different set of parameters without listing the skipped statements.

```
0124                           *     EXAMPLE  OF  CONDITIONAL  ASSEMBLY
0125             000001        V1    EQU     1
0126             000003        V2    EQU     3
0127             000000        V3    EQU     0
0128             000001        V4    EQU     1
0133  00337      0 10  00375         JST     A3X
0138  00340      0 10  00462         JST     C3X
0144  00341      0 02  00347         LDA     =-1
0145  00342      0 06  00665         ADD     TEMP
0146  00343      0 04  00665         STA     TEMP
0148  00344      0 05  00666         ERA     TTST
0149  00345      101040              SNZ
0153  00346      0 01  00301         JMP     X1
```

In the following example VI is greater than V2.

```
     0101                           *     EXAMPLE  OF  CONDITIONAL  ASSEMBLY
     0102             000007        V1    EQU     7
     0103             000003        V2    EQU     3
     0104             000000        V3    EQU     0
     0105             177770        V4    EQU     -8
O    0108                                 FAIL
```

SPECIAL SYMBOLS

***, Op Code Zero;
PZE, Op Code Zero

LOCATION	OPERATION	OPERAND
Normal	***, ****, PZE, or PZE*	Normal. Indexing may be specified.

These two pseudo-operations, *** and PZE, are assembled and loaded as memory reference instructions with an operation code of zero. Indirect addressing and indexing may be specified. The sector bit is set or reset depending on the sector in which the address is located. Since there is no memory reference instruction with an operation code of zero, it is expected that the proper code will be inserted during program execution and before attempting to execute this instruction.

ERROR CODE

The DAP-16 Assembler is able to detect various types of syntax errors commonly made during the coding of programs. These errors are indicated by one-letter error codes printed in the left margin of the assembly listing (see Figure 2-4 for an example).

Each error is treated differently; some result in zero in the erroneous field, others result in a guess at the desired result. In the case of multiply defined symbols, the first symbol definition is used. If the operation code is illegal for the object computer configuration indicated, the line will be properly assembled but flagged with an O-error. At the end of the assembly the following message is printed (DAP-16 Mod 2): 0000 WARNING OR ERROR FLAGS (DAP-16 prints NO ERRORS IN ABOVE ASSEMBLY). The number of errors is printed instead of 0000 if there are any (** for DAP-16).

See Table 3-3 for a list of the error flags and their meaning.

TABLE 3-3. WARNING AND ERROR FLAGS

A	Address field missing where normally required; error in address format
C	Erroneous conversion of a constant; address field of data-defining pseudo-operation in improper format
E	Executable code generated before EXT pseudo-operation; external name modified by addition; external name used in address field of something other than a memory reference instruction[a]
F	Major formatting error
L	Label (location field) missing where normally required; error in label symbol[a]
M	Multiply defined symbol
O	Operation field blank or not recognized; operation field not legal for object configuration
P	Phase error (different definitions in first and second passes)[a]
R	Relocation assignment error[a]
S	Address of variable field expression not in sector being processed or sector zero (applicable only in LOAD mode)
T	Improper use of index subfield; error in index subfield
U	Undefined symbol
V	Unclassified error in address field of multiple-subfield pseudo-operation
Z	Conditional assembly error; ELSE used outside of conditional assembly; END reached before all IFs matched by ENDCs[a]

[a]DAP-16 Mod 2 only.

EXAMPLE

Figure 3-6 shows a general flow chart of three programs that convert a binary number to an ASCII octal number and print it on the ASR; the assembled programs and their cross-reference listings are shown in Figures 3-7, 3-8, and 3-9. These three programs use a special format known as a Defined Character Address (DCA) for pointers to half-words. Bits 2 through 16 of the DCA are a pointer (DAC) to the word, and bit 1 tells which half of the word is to be accessed, with 0 meaning the left (high-order) half and 1 meaning the right (low-order) half).

These three programs operate correctly when loaded into core and linked to another program that supplies the number to convert. However, they were designed to show various aspects of assembly language programming and therefore are not as efficient as they could be.

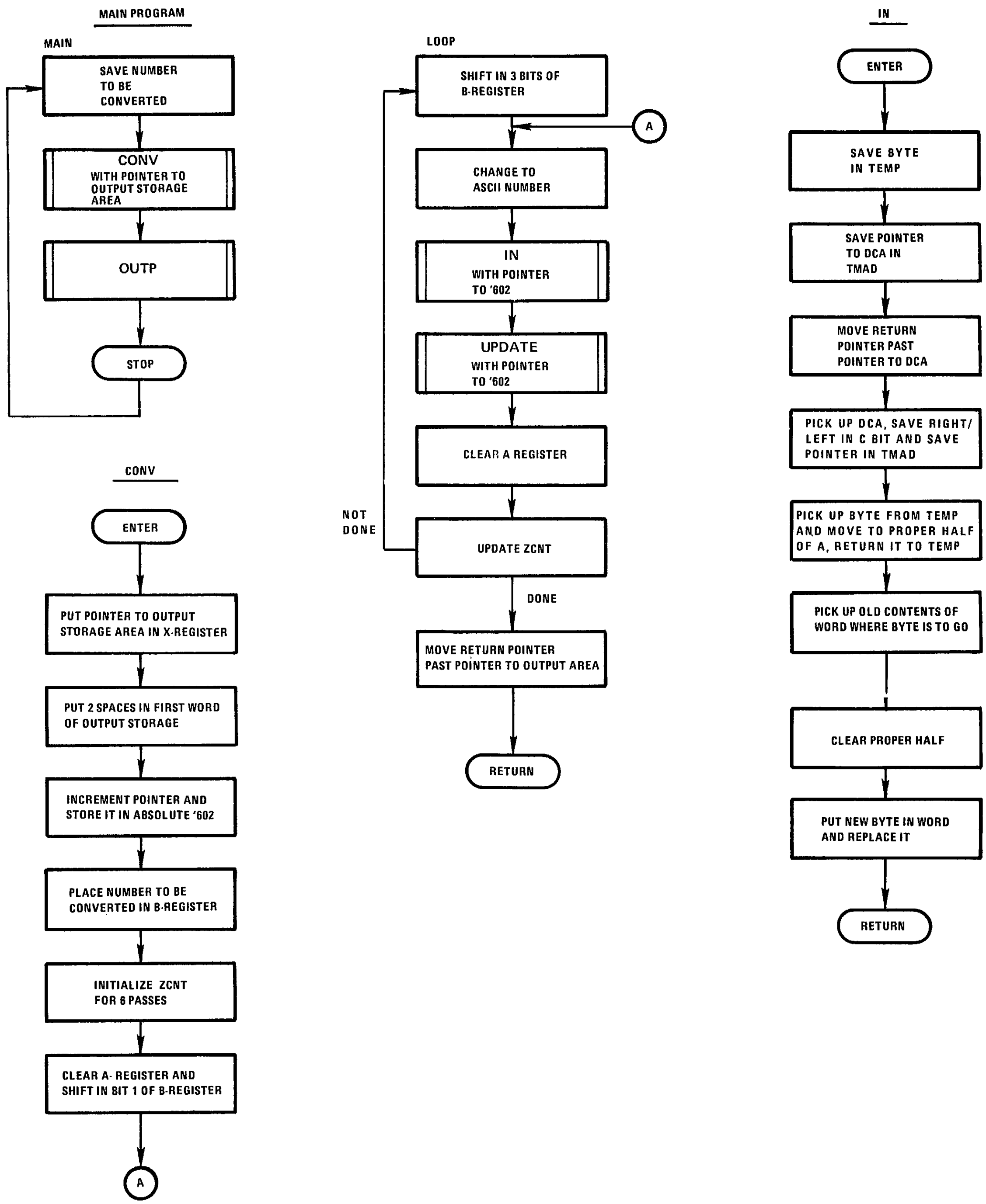


Figure 3-6. Flow Chart for Example in Figures 3-7 thru 3-9 (Part 1 of 2)

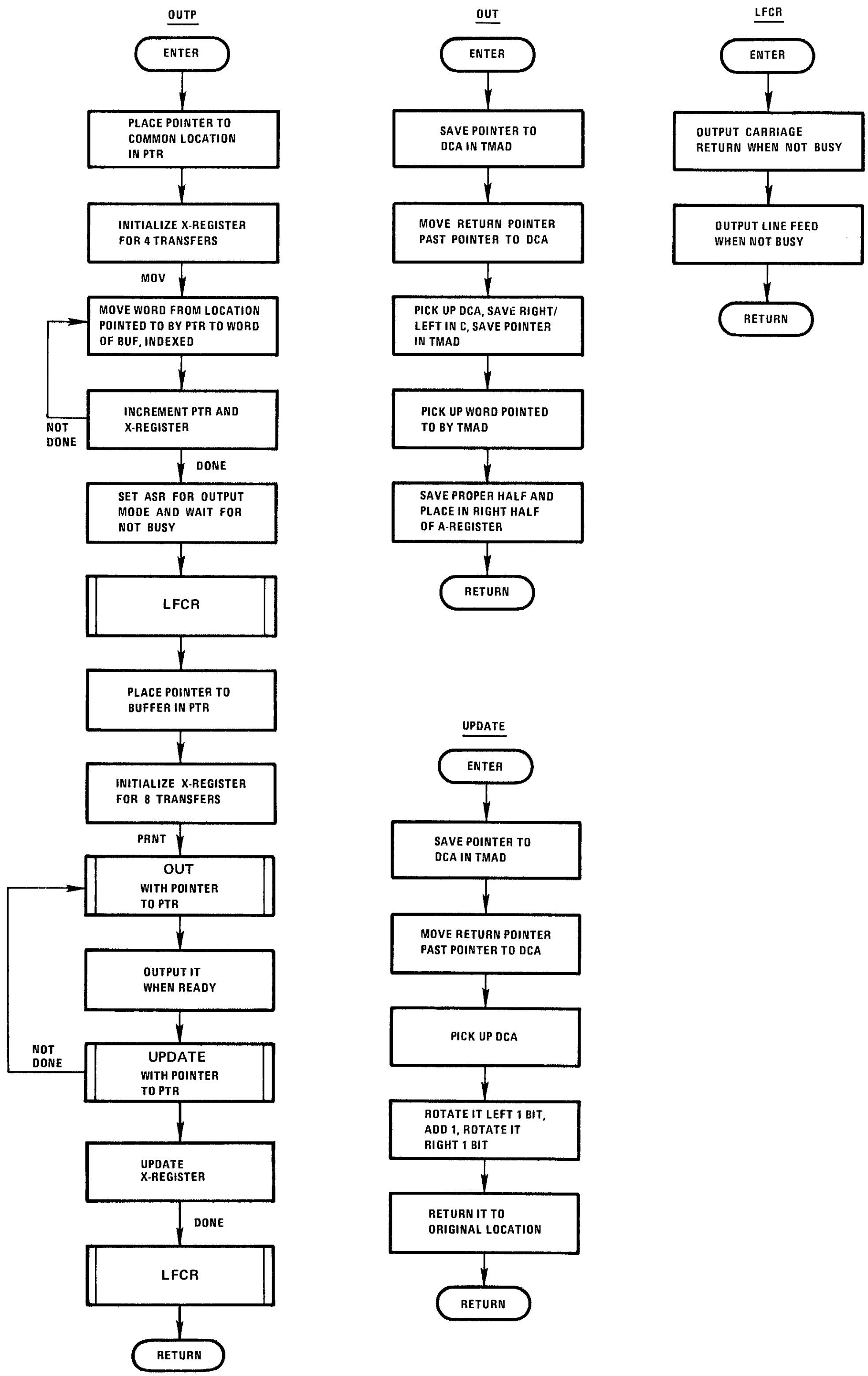


Figure 3-6. Flow Chart for Example in Figures 3-7 thru 3-9 (Part 2 of 2)

```
                         * ASSEMBLER MANUAL EXAMPLE--MAIN SEQUENCE
                         *
                              CF5                        CONFIGURATION IS 516 OR 316
                              ORG    '1000               ABSOLUTE PROGRAM STARTING AT '1000
                         *                                 ENTERS WITH NUMBER TO BE CONVERTED
                         *                                 IN A REGISTER
01000   0 04 00601  MAIN STA    '601                     PUT NUMBER IN ABSOLUTE LOCATION
                         *                                 601 (OCTAL)
01001   0 10 00000        CALL   CONV                    HAVE THE NUMBER CONVERTED
01002   0 037566          DAC    LOC                     STARTING AT THE COMMON
                         *                                 LOCATION DEFINED BELOW
01003   101000            NOP                            CONVERSION PROGRAM RETURNS HERE
                         *                                 (NOP NOT NECESSARY)
01004   0 10 00000        CALL   OUTP                    HAVE THE NUMBER PRINTED
01005   101000            NOP                            RETURN HERE
                         *                                 (NOP NOT NECESSARY)
01006   000000            HLT                            STOP FOR NEW NUMBER
01007   0 01 01000        JMP    MAIN                    WHEN RESTARTED IT JUMPS BACK
                         *                                 TO THE BEGINNING
        037566      LOC   COMN  .10                      POINTER TO LOCATION IN COMMON
                         *
                              ENT    IN,PUT              IDENTIFY EXTERNAL NAMES
                              ENT    OUT,PICK              FOR THESE THREE ROUTINES
                              ENT    UPDATE,DCUP
01010   0 000000    PUT   DAC    **                      ENTRY POINT TO PLACE A CHARACTER
01011   0 04 01063        STA    TEMP                    SAVE CHARACTER
01012  -0 02 01010        LDA*   PUT                     PICK UP POINTER TO DCA
01013   0 12 01010        IRS    PUT                     MOVE RETURN POINT PAST DCA POINTER
01014   0 04 01064        STA    TMAD                    SAVE POINTER
01015  -0 02 01064        LDA*   TMAD                    PICK UP DCA
01016   140320            CSA                            SAVE RIGHT/LEFT BIT IN C BIT
                         *                                 AND CLEAR BIT 1 OF A REGISTER
01017   0 04 01064        STA    TMAD                    STORE ADDRESS OF CHARACTER
01020   0 02 01063        LDA    TEMP                    PICK UP CHARACTER
01021   141050            CAL                            SAVE ONLY RIGHT 8 BITS
01022   101001            SSC                            TEST FOR RIGHT/LEFT
01023   141340            ICA                            INTERCHANGE IF LEFT
01024   0 04 01063        STA    TEMP                    RETURN CHARACTER TO TEMP
01025  -0 02 01064        LDA*   TMAD                    PICK UP OLD CONTENTS OF
                         *                                 CHARACTER ADDRESS
01026   101001            SSC                            CLEAR HALF TO BE MODIFIED
01027   141050            CAL                            (LEFT HALF IF C BIT SET)
01030   100001            SRC
01031   141044            CAR                            (RIGHT HALF IF C BIT RESET
01032   0 05 01063        ERA    TEMP                    ADD IN NEW CHARACTER
01033  -0 04 01064        STA*   TMAD                    REPLACE IT IN ORIGINAL LOCATION
01034  -0 01 01010        JMP*   PUT                     RETURN
                         *
01035   0 000000    PICK  DAC    **                      ENTRY POINT TO GET A CHARACTER
```

Figure 3-7. Example, Main Sequence (Part 1 of 3)

```
0050 01036  -0 02 01035       LDA*  PICK           PICK UP POINTER TO DCA
0051 01037   0 12 01035       IRS   PICK           MOVE RETURN POINT PAST DCA POINTER
0052 01040   0 04 01064       STA   TMAD           SAVE POINTER
0053 01041  -0 02 01064       LDA*  TMAD           PICK UP DCA
0054 01042   140320           CSA                  SAVE RIGHT/LEFT BIT IN C BIT
0055                     *                           AND CLEAR BIT 1 OF A REGISTER
0056 01043   0 04 01064       STA   TMAD           STORE ADDRESS OF CHARACTER
0057 01044  -0 02 01064       LDA*  TMAD           PICK UP WORD CONTAINING CHARACTER
0058 01045   101001           SSC                  TEST FOR RIGHT OR LEFT HALF
0059 01046   141340           ICA                  INTERCHANGE HALVES IF LEFT
0060 01047   141050           CAL                  CLEAR LEFT HALF
0061 01050  -0 01 01035       JMP*  PICK           RETURN
0062                     *
0063 01051   0 000000    DCUP DAC   **             ENTRY POINT TO UPDATE A DCA
0064 01052  -0 02 01051       LDA*  DCUP           PICK UP POINTER TO DCA
0065 01053   0 12 01051       IRS   DCUP           MOVE RETURN POINT PAST DCA POINTER
0066 01054   0 04 01064       STA   TMAD           SAVE POINTER
0067 01055  -0 02 01064       LDA*  TMAD           PICK UP DCA
0068 01056   0416 77          ALR   1              ROTATE TO PUT RIGHT/LEFT BIT
0069                     *                           IN BIT 1
0070 01057   141206           AOA                  INCREMENT IT
0071 01060   0406 77          ARR   1              ROTATE BACK TO ORIGINAL POSITION
0072 01061  -0 04 01064       STA*  TMAD           STORE IT
0073 01062  -0 01 01051       JMP*  DCUP           RETURN
0074                     *
0075 01063   000000      TEMP BSZ   2              TEMPORARY LOCATIONS
0076         001064      TMAD EQU   TEMP+1           FOR THESE ROUTINES
0077                     *
0078                          END

DCUP   001051A   LOC    037566A   MAIN   001000A   PICK   001035A
PUT    001010A   TEMP   001063A   TMAD   001064A

0000 WARNING OR ERROR FLAGS
DAP-16 MOD 2    REV. B    10-20-70
```

Figure 3-7. Example, Main Sequence (Part 2 of 3)

```
      CONV          9
63    DCUP          24     64     65C    73J
      IN            22
20    LOC           10
 7    MAIN          18J
      OUT           23
      OUTP          14
49    PICK          23     50     51C    61J
25    PUT           22     27     28C    47J
75    TEMP          26C    34     38C    45     76
76    TMAD          29C    30     33C    39     46C    52C
                    53     56C    57     66C    67     72C
      UPDATE        24


12    SYMBOLS
36    REFERS
78    RECORDS

016-XREF 05 OCT 70
```

Figure 3-7. Example, Main Sequence (Part 3 of 3)

```
                         * ASSEMBLER MANUAL EXAMPLE--CONVERSION ROUTINE
                         *
                                SUBR   CONV        NAME IS CONV
                                CF5                CONFIGURATION IS 316 OR 516
                                EXT    IN          IN AND UPDATE
                                EXT    UPDATE        ARE REFERENCED EXTERNAL ROUTINES
                                REL                RELOCATABLE PROGRAM
00000   0 000000         CONV   DAC    **          ENTRY POINT
00001  -0 35 00000              LDX*   CONV        PUT POINTER TO OUTPUT BUFFER
                         *                           IN X REGISTER
00002   0 02 00605              LDA    ='120240    LOAD 2 SPACES INTO A REGISTER
00003   1 04 00000              STA    0,1         STORE THE 2 SPACES IN THE
                         *                           FIRST WORD OF THE OUTPUT BUFFER
00004   0 12 00000              IRS    0           POINT TO THE NEXT WORD
00005   0 15 00602              STX    Z602        SAVE POINTER  TO BUFFER IN Z602
00006   0 02 00601              LDA    Z601        PICK UP THE NUMBER TO BE CONVERTED
                         *                           (WHICH WAS PUT HERE BY CALLING PROGRAM)
00007   000201                  IAB                PUT IN B REGISTER
00010   0 02 00604              LDA    =-6         INITIALIZE ZCNT FOR
00011   0 04 00600              STA    ZCNT          6 PASSES THROUGH THE CONVERSION
00012   140040                  CRA                ZERO A REGISTER.
00013   0410 77                 LLL    1             MOVE LEFTMOST BIT OF NUMBER TO A,
00014   100000                  SKP                  AND SKIP OVER 3-BIT SHIFT
00015   0410 75          LOOP   LLL    3           MOVE 3 BITS TO A
00016   0 06 00603              ADD    ='260       ADD ASCII ZERO, GIVING ASCII 0-7
00017   0 10 00000              JST    IN          PUT THE BYTE IN THE BUFFER
00020   0 000602                DAC    Z602          USING POINTER IN Z602
00021   0 10 00000              JST    UPDATE      UPDATE THE POINTER
00022   0 000602                DAC    Z602          IN Z602
00023   140040                  CRA                CLEAR A REGISTER BEFORE SHIFT
00024   0 12 00600              IRS    ZCNT        UPDATE THE LOOP COUNTER
00025   0 01 00015              JMP    LOOP        LOOP UNTIL DONE
00026   0 12 00000              IRS    CONV        MOVE RETURN POINT PAST BUFFER POINTER
00027  -0 01 00000              JMP*   CONV        RETURN TO CALLING PROGRAM
                         *
                         RET    ORG    '600        NOW ESTABLISH SOME LOCATIONS
                                ABS                  AND ALL LITERALS IN SECTOR 0
00600   000000           ZCNT   BSZ    1           COUNT OF CHARACTERS
00601   000000           Z601   BSZ    1           CALLING PROGRAM PUTS NUMBER HERE
00602   000000           Z602   BSZ    1           POINTER TO RESULT STORAGE LOCATIONS
00603   000260                  FIN                NOW ASSEMBLE LITERALS
00604   177772
00605   120240

                                ORG    RET         AND RETURN TO MAIN SEQUENCE SO THAT
                                REL                  SUCCEEDING PROGRAMS WILL LOAD IN
                         *                           THE RIGHT PLACE
                                END
```

Figure 3-8 Example, Conversion Routine (Part 1 of 2)

```
CONV   000000    IN     000000E   LOOP   000015   RET    000030
UPDATE 000000E   Z601   000601A   Z602   000602A  ZCNT   000600A

0000 WARNING OR ERROR FLAGS
DAP-16 MOD 2    REV. B    10-20-70


  8  CONV            3     9     33C    34J
     IN              5     26J
 24  LOOP            32J
 36  RET             42
     UPDATE          6     28J
 39  Z601            16
 40  Z602            15C   27    29
 38  ZCNT            20C   31C
     ='120240        11
     ='260           25
     =-6             19


 11  SYMBOLS
 19  REFERS
 45  RECORDS

016-XREF 05 OCT 70
```

Figure 3-8. Example, Conversion Routine (Part 2 of 2)

```
                              * ASSEMBLER MANUAL EXAMPLE--OUTPUT ROUTINE
                              *
                                   SUBR  OUTP,WRIT        EXTERNAL NAME IS OUTP,
                              *                             INTERNAL NAME IS WRIT
                                   CF5                    CONFIGURATION IS 516 OR 3.6
                                   EXT   OUT              OUT AND UPDATE ARE
                                   EXT   UPDATE             REFERENCED EXTERNAL ROUTINES
                                   REL                    RELOCATABLE PROGRAM
00000   0 000000              WRIT DAC   **               ENTRY POINT
00001   0 02 00050                 LDA   WLOC             PLACE POINTER TO COMMON LOCATION
00002   0 04 00046                 STA   PTR                OF MESSAGE IN TEMPORARY LOCATION
00003   0 35 00054                 LDX   =-4              INITIALIZE TO MOVE 4 WORDS
00004  -0 02 00046            MOV  LDA*  PTR              PICK WORD FROM COMMON
00005   1 04 00046                 STA   BUF+4,1          STORE IN BUF
00006   0 12 00046                 IRS   PTR              UPDATE SOURCE POINTER
00007   0 12 00000                 IRS   0                UPDATE DESTINATION POINTER AND
                              *                             SKIP IF ALL WORDS HAVE BEEN MOVED
00010   0 01 00004                 JMP   MOV              MOVE ANOTHER WORD IF ALL FOUR HAVEN'T
                              *                             BEEN MOVED
00011   14 0104                    OCP   '104             SET ASR FOR OUTPUT MODE
00012   34 0004                    SKS   4                TEST FOR ASR BUSY
00013   0 01 00012                 JMP   *-1              TEST AGAIN IF BUSY
00014   0 10 00032                 JST   LFCR             NOT BUSY--OUTPUT A CARRIAGE
                              *                             RETURN AND LINE FEED
00015   0 02 00047                 LDA   BDAC             PLACE POINTER TO BUFFER
00016   0 04 00046                 STA   PTR                IN TEMPORARY DCA
                              *                             (NOW POINTS TO LEFTMOST BYTE OF BUF)
00017   0 35 00053                 LDX   =-8              INITIALIZE FOR 8 CHARACTERS
00020   0 10 00000            PRNT JST   OUT              GET THE BYTE
00021   0 000046                   DAC   PTR                POINTED TO BY PTR
00022   74 0004                    OTA   4                OUTPUT IT
00023   0 01 00022                 JMP   *-1              DELAY UNTIL OUTPUT ACCEPTED
00024   0 10 00000                 JST   UPDATE           UPDATE PTR
00025   0 000046                   DAC   PTR                TO POINT TO NEXT BYTE
00026   0 12 00000                 IRS   0                UPDATE LOOP COUNTER
00027   0 01 00020                 JMP   PRNT             CONTINUE
00030   0 10 00032                 JST   LFCR             WHEN DONE, OUTPUT A CARRIAGE
                              *                             RETURN AND LINE FEED
00031  -0 01 00000                 JMP*  WRIT             RETURN TO CALLING PROGRAM
                              *
00032   0 000000              LFCR DAC   **               INTERNAL ENTRY FOR LINE FEED/
                              *                             CARRIAGE RETURN OUTPUT
00033   0 02 00052                 LDA   ='215            CARRIAGE RETURN TO A
00034   74 0004                    OTA   4                OUTPUT IT
00035   0 01 00034                 JMP   *-1              DELAY
00036   0 02 00051                 LDA   ='212            LINE FEED TO A
00037   74 0004                    OTA   4                OUTPUT IT
00040   0 01 00037                 JMP   *-1              DELAY
00041  -0 01 00032                 JMP*  LFCR             RETURN TO MAIN SEQUENCE
```

Figure 3-9. Example, Output Routine (Part 1 of 2)

```
0050                             *
0051 00042                       BUF  BSS   4              PRINT FROM THIS BUFFER
0052 00046   000000              PTR  BSZ   1              TEMPORARY POINTER
0053 00047   0 000042            BDAC DAC   BUF            POINTER TO TEMPORARY BUFFER
0054 00050   0 037566            WLOC DAC   C              POINTER TO COMMON LOCATION
0055         037566              C    COMN  10               WHERE MESSAGE IS FIRST FOUND
0056 00051   000212                   END
     00052   000215
     00053   177770
     00054   177774


BDAC   000047   BUF   000042   C     037566A   LFCR   000032
MOV    000004   OUT   000000E  PRNT  000020    PTR    000046
UPDATE 000000E  WLOC  000050   WRIT  000000

0000 WARNING OR ERROR FLAGS
DAP-16 MOD 2    REV. B    10-20-70



  53  BDAC        25
  51  BUF         14C     53
  55  C           54
  41  LFCR        23J     37J     49J
  13  MOV         18J
      OUT          6      29J
      OUTP         3
  29  PRNT        36J
  52  PTR         11C     13      15C     26C     30      34
      UPDATE       7      33J
  54  WLOC        10
   9  WRIT         3      39J
      ='212       46
      ='215       43
      =-4         12
      =-8         28


  16  SYMBOLS
  27  REFERS
  56  RECORDS

O16-XREF 05 OCT 70
```

Figure 3-9. Example, Output Routine (Part 2 of 2)

SECTION IV
USE OF FORTRAN PROGRAMS

FORTRAN and DAP-16 programs may be freely intermixed in a memory load and can communicate with each other through either COMMON, the argument transfer program F$AT, or argument transfer routines generated by the programmer. Entry points in a DAP-16 subroutine are declared using the ENT and SUBR pseudo-operation and in FORTRAN by the SUBROUTINE statement. The linkages are established by the DAP pseudo-operations EXT, XAC, and CALL, and by the FORTRAN statement CALL. Control is returned to the calling program by an assembly JMP* or a FORTRAN statement RETURN.

COMMON

Subroutines may transfer variables through COMMON without explicitly naming the variables in a subroutine call. Because FORTRAN COMMON and DAP-16 COMMON are handled differently, the user must deliberately locate the appropriate COMMON at the same place in core. COMMON may be relocated in the following ways.

a. During execution of TABLESIZ (that is, at the first execution of a DAP-16 Mod 2 Assembler System). This option is not possible with the conventional DAP-16 or FORTRAN.
b. During a DAP-16 Mod 2 assembly, using SETC.
c. During any assembly or FORTRAN compilation, by establishing blocks of dummy variables to move the effective COMMON location.
d. When loading, FORTRAN COMMON may be displaced by the operator.

The location of COMMON is further complicated by the Disc and Drum Operating Systems (DOPs). When using this method of communication the exact location of both FORTRAN and DAP-16 COMMON must be known for the local installation.

ARGUMENT TRANSFER SUBROUTINE F$AT

The compiler inserts a call to this subroutine at the beginning of FORTRAN-coded subroutines. F$AT transfers pointers (DACs) to the variables being communicated between the calling program and the subroutine. No call to F$AT is made for subroutines that need no arguments.

Calling a Subroutine

The sequence on the following page is used to call a subroutine that transfers arguments via F$AT. The variables are listed in the same order as in a FORTRAN CALL statement. If there is only one argument, the terminal zero must be omitted:

```
 (L  )     CALL    subroutine name
 (L+1)     DAC    <first variable>
 (L+2)     DAC    <second variable>
             .
             .
 (L+n)     DAC    <nth variable>
(L+n+1)    OCT     0                    Zero must be omitted for n = 1
(L+n+2)                                 Return point
```

The DACs to the variables can be indirect pointers; F$AT tracks down the indirect links and transfers a direct pointer. Note that variables themselves are never transferred. The reason for this is that the length of the variable is not known (it could be any length, since arrays are acceptable variables).

Calling F$AT

By convention, the first action of a subroutine is to call F$AT. Therefore the location preceeding the call points to the first argument to be transferred. F$AT transfers the arguments associated with the words following the call to F$AT. Then, F$AT increments the pointer to the calling program so that it now points to the conventional return point (following the zero). For example:

```
 (L  ) <name>  DAC    **           Subroutine entry point
 (L+1)         CALL  F$AT          Must immediately follow entry
 (L+2)         DEC   <number of arguments, n>
 (L+3) <name>  DAC    **           First argument address goes here
                .
                .
(L+n+2) <name> DAC    **           nth argument address goes here
(L+n+3)                            Return point for F$AT
```

The subroutine call may include extraneous arguments following those used by the called subroutine. Although only the number of arguments specified in L+2 of the call to F$AT are transferred, the return pointer is incremented until it points to the word following the zero in the subroutine call.

DAP-16 MAIN PROGRAM WITH FORTRAN SUBROUTINE

The DAP-16 main program and FORTRAN subroutine combination may be advantageous when assembly language programs must perform arithmetic or logical calculations, input/ output operations, or when FORTRAN procedures may be used to advantage. The DAP-16 main program must generate the call itself. Figures 4-1 through 4-5 present an example of this procedure. The DAP-16 AVGCOL program in Figure 4-1 calls another DAP-16 program MESURE (not shown) which accumulates single-precision floating-point data (for example from a peripheral measuring device). These numbers are accumulated in a buffer with the external name MINP. The number of points collected in a given run is stored in a location with the external name MNUM. Each time MESURE returns to AVGCOL, AVGCOL calls a FORTRAN subroutine STDDEV which calculates the average and standard deviation. STDDEV then prints the run number, the values, the average, and the standard deviation and passes these calculated values back to AVGCOL. In this example, AVGCOL does not use the calculated values.

```
      SUBR   AVGCOL,AVGC         EXTERNAL NAME
AVGC  LDA    =1                  INITIALIZE RUN
      STA    RUN                   NUMBER
      CALL   MESURE              SUBROUTINE TO ACCUMULATE VALUES
      CALL   STDDEV              FORTRAN PROGRAM TO CALCULATE
*                                  MEAN AND STANDARD DEVIATION
      DAC    RUN                 FIRST ARGUMENT (NRUN IN FORTRAN)
      DAC*   NUM                 SECOND ARGUMENT (NPT IN FORTRAN)
      DAC*   INP                 THIRD ARGUMENT (PT IN FORTRAN)
      DAC    STD                 FOURTH ARGUMENT (DEV IN FORTRAN)
      DAC    AVG                 FIFTH ARGUMENT (AMEAN IN FORTRAN)
      OCT    0
      IRS    RUN                 INCREMENT RUN NUMBER
      JMP    AVGC+2              COLLECT NEXT BATCH OF DATA
*     .
*     .
RUN   BSZ    1                   RUN NUMBER
NUM   XAC    MNUM                POINTER TO NUMBER OF POINTS
INP   XAC    MINP                POINTER TO DATA BUFFER
STD   DEC    0.0                 REAL STANDARD DEVIATION
AVG   DEC    0.0                 REAL AVERAGE
```

Figure 4-1. Portion of DAP-16 Program Calling FORTRAN Subroutine STDDEV

Figure 4-2 presents the FORTRAN subroutine STDDEV. An expanded listing is given in Appendix A. Figure 4-3 presents a load map for AVGCOL, MESURE, and STDDEV. Figure 4-4 is a typical output from STDDEV.

```
      SUBROUTINE STDDEV (NRUN, NPT, PT, DEV, AMEAN)
      DIMENSICN PT(100)
      SX = 0
      SX2 = 0
      DO 100 I = 1,NPT
      SX2 = SX2 + (PT(I))*(PT(I))
  100 SX = SX + PT(I)
      ANPT = NPT
      DEV = SQRT(SX2/ANPT-(SX/ANPT)*(SX/ANPT))
      AMEAN = SX/ANPT
      WRITE (1,1000) NRUN, (PT(J), J = 1,NPT)
 1000 FORMAT (////12H RUN NUMBER , I5// (E11.4,4E14.4))
      WRITE (1,2000) AMEAN, DEV
 2000 FORMAT (19H ARITHMETIC MEAN = ,E14.5,
     1/22H STANDARD DEVIATION = ,E11.5)
      RETURN
      END
$0
```

Figure 4-2. FORTRAN Subroutine STDDEV

```
*LOW    01000          REAL   03306
*START  01000          L$22   03306
*HIGH   06326          H$22   03316
*NAMES  71501          N$22   03334
*COMN   37777          F$AT   03346
*BASE   00300          ARG$   03430
AVGCCL  01000          F$W1   03450
MESURE  01024          O$AP   03544
MNUM    01564          C$AC   03616
MINP    01565          O$AF   03622
STDDEV  02010          F$IC   03632
SQRTX   02306          F$AR   04155
SQRT    02306          F$CB   04333
C$12    02422          F$ER   06252
S$22    02454          F$HT   06262
A$22    02462          AC1    06320
M$22X   02704          AC2    06321
M$22    02704          AC3    06322
D$22X   03065          AC4    06323
D$22    03065          AC5    06324
SNGL    03306                 37777
```

Figure 4-3. Loader Map for AVGCOL, MEASURE, and STDDEV

```
RUN NUMBER        7

0.7680E-01     0.7520E-01     0.7270E-01     0.7100E-01     0.7570E-01
0.7350E-01     0.7510E-01     0.7320E-01     0.7010E-01     0.7270E-01
0.7610E-01     0.6970E-01     0.7410E-01     0.7460E-01     0.7380E-01
0.7320E-01     0.7310E-01     0.7310E-01     0.7110E-01     0.7150E-01
0.7510E-01     0.7640E-01     0.7120E-01
ARITHMETIC MEAN =     0.73435E-01
STANDARD DEVIATION = 0.19745E-02
```

Figure 4-4. Output From STDDEV

FORTRAN MAIN PROGRAM WITH DAP-16 SUBROUTINE

The FORTRAN main program and DAP-16 subroutine combination is required when tasks which cannot be performed in FORTRAN must be done. In this case the DAP-16 program must handle the call to F$AT, or transfer the required arguments directly.

Figures 4-5 and 4-6 provide a sample of this combination. The FORTRAN main program requires input from paper tape in a special format as shown in Figure 4-7. The FORTRAN main program passes the start of message character (which may vary from application to application) to the DAP-16 subroutine. The subroutine then reads the tape. The first two words are integer values passed back through the calling parameters. The next two words are a real value also passed back through the calling parameters. The next four words are a complex value passed to the main program through COMMON. The COMMON base must be set to the same value by one of the methods mentioned above. Notice that X3 is part of COMMON in the FORTRAN program, but not involved in calling READT.

Figure 4-8 shows another version of READT that does not use F$AT but instead transfers the arguments directly.

```
      COMMON I(10,10),J1,J2,X1,X2,X3
      COMPLEX X2,X3
         .
         .
      ISTART = 129
C     129 IS OCTAL 201 (START OF MESSAGE)
      CALL READT (ISTART, J1, J2, X1)
```

Figure 4-5. FORTRAN Calling Sequence for DAP-16 Subroutine READT

```
      SUBR  READT,TAPE
      REL
TAPE  DAC   **                      ENTRY POINT (USED AS POINTER BY F$AT)
      CALL  F$AT                    CALL ARGUMENT TRANSFER SUBROUTINE
      DEC   4                       FOUR ARGUMENTS TO BE TRANSFERRED
CHAR  DAC   **                      POINTER TO CHAR GOES HERE
P1    DAC   **                      POINTER TO P1 GOES HERE
P2    DAC   **                      POINTER TO P2 GOES HERE
P3    DAC   **                      POINTER TO P3 GOES HERE
      LDA   CMPT                    PICK UP COMMON POINTER
      STA   CMN1                    STORE IN TEMPORARY LOCATION
      OCP   '0001                   TURN ON PAPER TAPE READER
      LDA*  CHAR                    PICK UP START OF MESSAGE CHARACTER
      STA   SOM                     SAVE IT
      INA   '1001                   CLEAR A AND INPUT CHARACTER
      JMP   *-1                     DELAY UNTIL READY
      ERA   SOM                     IS IT START-OF-MESSAGE CHARACTER?
      SZE                           IGNORE IF IT IS NOT
      JMP   *-4                     NOPE, TRY ANOTHER ONE
      JST   FORM                    FORM A WORD FROM THE NEXT TWO CHARACTERS
      STA*  P1                      THIS IS P1; RETURN IT TO CALLING PROGRAM
      JST   FORM                    FORM ANOTHER WORD
      STA*  P2                      THIS IS P2; RETURN IT
      JST   FORM                    FORM ANOTHER WORD
      STA*  P3                      THIS IS THE FIRST WORD OF P3
      IRS   P3                      POINT TO THE SECOND WORD
      JST   FORM                    FORM THE SECOND WORD OF P3
      STA*  P3                      STORE IT
*           NOW GET THE FOUR WORDS OF THE COMPLEX VARIABLE
      LDX   =-4                     FOUR WORDS TO BE FORMED
LOOP  JST   FORM                    FORM A WORD
      STA*  CMN1                    STORE IN COMMON LOCATION
      IRS   CMN1                    POINT TO NEXT COMMON LOCATION
      IRS   0                       UPDATE INDEX
      JMP   LOOP                    LOOP UNTIL 4 WORDS TAKEN CARE OF
      OCP   '0101                   NOW TURN OFF THE TAPE READER
      JMP*  TAPE                      AND RETURN TO CALLING PROGRAM
*
FORM  DAC   **                      ENTRY POINT
      INA   '1001                   CLEAR A AND INPUT CHARACTER
      JMP   *-1                     DELAY UNTIL READY
      ICR                           INTERCHANGE AND CLEAR RIGHT HALF
      INA   '0001                   INPUT CHARACTER
      JMP   *-1                     INPUT SECOND CHARACTER
      JMP*  FORM                    RETURN WITH WORD IN A REGISTER
CN    COMN  8
CMPT  DAC   CN                      POINTER TO FIRST WORD OF COMPLEX BLOCK
CMN1  BSZ   1                       TEMPORARY LOCATION FOR POINTER
SOM   OCT   0                       STORAGE FOR START OF MESSAGE CHARACTER
      END
```

Figure 4-6. DAP-16 Subroutine READT

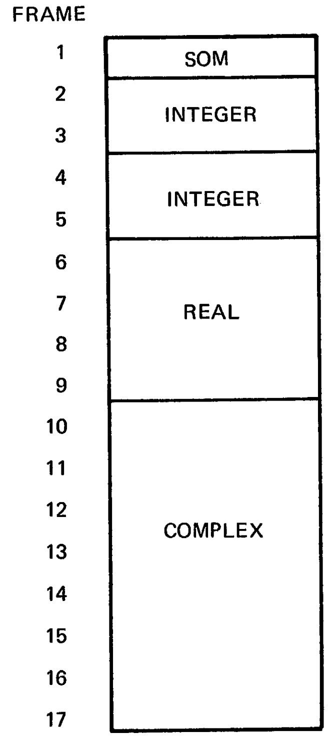


Figure 4-7. Paper Tape Input Format (for Figures 3-4 and 3-5)

```
     SUBR   READT,TAPE
     REL
TAPE DAC    **              ENTRY POINT (USED AS POINTER BY F$AT)
     LDA*   TAPE            PICK UP FIRST ARGUMENT (CHAR)
     JST    IND             RUN DOWN INDIRECT LINKS
     STA    CHAR            POINTER TO SOM CHARACTER
     IRS    TAPE            POINT TO NEXT ARGUMENT (P1)
     LDA*   TAPE            PICK IT UP
     JST    IND             RUN DOWN INDIRECT LINKS
     STA    P1              STORE IT
     IRS    TAPE            POINT TO NEXT ARGUMENT (P2)
     LDA*   TAPE            PICK IT UP
     JST    IND             RUN DOWN INDIRECT LINKS
     STA    P2              STORE IT
     IRS    TAPE            POINT TO NEXT ARGUMENT (P3)
     LDA*   TAPE            PICK IT UP
     JST    IND             RUN DOWN INDIRECT LINKS
     STA    P3              STORE IT
     IRS    TAPE            POINT TO NEXT ARGUMENT OR ZERO
     LDA*   TAPE            PICK IT UP
     SZE                    DONE IF IT IS ZERO
     JMP    *-3             KEEP INCREMENTING UNTIL ZERO REACHED
     IRS    TAPE            POINT TO RETURN POINT
     LDA    CMPT            PICK UP COMMON POINTER
     STA    CMN1            STORE IN TEMPORARY LOCATION
     OCP    '0001           TURN ON PAPER TAPE READER
     LDA*   CHAR            PICK UP START OF MESSAGE CHARACTER
     STA    SOM             SAVE IT
     INA    '1001           CLEAR A AND INPUT CHARACTER
```

Figure 4-8. DAP-16 Subroutine READT, Transferring Arguments Without Calling F$AT

```
       JMP    *-1                     DELAY UNTIL READY
       ERA    SOM                     IS IT START-OF-MESSAGE CHARACTER?
       SZE                            IGNORE IF IT IS NOT
       JMP    *-4                     NOPE, TRY ANOTHER ONE
       JST    FORM                    FORM A WORD FROM THE NEXT TWO CHARACTERS
       STA*   P1                      THIS IS P1; RETURN IT TO CALLING PROGRAM
       JST    FORM                    FORM ANOTHER WORD
       STA*   P2                      THIS IS P2; RETURN IT
       JST    FORM                    FORM ANOTHER WORD
       STA*   P3                      THIS IS THE FIRST WORD OF P3
       IRS    P3                      POINT TO THE SECOND WORD
       JST    FORM                    FORM THE SECOND WORD OF P3
       STA*   P3                      STORE IT
*             NOW GET THE FOUR WORDS OF THE COMPLEX VARIABLE
       LDX    =-4                     FOUR WORDS TO BE FORMED
LOOP   JST    FORM                    FORM A WORD
       STA*   CMN1                    STORE IN COMMON LOCATION
       IRS    CMN1                    POINT TO NEXT COMMON LOCATION
       IRS    0                       UPDATE INDEX
       JMP    LOOP                    LOOP UNTIL 4 WORDS TAKEN CARE OF
       OCP    '0101                   NOW TURN OFF THE TAPE READER
       JMP*   TAPE                      AND RETURN TO CALLING PROGRAM
*
FORM   DAC    **                      ENTRY POINT
       INA    '1001                   CLEAR A AND INPUT CHARACTER
       JMP    *-1                     DELAY UNTIL READY
       ICR                            INTERCHANGE AND CLEAR RIGHT HALF
       INA    '0001                   INPUT CHARACTER
       JMP    *-1                     INPUT SECOND CHARACTER
       JMP*   FORM                    RETURN WITH WORD IN A REGISTER
*
IND    DAC    **                      ENTRY POINT FOR REMOVING ALL
*                                       INDIRECT LINKS
       SMI                            INDIRECT POINTER?
       JMP*   IND                     NO--RETURN
       SSP                            YES--REMOVE INDIRECT FLAG AND TRY AGAIN
       STA    TEMP                    SAVE IT
       LDA*   TEMP                    PICK UP WHAT IT POINTS TO
       JMP    IND+1                   AND CHECK IT FOR INDIRECT
*
CN     COMN   8
CMPT   DAC    CN                      POINTER TO FIRST WORD OF COMPLEX BLOCK
CMN1   BSZ    1                       TEMPORARY LOCATION FOR POINTER
SOM    OCT    0                       STORAGE FOR START OF MESSAGE CHARACTER
TEMP   BSZ    1                       STORAGE USED FOR RUNNING DOWN INDIRECTS
CHAR   DAC    **                      POINTER TO CHAR GOES HERE
P1     DAC    **                      POINTER TO P1 GOES HERE
P2     DAC    **                      POINTER TO P2 GOES HERE
P3     DAC    **                      POINTER TO P3 GOES HERE
       END
```

Figure 4-8. DAP-16 Subroutine READT, Transferring Arguments Without Calling F$AT (Cont.)

SECTION V
PERFORMING AN ASSEMBLY (DAP-16 MOD 2)

Initially, the Assembler along with the proper IOS (Input/Output Supervisor) subroutines must be loaded. Normally a system is generated rather infrequently and a reloadable core dump (binary record) made for general use. The core dump is loaded from paper tape, cards, disc, etc. whenever an assembly is to be performed.

The source (tape, deck, or disc file) is loaded on the proper input device and the bits of the A-Register are set to indicate the mode of assembly and the devices being used for input and output (see Figure 5-1). Some Input/Output Supervisors also require a B-Register setting. Set the P-Register to '400 and push the START button (see Table 5-1 for other starting addresses).

At the end of the first pass the computer will halt. If a two-pass assembly is being performed, press the START button when the source has been repositioned. When the source is on magnetic tape or disc, automatic positioning can be specified and the computer in this case does not halt.

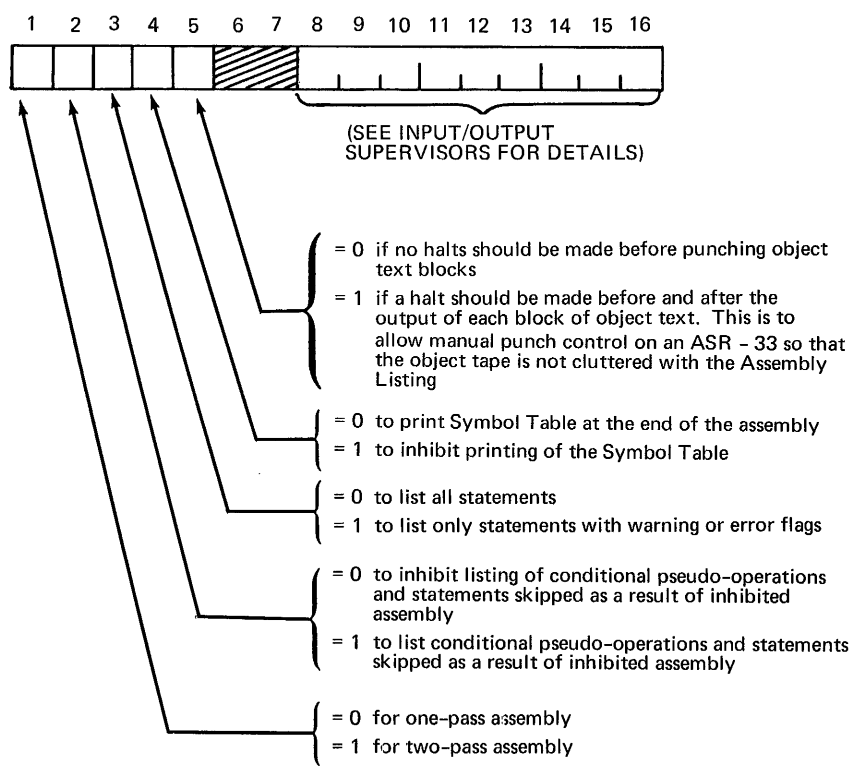


Figure 5-1. A-Register Settings for Assembler Initialization

TABLE 5-1. ASSEMBLER STARTING ADDRESSES

'400	Start normal assembly
'401	Continue assembly (used after halts for read errors etc.)
'402	Start subroutine assembly (no end-of-file will be placed in the object text)
'403	Terminate assembly (place end-of-file in the object text)
'404	Restart second pass for additional listing or additional object text (A-Register bit changes accepted).

ESTIMATION OF SYMBOL TABLE SIZE

The Symbol Table occupies the core area above the Assembler System. If this table overflows, the assembly cannot be performed. Each entry occupies three words, and as a general rule one entry is produced for every four or five lines of source text (2/3 words in the Symbol Table per line of text). The programmer may minimize the number of entries by use of displacements from symbolic values or the asterisk element.

ASSEMBLER SUPPORT PROGRAMS

The following programs must be linked to the Assembler for proper operation. The Input/Output Supervisors are described following discussion of these programs.

O16-DECS, O16-DECL

These programs, O16-DECS and O16-DECL, provide the ASCII-to-binary conversion capability of the Assembler. O16-DECS must be used for systems with up to 4K memory locations. However O16-DECS does not provide floating-point or double-precision conversions. O16-DECL may be used with any system having more than 4K memory locations. The full range of conversions as described under DEC, DBP, OCT, and HEX is available with O16-DECL.

SYMLIST, Symbol Table Printer

The program SYMLIST performs an alphabetic sort of all entries in the Symbol Table and prints out these entries, four per line, following the assembly. The last value printed is the one for symbols established by SET. Following the value of each symbol is a blank if the symbol is relocatable, an A if it is absolute, and an E if it is external (external symbols always equal zero). The Symbol Table may be suppressed by entering a 1 in bit 4 of the A-Register when starting the Assembler. Figures 3-6 and 3-7 show two assemblies with Symbol Tables.

TABLESIZ

The last Assembler support program loaded must be TABLESIZ. This program is called at the start of the first assembly by the Input/Output Supervisor. Functionally, TABLESIZ derives the top of memory and returns this location and the COMMON base ('177 locations below the top of memory) to the supervisor. The symbol table overlays TABLESIZ, and it is not called for subsequent assemblies. If Sense Switch 1 is set during execution of TABLESIZ, the computer will halt with the highest memory location in the A-register. This location may then be changed manually. The computer will then halt again with the COMMON base displayed for the operator to change if desired.

INPUT/OUTPUT SUPERVISORS

DAP-16 input/output supervisors are designed to operate with standard Honeywell drivers (using their calling sequences and their expected results). These drivers are described in the Programmers Reference Manuals for the specific peripheral devices.

One IOS program and the appropriate driver programs must be linked within an assembler system along with the programs listed in the previous section. TABLESIZ must be the last program (highest core address) in the system following the drivers.

NOTE

This section generally indicates the features available to the programmer in the assembler system as generated from standard software. An installation that performs a large number of assemblies will normally find it worthwhile to tailor an IOS to the installation standard. This tailoring may include card-to-tape or card-to-disc transfer on the first pass, source blocking, simultaneous peripheral transfer and computation, and operating system interfaces. Some of these features are available on a standard item basis.

Dedicated IOS Programs

Computer systems with 4K memory locations must use one of the dedicated input/output supervisors. Each of these IOS programs uses a fixed set of peripheral devices. Therefore, no bits need to be set for device selection when starting the assembly. Table 5-2 lists the programs and the devices to which they are dedicated.

TABLE 5-2. DEDICATED INPUT/OUTPUT SUPERVISORS

Name	Symbolic Input	Object Text	Listing
IOS-OAAA	ASR	ASR	ASR
IOS-ORAA	High-Speed Paper Tape Reader	ASR	ASR
IOS-ORPA	High-Speed Paper Tape Reader	High-Speed Paper Tape Punch	ASR

With any of these dedicated supervisors Sense Switches 3 and 4 respectively may be used to suppress the object text and listing. If Sense Switch 3 is set during the assembly, no object will be produced. If Sense Switch 4 is set, no listing will be produced.

IOS-016D

IOS-016D is the supervisory program that permits a choice of input and output devices. This program must be used only on computer systems with 8K or more memory locations. Table 5-3 lists the options available for input and output with this supervisor. The octal numbers are entered in the A-register before starting the assembly. Table 5-4 lists the B-register settings used when magnetic tape is specified. These settings define the file more fully for the supervisor.

When IOS-016D is used with a disc or drum the appropriate DOP (Disc Operating Program) must be present. There is a DOP for each standard disc and drum in the Honeywell product line. DOP asks the operator which files (by name) are to be attached as pseudo-devices for the current assembly. Access to these files is handled by DOP.

TABLE 5-3. DEVICE SELECTION WITH IOS-016D

	IOS-016D
Symbolic Input Bits 8-10	
0	Undefined
1	ASR
2	High-Speed Paper Tape Reader
3	Card Reader
4	Magnetic Tape
5	Disc or Drum
6-7	Undefined
Object Text Outputs Bits 11-13	
0	No object text
1	ASR
2	High-Speed Paper Tape Punch
3	Card Punch
4	Magnetic Tape
5	Disc or Drum
6-7	Undefined
Listing Output Bits 14-16	
0	No listing
1	ASR
2	High-Speed Paper Tape Punch
3	Line Printer
4	Magnetic Tape
5	Disc or Drum
6-7	Undefined

TABLE 5-4. B-REGISTER SETTINGS FOR MAGNETIC TAPE INPUT/OUTPUT

Bits 1-2	Logical Tape Unit Number for source. Default is logical unit 1.
Bits 3-4	Logical Tape Unit Number for object. Default is logical unit 2.
Bits 5-6	Logical Tape Unit Number for listing. Default is logical unit 3.
Bit 7	=0 Normal operation. =1 Continuous mode operation. The computer will immediately halt. At this time the operator should enter the number of files to be processed into the B-Register. Zero means all files until a double EOF (blank file) is encountered. The computer will not stop again until the indicated number of assemblies have been performed. Operative only with magnetic tape input.
Bits 9-16	How many files to skip before starting the assembly.

SECTION VI
PERFORMING AN ASSEMBLY (DAP-16)

Initially, the Assembler along with the proper IOS (Input/Output Supervisor) subroutines must be loaded. Normally a system is generated rather infrequently and a reloadable core dump (binary record) made for general use. The core dump is loaded from paper tape, cards, disc, etc., whenever an assembly is to be performed.

The source (tape, deck, or disc file) is loaded on the proper input device and the bits of the A-Register are set to indicate the mode of assembly and the devices being used for input and output (see Figure 6-1). Some Input/Output Supervisors also require a B-Register setting. Set the P-Register to '400 and push the START button (see Table 6-1 for other starting addresses).

At the end of the first pass the computer will halt. If a two-pass assembly is being performed, press the START button when the source has been repositioned. When the source is on magnetic tape or disc, automatic positioning can be specified and the computer in this case does not halt.

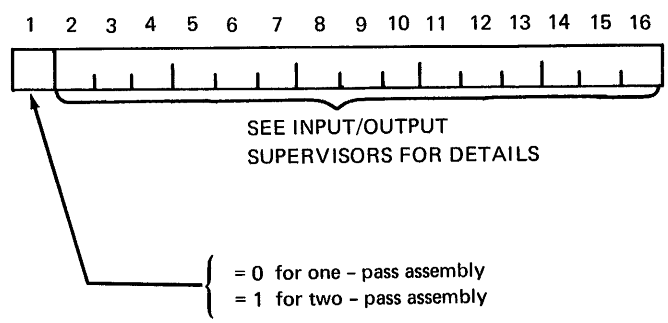


Figure 6-1. A-Register Settings for Assembler Initialization

TABLE 6-1. ASSEMBLER STARTING ADDRESSES

'400	Start normal assemble
'401	Continue assembly (used after halts for read errors etc.)
'402	Start subroutine assembly (no end-of-file will be placed in the object text)
'403	Terminate assembly (place end-of-file in the object text)
'404	Restart second pass for additional listing or additional object text (A-Register bit changes accepted).

ESTIMATION OF SYMBOL TABLE SIZE

The Symbol Table occupies the core area above the Assembler System. If this table overflows, the assembly cannot be performed. Each entry occupies three words, and as a general rule one entry is produced for every four or five lines of source text (2/3 words in the symbol table per line of text). The programmer may minimize the number of entries by use of displacements from symbolic values or the asterisk element.

ASSEMBLER SUPPORT PROGRAMS

The following programs must be linked to the Assembler for proper operation. The Input/Output Supervisors are described following the discussion of these programs.

DECCS, DECCL

DECCS and DECCL provide the ASCII-to-binary conversion capability of the Assembler. DECCS must be used for systems with up to 4K memory locations. DECCS does not provide floating-point or double-precision conversions. DECCL may be used with any system having more than 4K memory locations. The full range of conversions as described under DEC, DBP, and OCT is available with DECCL.

MEMSIZ, SETSIZ

One of these programs (MEMSIZ or SETSIZ) must be the last assembler support program loaded (MEMSIZ for 4K systems; and SETSIZ for systems with more than 4K memory locations). MEMSIZ or SETSIZ is called at the start of the first assembly by the Input/Output Supervisor. Functionally MEMSIZ or SETSIZ derives the top of memory and returns this location and the COMMON base ('177 locations below the top of memory) to the Supervisor. The Symbol Table overlays MEMSIZ or SETSIZ and the pertinent program is not called for subsequent executions.

INPUT/OUTPUT SUPERVISORS

DAP-16 Input/Output Supervisors are designed to operate with the standard Honeywell drivers (using their calling sequences and their expected results). These drivers are described in the Programmers Reference Manuals for specific peripheral devices.

One IOS program and the appropriate driver programs must be linked within an Assembler system along with the programs listed in the previous section. TABLESIZ must be the last program (highest core address) in the system, following the drivers.

NOTE

This section generally indicates the features available to the programmer in the Assembler System as generated from standard software. An installation which performs a large number of assemblies will normally find it worth-while to tailor an IOS to the installation standard. This tailoring may include card-to-tape or card-to-disc transfer on the first pass, source blocking, simultaneous peripheral transfer and computation, and operating system interfaces. Some of these features are available on a standard item basis.

Dedicated IOS Programs

Computer systems with up to 4K memory locations must use one of these dedicated input/output supervisors. Each of these IOS programs uses a fixed set of peripheral devices. Therefore, no bits need to be set for device selection when starting the assembly. Table 6-2 lists the programs and the devices to which they are dedicated.

TABLE 6-2. DEDICATED INPUT/OUTPUT SUPERVISORS

Name	Symbolic Input	Object Text	Listing
IOS-5AAA	ASR	ASR	ASR
IOS-5RAA	High-Speed Paper Tape Reader	ASR	ASR
IOS-5CAA	Card Reader	ASR	ASR
IOS-5RPA	High-Speed Paper Tape Reader	High-Speed Paper Tape Punch	ASR
IOS-5CPA	Card Reader	High-Speed Paper Tape Punch	ASR

IOS-516X, IOS-516D

IOS-516X and IOS-516D are supervisory programs that permit a choice of input and output devices. These programs must be used only on computer systems with 8K or more memory locations. Table 6-3 lists the options available for input and output with these supervisors. The indicated bits are filled in the A-register before starting the assembly. Table 6-4 lists the B-register settings used when magnetic tape is specified. These settings define the file more fully for the supervisor.

When IOS-516D is used, the appropriate DOP (Disc Operating Program) must be present. There is a DOP for each standard disc and drum in the Honeywell product line. DOP asks the operator which files (by name) are to be attached as pseudo-devices for the current assembly. Access to these files is handled by DOP.

TABLE 6-3. DEVICE SELECTION WITH IOS-516X AND IOS-516D

	IOS-516X	IOS-516D
Symbolic Input		
Bit 2	Teletypewriter	Teletypewriter
Bit 3	High-Speed Paper Tape Reader	High-Speed Paper Tape Reader
Bit 4	Card Reader	Card Reader
Bit 5	Magnetic Tape	Magnetic Tape
Bit 6	Teletypewriter with program halts for manual action	Teletypewriter with program halts for manual action
Bits 2-6 all = 0	Undefined	Disc
Object Text Output		
Bit 7	Teletypewriter	Teletypewriter
Bit 8	High-Speed Paper Tape Punch	High-Speed Paper Tape Punch
Bit 9	Undefined	Undefined
Bit 10	Magnetic Tape	Magnetic Tape
Bit 11	No object text	No object text
Bits 7-11 all = 0	Undefined	Disc
Listing Output		
Bit 12	Teletypewriter	Teletypewriter
Bit 13	High-Speed Paper Tape Punch	High-Speed Paper Tape Punch
Bit 14	Magnetic Tape	Magnetic Tape
Bit 15	Line Printer	Line Printer
Bit 16	No listing	No listing
Bits 12-16 all = 0	Undefined	Disc

Table 6-4. B-Register Settings for Magnetic Tape Input/Output

Bits 1-2	Logical Tape Unit Number for source. Default is logical unit 1.
Bits 3-4	Logical Tape Unit Number for object. Default is logical unit 2.
Bits 5-6	Logical Tape Unit Number for listing. Default is logical unit 3.
Bit 7	=0 Normal operation. =1 Continuous mode operation. The computer will immediately halt. At this time the operator should enter the number of files to be processed into the B-Register. Zero means all files until a double EOF (blank file) is encountered. The computer will not stop again until the indicated number of assemblies have been performed. Operative only with magnetic tape input.
Bits 9-16	How many files to skip before starting the assembly.

SECTION VII
GENERATING AN ASSEMBLER SYSTEM

This section describes the process of generating a DAP-16 Mod 2 Assembly System from paper tape objects. Most systems (notably conventional DAP-16) are generated analogously. With conventional DAP-16, however, care must be taken to avoid filling the base sector beyond '377, which would overwrite the assembler. To avoid filling that portion of the base sector, as many programs as possible should be loaded starting on a sector boundary.

The system described in this section was generated on a computer with 12K memory locations. To generate this system on an 8K computer, at least one driver package must be left out. O16-DECL is used for decimal conversion, and the input/output supervisor used is IOS-O16D.

LOADING LOADER

LDR-APM must be loaded into high sectors of memory before starting. A self-loading form is available which loads in sectors 4 through 7. This program may be used to load the loader object starting at any even sector boundary.

LOADING ASSEMBLER

The starting location of the cross-sector references must be set as low as possible in order to provide enough room. The lowest possible address is '40. In this example, '60 was used. This address should be entered in the B-register before loading the assembler. If no B-register entry is made, '100 is assumed. If DMC, Real-Time Clock, Memory Lockout, Standard Interrupt, or Priority Interrupt/Memory Increment are used, their needs must be taken into account when making this setting.

Enter relative location '3000 into the P-register. If the loader, for example, starts at the beginning of sector '24, '27000 is relative location '3000. Mount the assembler object text on the proper input device and press START. The computer will halt to receive the input device selection in the A-register. After the proper code is entered, press START again and the assembler will load.

Generating Map

Start the loader at relative location '3002. If the computer is allowed to print the entire map, MR will be printed and the computer will halt. Usually, the first six lines of the map (especially *HIGH and *BASE) are all that are pertinent. The remaining lines tell what additional routines are needed. The computer may be halted during a map with the MA/SI/RUN Switch and the map printer reinitialized by again starting at relative location '3002. A map (or the first six lines of a map) taken after almost every load step is helpful.

After the assembler has been loaded, *HIGH should be in sector 5 and *BASE should be not far above the value initialized in the B-register. The next routine loaded will load at *HIGH and start its cross-sector links at *BASE.

LOADING IOS-O16D

To conserve cross-sector references, the selected IOS should start at the beginning of sector 6 rather than at the current value of *HIGH. Set '6000 in the A-register, mount the IOS object, and start the computer at relative location '3003. From then on, the input device for the loader does not need to be reselected.

LOADING O16-DECL

This routine (or O16-DECS) need not start on a sector boundary. Therefore, it may be loaded simply by starting the computer at relative location '3003.

LOADING SYMLIST

This routine (if desired) may also be started at the current value of *HIGH. Start the computer at relative location '3003.

LOADING IOS DRIVERS

The following IOS driver packages can be loaded: ASR, Paper Tape Reader and Punch, Card Reader, Card Punch, Line Printer, and Magnetic Tape. Each of these packages includes several routines, some of which are not used by the assembler system. For some input libraries, START must be pressed for reading each routine, whether or not it is actually loaded. Other libraries do not have stop codes other than the physical end of tape, which is a real convenience.

When using magnetic tape, routine M$UNIT must be configured to the installation standard. See the appropriate magnetic tape programmers reference manual for details.

Maps should be taken at this time to ensure that there is still room in the base sector. If the number of remaining locations is critical, specific routines should be loaded on sector boundaries. To do this, set the loading location in the A-register and start the loader at relative location '3003.

The calls to any omitted packages should be satisfied by a dummy, which is an object text with entry points for each external name called. The safest way to handle these entries is to point each one to a halt or generate an error message. Dummy texts (e.g., DUMY-X16) are available from Honeywell upon request.

Figure 7-1 shows the source of a dummy that satisfies calls to the card punch routines. Normally one dummy with a lengthy list of SUBR statements is used to avoid wasting operator time and core space.

```
       SUBR   C$CB,DUMY
       SUBR   O$CS,DUMY
       REL                         RELOCATABLE SUBROUTINE
DUMY   DAC    **                   ALL CALLS TO CARD PUNCH COME HERE
       HLT                         HALT TO ALERT OPERATOR
       JMP    *-1                  DO NOT ALLOW RESTART FROM HERE
       END
```

Figure 7-1. Dummy Example

LOADING TABLESIZ

After all other routines and the dummy have been loaded, the object for TABLESIZ should be loaded. This must be the last (highest in memory) routine loaded.

PRODUCING SELF-LOADING CORE IMAGE

Figure 7-2 shows the result in core for this example. This result may be preserved and reused if a self-loading (binary) core image text is made. For disc or drum systems, DOP can store the binary image on the disc or drum. A paper tape image may be made using PAL-AP. An 8K version of PAL-AP may be used as shown in Figure B-2.[a] PAL-C is the proper program for producing a core image in binary cards. Either of these programs must load on a sector boundary. Both are started at their relative location '000.

---

[a]An 8K version of PAL-AP may be generated by the following steps. Use any Loader to load the object text of PAL-AP into sector 7 (the Loader is no longer needed and can be overwritten). Change the contents of location '7575 (for Rev. E of PAL-AP) from '7600 to '17600. Execute PAL-AP starting at '7000 to dump the other version from '17000 to '17577. This dump is a version of PAL-AP that will load into, and execute properly from the uppermost sector of an 8K memory. It may be used to dump core from '70 to 16777.

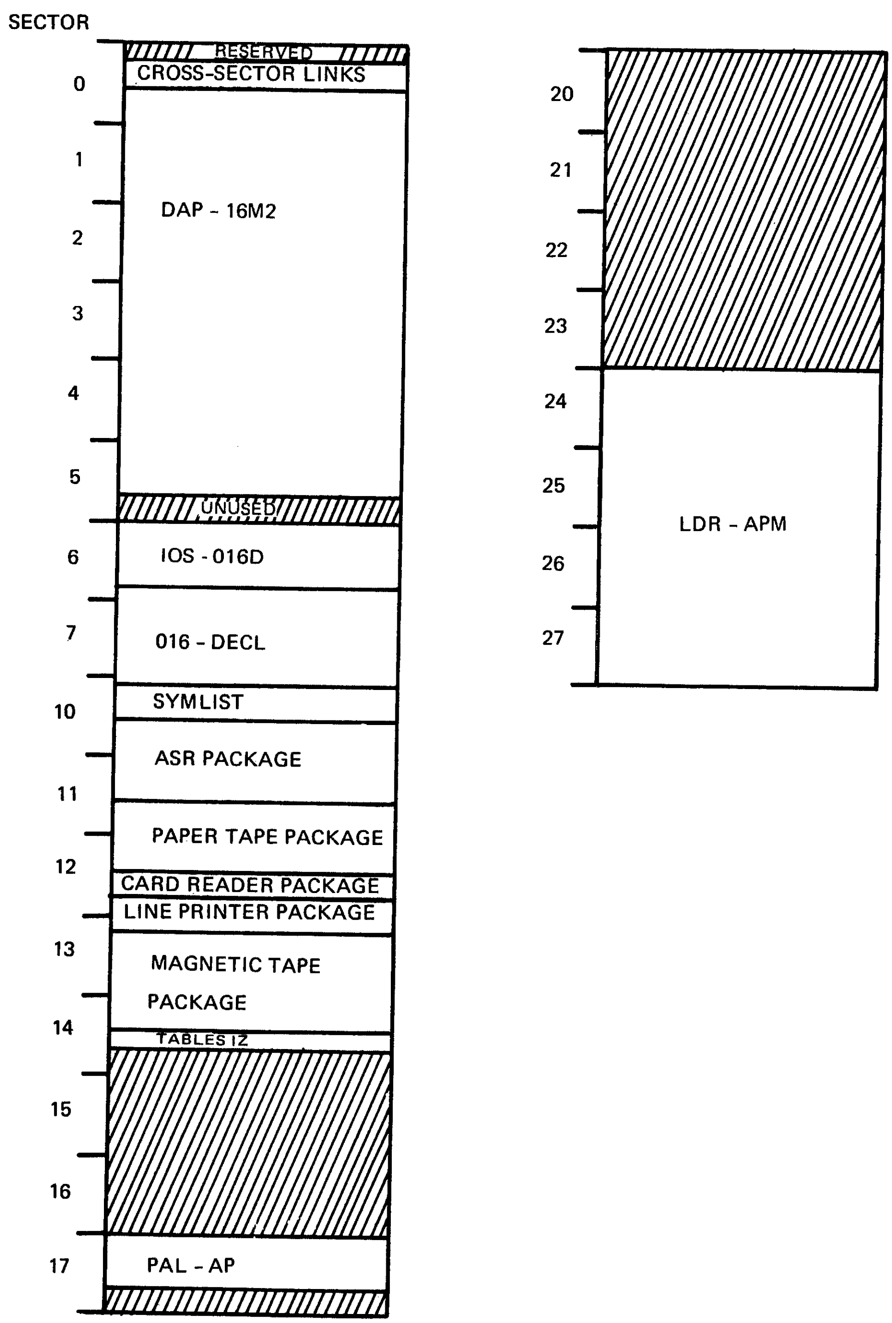


Figure 7-2. Core Map, After Generating Assembler System

```
 2000 FORMAT (19H ARITHMETIC MEAN = ,E14.5,
             STG     ←2000
000213       JMP     000000
000214       OCT     124261
000215       OCT     134710
000216       OCT     120301
000217       OCT     151311
000220       OCT     152310
000221       OCT     146705
000222       OCT     152311
000223       OCT     141640
000224       OCT     146705
000225       OCT     140716
000226       OCT     120275
000227       OCT     120254
000230       OCT     142661
000231       OCT     132256
000232       OCT     132654

     1/22H STANDARD DEVIATION = ,E11.5)
000233       OCT     127662
000234       OCT     131310
000235       OCT     120323
000236       OCT     152301
000237       OCT     147304
000240       OCT     140722
000241       OCT     142240
000242       OCT     142305
000243       OCT     153311
000244       OCT     140724
000245       OCT     144717
000246       OCT     147240
000247       OCT     136640
000250       OCT     126305
000251       CCT     130661
000252       OCT     127265
000253       OCT     124640
             STG     000213

      RETURN
000254       JMP*    000000

      END
             STG     ='000001
000255       OCT     000001
000003       DAC     NRUN
000004       DAC     NPT
000005       DAC     PT
000006       DAC     DEV
000007       DAC     AMEAN
             STG     SX
000256       OCT     120240
000257       OCT     120240
             STG     ='000000
000260       OCT     000000
             STG     SX2
000261       OCT     120240
000262       OCT     131240
000042       DAC     ←100
             STG     I
000263       OCT     004640
             STG     T$1000
000264       OCT     012244
             STG     ANPT
000265       OCT     120240
000266       OCT     150324
000000       DAC     SQRT
             STG     T$2000
000267       OCT     130260
000270       OCT     131260
             STG     T$2001
000271       OCT     130261
000272       OCT     131260
             STG     T$2002
000273       OCT     130262
000274       OCT     131260
000155       DAC     ←1000
             STG     J
000275       OCT     005240
000213       DAC     ←2000

$0
```

Figure A-1. Expanded Listing of STDDEV (Cont.)

APPENDIX A
EXPANDED STDDEV LISTING

```
      SUBRCUTINE STDDEV (NRUN, NPT, PT, DEV, AMEAN)
000000     CCT    000000
000001     CALL   F$AT
000002     CCT    000005
000003     CCT    000000
000004     CCT    000000
000005     CCT    000000
000006     CCT    000000
000007     CCT    000000

      DIMENSICN PT(100)

      SX = 0
000010     JMP    000000
           STG    000010
000011     LDA    ='000000
000012     CALL   C$12
000013     CALL   H$22
000014     DAC    SX

      SX2 = 0
000015     LDA    ='000000
000016     CALL   C$12
000017     CALL   H$22
000020     DAC    SX2

      DC 100 I = 1,NPT
000021     LDA    ='000001
000022     STA    I

      SX2 = SX2 + (PT(I))*(PT(I))
000023     LDA    I
000024     ALS1   000000
000025     ADD    PT
000026     ADD    000030
000027     JMP    000031
000030     CCT    177776
000031     STA    T$1000
000032     CALL   L$22
000033     DAC*   T$1000
000034     CALL   M$22
000035     DAC*   T$1000
000036     CALL   A$22
000037     DAC    SX2
000040     CALL   H$22
000041     DAC    SX2
```

Figure A-1. Expanded Listing of STDDEV

```
 100 SX = SX + PT(I)
000042     LDA     I
000043     ALS1    000000
000044     ADD     PT
000045     ADD     000047
000046     JMP     000050
000047     CCT     177776
000050     STA     T$1000
000051     CALL    L$22
000052     DAC*    T$1000
000053     CALL    A$22
000054     DAC     SX
000055     CALL    H$22
000056     DAC     SX
000057     LDA     I
000060     ADD     ='000001
000061     CAS*    NPT
000062     JMP     000065
000063     JMP     000022
000064     JMP     000022

      ANPT = NPT
000065     LDA*    NPT
000066     CALL    C$12
000067     CALL    H$22
000070     DAC     ANPT

      DEV = SQRT(SX2/ANPT-(SX/ANPT)*(SX/ANPT))
000071     CALL    L$22
000072     DAC     SX
000073     CALL    D$22
000074     DAC     ANPT
000075     CALL    H$22
000076     DAC     T$2000
000077     CALL    M$22
000100     DAC     T$2000
000101     CALL    H$22
000102     DAC     T$2001
000103     CALL    L$22
000104     DAC     SX2
000105     CALL    D$22
000106     DAC     ANPT
000107     CALL    S$22
000110     DAC     T$2001
000111     CALL    H$22
000112     DAC     T$2002
000113     CALL    SQRT
000114     DAC     T$2002
000115     CALL    H$22
000116     DAC*    DEV

      AMEAN = SX/ANPT
000117     CALL    L$22
000120     DAC     SX
000121     CALL    D$22
000122     DAC     ANPT
000123     CALL    H$22
000124     DAC*    AMEAN
```

Figure A-1. Expanded Listing of STDDEV (Cont.)

```
       WRITE (1,1000) NRUN, (PT(J), J = 1,NPT)
000125      CALL   F$W1
000126      DAC    ←1000
000127      CALL   F$AR
000130      OCT    000001
000131      DAC*   NRUN
000132      LDA    ='000001
000133      STA    J
000134      LDA    J
000135      ALS1   000000
000136      ADD    PT
000137      ADD    000141
000140      JMP    000142
000141      OCT    177776
000142      STA    T$1000
000143      CALL   F$AR
000144      OCT    000002
000145      DAC*   T$1000
000146      LDA    J
000147      ADD    ='000001
000150      CAS*   NPT
000151      JMP    000154
000152      JMP    000133
000153      JMP    000133
000154      CALL   F$CB

  1000 FORMAT (////12H RUN NUMBER , I5// (E11.4,4E14.4))
            STG    ←1000
000155      JMP    000000
000156      OCT    124257
000157      OCT    127657
000160      OCT    127661
000161      OCT    131310
000162      OCT    120322
000163      OCT    152716
000164      OCT    120316
000165      OCT    152715
000166      OCT    141305
000167      OCT    151240
000170      OCT    126211
000171      OCT    132657
000172      OCT    127650
000173      OCT    142661
000174      OCT    130656
000175      OCT    132254
000176      OCT    132305
000177      OCT    130664
000200      OCT    127264
000201      OCT    124651
            STG    000155

       WRITE (1,2000) AMEAN, DEV
000202      CALL   F$W1
000203      DAC    ←2000
000204      CALL   F$AR
000205      OCT    000002
000206      DAC*   AMEAN
000207      CALL   F$AR
000210      OCT    000002
000211      DAC*   DEV
000212      CALL   F$CB
```

Figure A-1. Expanded Listing of STDDEV (Cont.)

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form*

TITLE: SERIES 16 DAP-16 AND DAP-16 MOD 2 ASSEMBLY LANGUAGE

ORDER No.: BY09, REV. 0

DATED: JUNE 1971

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME ____________________ DATE: ____________

COMPANY ____________________

TITLE ____________________

____

____

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

CUT ALONG LINE

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

Business Reply Mail

Postage Stamp Not Necessary if Mailed in the United States

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

FOLD ALONG LINE

Honeywell