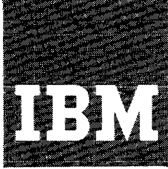
File No. 7090-21
Form C28-6235-3



Systems Reference Library

IBM 7090/7094 Programming Systems

FORTRAN II Assembly Program (FAP)

This publication describes the 7090/7094 FORTRAN II Assembly Program (FAP) in sufficient detail for the programmer to code in the FAP language. FAP and 7090/7094 FORTRAN II may be used with either the IBM FORTRAN Monitor or the 7090/7094 Basic Monitor (IBSYS). FAP is a machine-oriented symbolic language. The programmer can write the major part of his program in FORTRAN, using FAP subroutines where necessary to accomplish those parts of the job for which FORTRAN is not suitable; or he could write the major part of the program in FAP, using FORTRAN subroutines for certain computational and input/output operations.

MINOR REVISION (April, 1964)
This publication, Form C28-6235-3, is a reprint of the previous edition, Form C28-6235-2, incorporating changes released in Technical Newsletter N28-0084. This edition supersedes, but does not obsolete, previous editions and associated Newsletters.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Dept. D91, PO Box 390, Poughkeepsie, N.Y. 12602

© 1962, 1963 by International Business Machines Corporation

A 709/7090 machine language program is a sequence of machine instructions in the form of binary numbers that directs the computer in the performance of a particular task. A symbolic language program is a representation of a machine language program in a form that is more convenient to the programmer. For the most part, this involves the substitution of mnemonic symbols for the binary instruction desired, such as TRA for the binary unconditional transfer instruction. Because of its close relation to machine language, the symbolic language enables the programmer to use all the facilities of the computer that would be available to him if he were to code his program directly in machine language.

The translation of a programmer's symbolic language program into a machine language program is accomplished by an assembly program, or assembler, which interprets the mnemonics and substitutes the required binary instructions.

An assembler is similar to a compiler (such as FORTRAN) in that it produces machine language programs. However, the symbolic language used with an assembler is closely related to the machine language of the computer, while the source language used with a compiler resembles a language in which problems are commonly stated, such as mathematical notation.

Compilers have several advantages over assemblers: the language used with a compiler is easier to learn and to use; the programmer using a compiler usually does not need an intimate knowledge of the internal operations of the computer; programming is faster; and the time required to produce a finished, working program is greatly reduced, since there is less chance of programmer errors, and since most errors which are made are detected by the compiler.

The assembler compensates for these disadvantages by offering the programmer a degree of flexibility not available with any present-day compiler.

FAP (FORTRAN Assembly Program) provides a compromise between the convenience of a compiler and the flexibility of an assembler. Using FAP and 709/7090 FORTRAN operating under either the IBM FORTRAN Monitor or the 7090/7094 Basic Monitor (IBSYS), a programmer may write the major part of his program in FORTRAN, using FAP subroutines where necessary to accomplish those parts of the job for which FORTRAN is not suitable. Likewise, he may write the major part of the program in FAP, using FORTRAN subroutines for certain computational and input/output operations. For those jobs which are coded entirely in symbolic language, FAP may be used to produce an "absolute" program which can be either loaded by the Monitor or used independently of the Monitor.

Although the primary objective of FAP is the translation of symbolic language programs into machine language programs, it is also used for updating a symbolic tape by changing, deleting, or adding instructions.

A FAP program must be assembled on an IBM 709, 7090, or 7094, but can produce an object program that can be executed on an IBM 704, 709, 7090, or 7094.

This manual describes the FAP language and the programming facilities it affords. The reader is assumed to be familiar with the IBM reference manual, IBM 7090 Data Processing System, Form A22-6528.

CONTENTS

Part 1: The FAP Language 1
Chapter 1: Elements of the Language 1
Symbolic Card Format 1
Sequence Checking 1
Chapter 2: Types of Assembly 1
Relocatable Assembly 1
Absolute Assembly 2
Chapter 3: The Assembly Listing 2
Flags 2
Page Heading 3
Chapter 4: Symbols 4
Symbol Definition 4
Types of Symbols 4
Chapter 5: Symbolic Expressions 4
Elements and Terms. 4
The Characters * as an Element 4
Data Items 5
Literals. 6
Expressions 7
Evaluation of Expressions 7
Types of Expressions. 7
Boolean Expressions 8
Chapter 6: Symbolic Instructions 9
Location Field 9
Operation Field 9
Indirect Addressing 9
Variable Field 10
* (Remarks Card). 11

Part 2: Operations and Pseudo-Operations 12
Chapter 7: Machine Operations 12
Chapter 8: Extended Machine Operations 12
Sense Operations. 12
Selected and Related Operations 14
Additional Mnemonics 14
Prefix Codes 14
Chapter 9: Variable-Channel Tape Operations. 16
Chapter 10: Pseudo-Operations 18
First Card Group 18
Previously Defined Symbols 18
Phase Relocation Error 18
Chapter 11: Assembly Information Pseudo-Operations . . 18
The COUNT Pseudo-Operation 18
The END Pseudo-Operation 19
The IFF Pseudo-Operation 19
Chapter 12: Symbol-Defining 20
The EQU and SYN Pseudo-Operation 20
The BOOL Pseudo-Operation 20
The SET Pseudo-Operation. 20
The TAPENO Pseudo-Operation 21
The MAX Pseudo-Operation 21
The MIN Pseudo-Operation. 21
The SST Pseudo-Operation. 21
The HEAD Pseudo-Operation 21
The HED Pseudo-Operation 23
Chapter 13: Storage-Allocating Pseudo-Operations . . . 23
Location Counter and Program Counter 23
The BSS Pseudo-Operation 23
The BES Pseudo-Operation 24
The COMMON Pseudo-Operation 24
The ORG Pseudo-Operation 25
The LOC Pseudo-Operation 25
The EVEN Pseudo-Operation 25
Chapter 14: Data-Generating Pseudo-Operations 26
The OCT Pseudo-Operation 26
The DEC Pseudo-Operation 26
The BCI Pseudo-Operation 27
The BCD Pseudo-Operation 27
The VFD Pseudo-Operation 28
The ETC Pseudo-Operation 29
The DUP Psuedo-Operation 30
Chapter 15: Program-Linking Psuedo-Operations 31
The ENTRY Pseudo-Operation. 31
The CALL Pseudo-Operation 32
Standard Error Procedure Option 32
Subroutine Reference Using the $ Character 33
The IFEOF Pseudo-Operation 34
The EXTERN Pseudo-Operation 35
Chapter 16: Operation Code-Defining Pseudo-Operations . 35
The 704 Pseudo-Operation 35
The OPD Pseudo-Operation. 35
The OPVFD Pseudo-Operation 35
The OPSYN Psuedo-Operation 35
Machine Operation Code Definition 36
Chapter 17: Card Format-Control Pseudo-Operations. . . 36
The ABS Pseudo-Operation. 37
The FUL Pseudo-Operation. 37
The 9LP Psuedo-Operation 37
The TCD Pseudo-Operation 37
Chapter 18: List-Control Pseudo-Operations 37
The REM Pseudo-Operation 37
The SPACE Pseudo-Operation 38
The EJECT Pseudo-Operation 38
The UNLIST Pseudo-Operation. 38
The LIST Pseudo-Operation 38
The TITLE Pseudo-Operation 38
The DETAIL Pseudo-Operation 38
The LBL Pseudo-Operation 39
The PCC Pseudo-Operation. 39
The REF Pseudo-Operation. 39
The TTL Pseudo-Operation. 39
The INDEX Pseudo-Operation 39
The NULL Pseudo-Operation 40
Chapter 19: Binary Output from the Assembler 40
Relocatable Output 40
Absolute Output 40
The PMC Pseudo-Operation 40
Full Output 41
9LP Output. 41
Label and Serialization (FORTRAN Monitor) 41
FORTRAN Monitor Control Cards 41

Part 3: The Macro-Operation Processor 42
Chapter 20: General Description 42
Chapter 21: Macro-Definition Heading Card 44
The MACRO Pseudo-Operation 44
Extending the Argument List 44
Alternative Format of MACRO 44
The MOP Psuedo-Operation 44
Chapter 22: The Prototype 45
The Location Field of a Prototype Instruction. 45
The Operation Field of a Prototype Instruction 45
The Variable Field of a Prototype Instruction. 45
Chapter 23: Nesting Macro-Definitions 46
Chapter 24: Macro-Instructions 46
The MAC Pseudo-Operation 47
Punctuation in Macro-Instructions 47
Argument Strings. 47
Nested Macro-Instructions. 48
Chapter 25: The Generated Instructions 48
Created Symbols. 49
The NOCRS Pseudo-Operation 49
The ORGCRS Pseudo-Operation 49
Chapter 26: Additional Pseudo-Operations 50
The IRP Pseudo-Operation 50
The RMT and RMT* Pseudo-Operations 50

Part 4: Updating Symbolic Tapes. 52
Chapter 27: General Discussion. 52
Uses of the Update Facility 52
Blocked Update Output Tape 52
Sequence Checking and Serialization 52
Tape Positioning. 52
Illegible Input Instructions. 53
Updating a FORTRAN Source Program Deck 53
Listing Update Pseudo-Instructions 53
Chapter 28: Update Pseudo-Operations 53
The UPDATE Pseudo-Operation 53
The NUMBER Pseudo-Operation 54
The DELETE Pseudo-Operation 54
The IGNORE Pseudo-Operation 54
The SKIPTO Pseudo-Operation 54
The ENDFIL Pseudo-Operation 54
The REWIND Pseudo-Operation 55
The UNLOAD Pseudo-Operation 55
The SKPFIL Pseudo-Operation. 55
The UMC Pseudo-Operation 55
The ENDUP Pseudo-Operation. 56
The PRINT Pseudo-Operation 56
Chapter 29: Optional Serialization in Update Pseudo-Instructions 56
Chapter 30: Update Examples 56

Part 5: General Information 58
Chapter 31: Subroutines 58
Open and Closed Subroutines 58
Linkages 58
Calling Sequences 58
FORTRAN Linkage and Calling Sequences 59
Segmentation. 59
Common Storage 59
Relocatable Binary 60
Transfer Vector 60
Chapter 32: A Brief Description of the Assembly Process . 60

Appendix A: Combined Operations Table 63
Pseudo-Operations 63
Machine and Extended Machine Instructions. 63
Disk File Orders 67
Hypertape Orders 67
Appendix B: The FAP BCD Character Code. 68
Appendix C: System Symbol Table, FORTRAN Monitor . . 69
Appendix D: The Assemble Only Mode of FAP Operating Under The Basic Monitor 70
IBSFAP Operations 70
Control Cards. 70
System Symbol Table 70
Appendix E: Field Modifications to FAP. 71
Appendix F: Table Limits 72

CHAPTER 1: ELEMENTS OF THE LANGUAGE

FAP incorporates all 704, 709, 7090, and 7094 machine language and extended operation codes described in the various machine reference manuals. A list of all permissible operation codes is given in Appendix A, "Combined Operations Table."

A symbolic instruction consists of five major divisions: location field, operation field, variable field, comments field, and identification field.

The location field may contain a name by which the symbolic instruction may be referenced. The operation field contains the name of the operation to be performed, and the variable field contains the location of the operand or, in certain cases, the operand itself (e.g., ARS 16). The comments field may contain the programmer's comments, but has no effect on the machine operation. The identification field may be used to insure safety in card handling and is useful in updating. Figure 1-1 illustrates the various fields of a symbolic instruction. The instruction in Figure 1-1 might be the first instruction of a routine entered by a transfer instruction whose variable field contains the symbolic address CASEB. The machine operation here is "Clear and Add," and the address of the operand is a storage location that is referred to as TMP. The comment "THIRD CASE" is for the use of the programmer. The serialization F0113750 indicates the relative position of this instruction in the program deck.

Symbolic Card Format

Symbolic instructions are punched one per card in the following format. The location field, which may be blank, occupies card columns 1-6. Card column 7 is always blank. The operation field begins in column 8 and is from three to seven characters in length. A blank column or a comma separates the operation field and the variable field. The variable field may begin immediately following the operation field, but must not begin after column 16. An open parenthesis immediately following the operation code terminates the operation field and is considered part of the variable field. The variable field may not extend beyond column 72 and must be followed by a blank column to separate it from the comments field. If the variable field includes column 72, the assembler assumes that a terminating blank is present.

The comments field follows the variable field and extends through column 72. In the absence of a variable field, the comments field may not begin before column 17. Columns 73-80 are used for identification and serial numbering. Figure 1-2 illustrates the several different ways of separating the operation field and the variable field. See Chapter 3 for a description of the assembly listing.

Sequence Checking

If a BCD source deck is serialized in columns 73-80, sequencing information will be checked, and any card out of sequence will be listed both on- and off-line. If a group of correctly sequenced cards is inserted into a deck out of sequence, only the first card of the group will be listed.

For purposes of sequencing, a blank is not considered to be zero; it is given the octal value 60. A serialized card following a card with all blanks in card columns 73-80 will not be sequence checked.

CHAPTER 2: TYPES OF ASSEMBLY

FAP permits two types of assembly: relocatable and absolute.

Relocatable Assembly

Every relocatable program or subprogram produced by the FORTRAN Monitor System nominally begins in location zero. Since a job to be executed may contain several subprograms, it is obvious that they may not all be loaded into locations starting with location zero. In fact, no program is ever loaded at location zero; instead, each program is relocated. The first program or subprogram is loaded into lower core storage. Successive subprograms are then loaded into successively higher locations of core storage, each beginning with the location after the last location of lower core storage used by the preceding subprogram. When a particular program has been loaded, the address of the first word is called that program's load address, and the address-plus-one, relative to zero, of the last word is called that program's program break.

Thus, the address in core storage actually occupied by an instruction of the program is the

```
CASEB CLA       TMP                 THIRD CASE ••••                                        F0113750
```

Figure 1-1

```
00000   0500 00 0 00004         CLA TEMP TEMP IS THE FIRST SUBFIELD
00001   0500 00 0 00004         CLA,TEMP TEMP IS THE FIRST SUBFIELD
00002   0500 00 4 00000         CLA ,TEMP FIRST SUBFIELD VOID, TEMP IS SECOND SUBFIELD
00003   0500 00 0 00006         CLA(TEMP (TEMP IS THE FIRST SUBFIELD
```

Figure 1-2

address assigned to that instruction during assembly plus the load address of that program. To keep the program self-consistent, the load address must be added to the addresses and decrements of many (but not all) of the instructions.

This process of conditionally adding the load address is performed by the loading program prior to execution and is called relocation. In relocating instructions, the loading program is guided by relocation indicator bits that are inserted into the binary object program cards when the program is compiled or assembled. References to common storage are subject to a special type of relocation that is controlled by control cards during loading.

A more extensive discussion of relocation is given in Part 5.

Absolute Assembly

The binary object program resulting from an absolute assembly is loaded into locations specified by the programmer in his source program by the use of the ORG pseudo-operation. The loader makes no adjustment to any field during loading.

CHAPTER 3: THE ASSEMBLY LISTING

The printed output of a FAP assembly is called the assembly listing. This listing has three parts: the pre-processor assembly listing; the assembled program listing; and the post-processor assembly listing.

A subheading is provided for the pre-processor and post-processor listings, each part being separately paginated. The programmer may provide subheadings for the assembled program listing by using the TTL pseudo-operation.

The pre-processor assembly listing is composed of the results of the update pseudo-instructions used in the program and an indication that no card count estimate was given (if applicable).

The assembled program listing is essentially a printout of the symbolic cards in the order in which they appeared in the symbolic deck, together with the octal representation of the binary words produced by the assembler. A portion of a typical listing is given in Figure 3-1.

```
    00004  0500 00 0 00011   CASEB CLA    TMP
    00005  0774 00 4 00014         AXT    12,4
    00006  0400 00 4 00031         ADD    FUNC+12,4
    00007  2 00001 4 00006         TIX    *-1,4,1
    00010  0020 00 0 00034         TRA    CALC
    00011                      TMP BSS    4
    00015                     FUNC BSS 12
    00031  0 00000 0 00000         ...
    00032  0760 00 0 00000   CASEC CLM
    00033  0 00000 0 00000         ...
O   00034    00000 0 00015    CALC CLW    FUNC
```

Figure 3-1

The left-hand portion of the listing is that produced by the assembler. The first column shows the location of each instruction, in octal. The next column shows, in octal, the binary word assigned to that location; machine operations are broken up into their appropriate parts. The right-hand portion of the assembly listing is a listing of the symbolic deck.

Note that the last symbolic instruction contains an error. The operation code was incorrectly punched. The assembler has indicated this error by placing the error flag O in the left margin opposite the erroneous instruction, and it has left blank the first (prefix) digit of the octal word.

The first item on an assembly listing is the transfer vector and linkage director. Following this is a list of the instructions in the symbolic deck, together with the octal representation of the words generated. If any literals are used in the program, a list of the data words that are generated will follow.

The post-processor assembly listing is composed of the program break (the first location, in octal, not used by the program), the COMMON break (the first location, in octal, below the common storage area required by the program), the Symbolic Reference Table, and an indication of whether or not the assembly was in error. If there are errors in the program, they will be flagged in the assembled program listing (see description of flags, below). The Symbolic Reference Table includes all symbols which appear in the location field of an instruction and all program counter references to each symbol. In addition, the locations of definitions of symbols defined by BOOL, COMMON, EQU, MAX, MIN, SYN, and TAPENO will appear in this table. Multiply-defined symbols will be flagged M. A table of references to undefined symbols will follow the table of references to defined symbols.

Flags

The FAP assembler indicates that it has detected an error or inconsistency by placing a flag in the left margin of the assembly listing opposite the instruction containing the error. The left margin of the listing is reserved for flags and will be left blank if no errors are detected.

There are two types of flags: error flags that cause printing of the message ERROR IN ABOVE ASSEMBLY and suspension of relocatable binary output; and warning flags that indicate a possible inaccurate or incomplete instruction.

The following are error flags:

B This flag indicates that the flagged instruction contains a relocatable or common symbol in a Boolean expression, or a digit greater than 7 in an octal expression.

E This flag indicates that a data-generating pseudo-operation contains an error. If the pseudo-operation is DEC, the field in error will contain zeros; if the pseudo-operation is BCD or BCI, the field in error will contain blanks.

L This flag indicates that the flagged instruction contains an error in a literal.

O This flag indicates that the operation code in the flagged instruction is undefined or invalid. Indirect addressing of a pseudo-operation will result in an undefined operation code.

( ) This flag indicates unbalanced parentheses in a macro-definition or macro-instruction.

P This flag indicates that the flagged pseudo-operation either contains an invalid reference to a symbol which has not been previously defined (phase error), or it is a storage-allocating pseudo-operation containing a relocatable or common expression in its variable field (phase relocation error).

R This flag indicates that the flagged instruction contains an expression which is a relocation error.

U This flag indicates that the flagged instruction contains a reference to an undefined symbol; the field in error will be blank.

The following are warning flags:

N This flag indicates the use of an operation code which is defined by the programmer (through the use of the OPD or OPVFD pseudo-operation) and for which the mode of assembly is not specified.

4, 9 If a 704 instruction appears in a 7090 assembly, it will be flagged with a 4; if a 7090 instruction appears in a 704 assembly, it will be flagged with a 9.

A This flag will appear whenever the address field of a machine instruction or the variable field of a pseudo-instruction is expected and is missing.

T This flag indicates that a tag field of a machine instruction is expected and is missing.

D This flag indicates that a decrement field of a machine instruction is expected and is missing or that a decrement field is provided for a Type B or E instruction, or that a decrement field longer than eight bits is provided for a Type C instruction. Assembly of decrement fields in Type B or E instructions is provided for compatibility with existing FAP programs.

I This flag indicates that a decrement field in a Type B or Type C instruction appears to be an indirect address, or that indirect addressing has been specified by an asterisk for an operation code which is not indirectly addressable.

F This flag indicates either an excessive field in a machine instruction or pseudo-instruction or an improper field in a pseudo-instruction. Excessive fields in Type D instructions will be flagged with an F and will not result in additional bits in the generated machine word. Improper fields in symbol-defining pseudo-instructions will be flagged with an F, but will not cause relocatable binary output to be deleted.

```
BETA SYN       ALPHA
```

Figure 3-2

If ALPHA has not been previously defined, the instruction in Figure 3-2 will be flagged with an F. BETA will remain undefined, and any reference to BETA will be flagged with a U, which will cause relocatable binary output to be deleted.

The following flag may be either an error flag or a warning flag:

M This flag indicates that the instruction contains a reference to a symbol which has been defined more than once (multiply). If a symbol is multiply defined by its appearance in the location field of more than one of the following operations, the point of definition will be given the warning flag M: BCD, BCI, BES, BOOL, BSS, CALL, COMMON, DEC, DUP, END, EQU, IFEOF, MAX, MIN, OCT, SYN, TAPENO, VFD, or any machine instruction. However, any reference to a multiply-defined symbol in the variable field of a machine operation or pseudo-operation will be given the error flag M. It is possible to have two M flags in the left-hand margin opposite a single instruction: one referring to the point of definition (location field); the other, to the point of use (variable field). The M flag also indicates redefinition of an existing operation code (see Chapter 20, "General Description").

Page Heading

The assembly listing tape is intended to be printed on an off-line printer that utilizes programmed carriage control. (Note: The number of lines per page is preset; it may be changed by a modification to IOP.) A page number appears at the top of each page; page numbering starts anew with page 1 for each assembly. The page numbering may be reset by a TTL pseudo-instruction. By using a page title card, the programmer may cause the assembler to write one line of information at the top of every page. By using a TTL pseudo-instruction, he may cause the assembler to write a second line, or subheading.

The page title card is identified as the first card with an * or $ in column 1 in the first card group (see

"First Card Group" in Chapter 10). If there is no such card in the first card group, the page will be headed by the page number and, if a date card has appeared, the date. Any pseudo-instruction in the first card group may precede the page title card. The Monitor control cards that precede the symbolic deck are not regarded as part of it. If a page title card appears in the first card group, the contents of columns 2-72 of this card will appear at the top of every page of the assembly listing together with the page number.

CHAPTER 4: SYMBOLS

A symbol (also referred to by the terms "location symbol" and "symbolic address") is a string of one to six non-blank alphabetic and numeric characters, at least one of which is non-numeric. In addition, a symbol may include the characters decimal point, left parenthesis, and right parenthesis. A symbol may not include the following set of special characters:

+ plus sign	$ dollar sign
- minus sign	= equal sign
* asterisk	' apostrophe
/ slash mark	, comma

For example,

A	A.1
1234X	(I2

are all valid symbols. A symbol may be used as a name for a storage location, tape address, or other program parameter. The use of symbols that contain embedded parentheses is restricted in macro-operations (see page 47).

Symbol Definition

A symbol is defined in one of three ways:

1. By its appearance in the location field of an instruction; or
2. By its use as the name of a subprogram; or
3. By its appearance in a System Symbol Table (see SST pseudo-operation, page 21).

Every symbol used in the program must be defined only once. A flag will be given by the assembler if any symbol is used but never defined, or if any symbol is defined more than once. The SET pseudo-operation may be used to define symbols so that they may be redefined (see page 20).

Types of Symbols

Each symbol encountered in the assembly process will be classified according to type at the time it is defined. In a relocatable assembly, FAP recognizes three types of symbols. An absolute symbol refers to a specific number, such as a tape address; it is usually not used to refer to a location in core storage. A common symbol refers to a location in common storage (common storage is described on page 59). Any symbol that is not either absolute or common is classified as relocatable; in particular, a symbol that occurs in the location field of an instruction is a relocatable symbol, except for symbols that occur in the location field of a symbol-defining pseudo-operation. In the case of the symbol-defining pseudo-operation, the symbol in the location field is defined to be of the same type as the symbol in the variable field.

When an absolute assembly has been specified by the use of the ABS, FUL, or 9LP pseudo-operation, all symbols are treated as absolute symbols.

CHAPTER 5: SYMBOLIC EXPRESSIONS

In writing symbolic instructions, the programmer is concerned with the problem of building expressions to represent, in the case of machine instructions, the address, tag, and decrement or count portions of the instructions. Expressions are also used in the variable fields of channel commands, I/O device orders, or pseudo-instructions, in accordance with the rules set forth in each specific case.

Before discussing expressions, it is necessary to describe their components: elements, terms, and operators.

Elements and Terms

The smallest components of an expression are elements. An element is either a single symbol or a single integer less than 2^{36} (the asterisk may also be used as an element; see below). An absolute, relocatable, or common symbol is regarded as an absolute, relocatable, or common element, respectively. An integer is always an absolute element.

A term is a string of elements and the operators:

* (multiplication)
/ (division)

A term may consist of a single element, of two elements separated by * or /, of three elements separated by two operators, etc. A term must begin with an element and end with an element. It is not permissible to write two operators in succession or to write two elements in succession.

Examples of terms are given in Figure 5-1.

```
TMP*FUNC*TAXY
FIRST/SCND*THRD*4
3
6*4096
5*X
OFICA
```

Figure 5-1

The Character * as an Element

In addition to being used as an operator, the asterisk

is also used as an element. When it is used in this way, the asterisk is a relocatable element that stands for the location of the instruction in which it appears. Thus, the element * will have different values in different instructions.

```
06313  0020 00 0 06315    ALPHA  TRA    *+2

06313  0020 00 0 06315    ALPHA  TRA    ALPHA+2
```

Figure 5-2

The instructions in Figure 5-2 are equivalent and each represents a transfer to the second location following the location containing the transfer instruction. There is no ambiguity between this use of the asterisk and the use of the asterisk to denote multiplication, since the position of the asterisk always makes clear what is meant. Thus **10 means "the location of this instruction multiplied by 10."

The expressions

**

and

-

are commonly used for listing purposes to denote an address or decrement which must be computed by the program. Both are absolute expressions whose value is zero.

Data Items

Data items may also be considered elements of expressions when used in the variable field of certain pseudo-instructions. There are three types of data items: octal, decimal, and alphameric.

Octal Data Items. An octal data item specifies, in octal, a word, or part of a word, of data to be converted to binary. An octal data item may be specified in one of three ways:

1. It may be preceded by the characters = O to form an octal literal; or
2. It may be a subfield of the variable field of a BOOL, OCT, or VFD pseudo-instruction; or
3. It may be the variable field of a Type D machine instruction.

One type of octal data item is recognized by FAP: the octal integer. An octal integer is composed of a string of not more than twelve digits, from 0 through 7, which may be preceded by a plus or minus sign.

Decimal Data Items. A decimal data item specifies, in decimal, a word, or part of a word, of data to be converted to binary. Decimal data items may be specified in the following three ways:

1. It may be preceded by the character = to form a decimal literal; or

2. It may be a subfield of the variable field of a DEC pseudo-instruction; or

3. It may be a subfield of the variable field of certain other pseudo-instructions (such as VFD) which do not require an octal or Boolean subfield (this use is restricted to decimal integers).

There are three types of decimal data items: decimal integers, floating-point numbers, and fixed point numbers.

1. Decimal integers. A decimal integer is composed of a string of digits, from 0 through 9, which may be preceded by a plus or minus sign. A decimal integer is distinguished from other types of decimal data items by the absence of the letter B, the letter E, and the decimal point (the use of the letters B and E is described below).

2. Floating-point numbers. A floating-point number has two components:
   a. The principal part is a signed or unsigned decimal number written with or without a decimal point. The decimal point may appear at the beginning, at the end, or within the principal part, or it may be omitted if the exponent part is present. If the decimal point is omitted, it is assumed to be located at the right-hand end of the principal part.
   b. The exponent part consists of the letter E followed by a signed or unsigned decimal integer. The exponent part must follow the principal part; it may be omitted if the principal part contains a decimal point.

A floating-point number is distinguished from a decimal integer by the presence of a decimal point or the letter E, or both. It is distinguished from a fixed-point number by the absence of the letter B.

3. Fixed-point numbers. A fixed-point number has three components:
   a. The principal part is a signed or unsigned decimal number written with or without a decimal point. The decimal point may appear at the beginning, at the end, or within the principal part, or it may be omitted if the exponent part is present. If the decimal point is omitted, it is assumed to be located at the right-hand end of the principal part.
   b. The exponent part consists of the letter E followed by a signed or unsigned decimal integer. The exponent part must follow the principal part; it may be omitted if the principal part contains a decimal point.
   c. The binary-place part consists of the letter B followed by a signed or unsigned decimal integer. The binary-place part must be present in a fixed-point number and must follow the principal part. If the number has an exponent part, the binary-place part may precede or follow the exponent part.

A fixed-point number is distinguished from other types of decimal data items by the presence of the letter B.

A decimal integer may represent any positive or negative binary number whose magnitude is less than 2^{35}. For example, the decimal integer -31 would be converted to the 36 bit number whose octal representation is

400000000037

A floating-point number will be converted to a normalized floating-point binary word in the standard floating-point binary format. If present, the exponent part specifies a power of ten by which the principal part will be multiplied during conversion. For example, all of the following floating-point numbers are equivalent and will be converted to the same floating-point binary number:

3.14159
31.4159E-1
314159.E-5
314159E-5
.314159E1

A fixed-point number is converted to a fixed-point binary number which contains an understood binary point. The purpose of the binary-place part of the number is to specify the location of this understood binary point within the word. The number that follows the letter B specifies the number of binary places in the word to the left of the binary point (that is, the number of integral places in the word). The sign bit is not counted. Thus, the binary-place part 0 specifies a 35-bit binary fraction. B2 specifies two integral places and 33 fractional places. B35 specifies a binary integer. B-2 would specify a binary point located two places to the left of the leftmost bit of the word; that is, the word would contain the low-order 35 bits of a 37-bit binary fraction. As with floating-point numbers, the exponent part (if present) specifies a power of ten by which the principal part will be multiplied during conversion.

In the process of shifting the converted word to position the binary point, significant bits may be shifted past the right-hand end of the word and lost; no error will be indicated. However, if nonzero bits must be shifted past the left-hand end of the word, an error will be indicated by the assembler. Thus, the integral part of a fixed-point number must be small enough to fit in the number of integral places allowed. Also, if the binary-place part is zero or negative, the number must be an appropriately small fraction.

For example, the following fixed-point numbers specify the same configuration of bits; but not all of them specify the same location for the understood binary point:

22.5B5
11.25B4
1125B4E-2
1125E-2B4
9B7E1

All of these fixed-point numbers will be converted to the binary configuration whose octal representation is 264000000000.

Alphameric Data Items. An alphameric data item is a string of alphabetic and numeric characters. In certain cases, the alphameric string may contain the special characters. An alphameric data item may be specified in the following two ways:

1. Preceded by the characters =H to form an alphameric literal; or
2. As a subfield of the variable field of a BCD, BCI, or VFD pseudo-instruction.

Literals

Often a programmer wishes to refer to a location containing a constant. For example, if he wishes to add the number 1 to the contents of the accumulator, somewhere in core storage he must have a location containing the number 1. Pseudo-operations are provided to allow the introduction of data words and constants into the program, but often this introduction is more easily accomplished by the use of a literal.

In contrast to other types of subfields, the content of a literal subfield is itself the data to be operated upon. The appearance of a literal directs the assembler to prepare a constant that is equivalent in "value" to the contents of the literal subfield, to store this constant in a location at the end of the program, and to replace the address field of the instruction containing the literal with the address of the constant thus generated. Three types of literals are permitted: decimal, octal, and alphameric.

A decimal literal consists of the character = followed by a decimal data item (see page 5). Thus, the instruction in Figure 5-3 means "multiply the contents of the MQ by the number -3." (That is, multiply the contents of the MQ by the contents of a location which contains the number 400000000003_8.)

```
MPY          =-3
```

Figure 5-3

An octal literal consists of the character =, followed by the letter O, and then followed by a signed or unsigned octal integer. Thus, the instruction in Figure 5-4 means "perform the operation 'And to Accumulator' with an operand word whose leftmost 31 bits are zeros and whose rightmost five bits are ones.

```
ANA          =O37
```

Figure 5-4

An alphameric literal consists of the character =, followed by the letter H, and then followed by six characters of alphameric data. Thus, after the execution of the instruction in Figure 5-5, the contents

of the MQ would be 010221226060_8. The six characters following the letter H are taken as data even if one or more of them is a comma or a blank.

```
LDQ       =H12AB
```

Figure 5-5

A literal may occur only as the address subfield of the variable field of a machine instruction. This subfield must consist solely of that literal. A literal may not appear as a tag or decrement, or in the variable field of a pseudo-instruction. Furthermore, a literal may not appear in the variable field of a Type D machine operation.

Other subfields may be present following an address subfield containing a literal. In this case, the separating comma is used in the usual manner, except that, when an alphameric literal is used, the separating comma must be the eighth character following the equal sign.

A decimal literal may not contain anything other than a legitimate decimal data item. An octal literal may not contain any characters other than

+ - 0 1 2 3 4 5 6 7

If this rule is violated, the address field of the octal portion of the listing of that instruction will be left blank, and the error flag will be given.

The data words generated by literals are sorted according to their magnitudes when regarded as 36-bit positive numbers, and then they are assigned to consecutively higher locations following the highest location which the assembler has assigned to an instruction word, a data word, or a block of storage other than common. Many literals referring to the same binary data word cause only one data word to be generated.

Expressions

An expression is a string of terms separated by the operators:

+ (addition)

- (subtraction)

An expression may consist of a single term, of two terms separated by + or -, of three terms separated by two operators, etc. It is not permissible to write two operators in succession or to write two terms in succession, but an expression may begin with + or -. Examples of expressions are given in Figure 5-6.

```
3
OFICA
TMP-4
-77
-TMP*FUNC/X-7*H+13759601*YMNG*ZWTY/4+3
```

Figure 5-6

Evaluation of Expressions

An expression is evaluated as follows. First, each element is replaced by its numerical value. Second, each term is evaluated by performing the indicated multiplications and divisions from left to right, in the order in which they occur. In division, the integral part of the quotient is retained, and the remainder (which has the same sign as the dividend) is immediately discarded. In multiplication, the low-order 35 bits are retained.

For example, the value of the term 5/2*2 is 4. In the evaluation of an expression, division by zero is equivalent to division by one and is not regarded as an error; thus, an expression that reduces to 0/0 can be evaluated, and is equal to 0. Third, the terms are combined from left to right, in the order in which they occur. If the result is negative, it is replaced by its 2's complement (that is, the number 2^{36} is added to a negative result). Finally, this result is reduced modulo 2^{15} (except in the variable field of a VFD or BOOL pseudo-operation); that is, only the rightmost 15 bits are retained, and the preceding bits are dropped. Grouping of terms, by parentheses or otherwise, is not permitted, but this restriction may often be circumvented. For instance, the product of A with the quantity B+C may be expressed as

A*B+A*C

Types of Expressions

In addition to evaluating expressions, the assembler must decide whether each expression is absolute, common, or relocatable. Without this decision the assembler would be unable to assign the proper relocation indicator bits for the information of the loading routine (see "Relocatable Binary," in Chapter 31). The rules by which the type of expression is determined are as follows:

A relocatable element is a relocatable expression.

A relocatable element plus or minus an absolute element is a relocatable expression.

An absolute element is an absolute expression.

Any expression containing only absolute elements is an absolute expression.

The difference of two relocatable elements is an absolute expression.

A common element is a common expression.

A common element plus or minus an absolute element is a common expression.

The difference of two common elements is an absolute expression.

A relocatable element plus a common element minus another common element is a relocatable expression.

These rules may be clarified by examples.

Assume that a programmer wishes to incorporate a table into his program, and he knows that later

he may wish to add or delete items in the table without changing program references to it. His first step is to assign symbols to the low-order word in the table and to the location immediately after the high-order word of the table; these symbols could be BGTBL and ENTBL, respectively. Regardless of the number of items in the table or of the number of later additions or deletions, the number of words in the table will always be equivalent to the value of the expression ENTBL-BGTBL.

This illustrates the rule that the difference of two relocatable elements is an absolute expression.

As another example, assume that the same programmer wishes to employ a second table of the same length as the first. He may indicate the low-order word of the second table by the symbol STBL. Then, the location following the high-order word of the second table may be indicated by the expression

STBL+ENTBL-BGTBL

This address is subject to relocation; hence, the expression must be a relocatable expression.

The following expressions are examples of relocation errors. If such an expression appears in the source program, it will be flagged.

The negative (complement) of a relocatable element.

The negative (complement) of a common element.

An absolute element minus a relocatable element.

An absolute element minus a common element.

The sum of two relocatable elements.

The sum of two common elements.

The sum of a relocatable element and a common element.

A product or quotient involving a relocatable or common element.

The discussion that follows describes a procedure for determining the type of an expression. First, discard any term that contains only absolute elements. Next, examine each term of the expression. If any term contains more than one relocatable element, more than one common element, or one common element and one relocatable element, the expression is a relocation error. Also, if in any term the character / follows a relocatable or common element, the expression is a relocation error. For example, if TRANS and FUNC are relocatable (or common) symbols, then the expression

TRANS*FUNC+TRANS*2/2

violates both the above rules.

If the expression passes these tests, replace each relocatable element by the symbol r, each common element by the symbol k, and each absolute element by its value. This yields a new expression which involves only numbers and the symbols r and k. Evaluate this expression using the rules given in the section above. If the result is a number, including zero, then the original expression is absolute. If the result is r, then the original expression is relocatable. If the result is k, then the original expression is common. If the result is anything else, the original expression is a relocation error.

The following examples illustrate this procedure.

Example 1.

Consider the expression

4+3*TRANS-2*FUNC+COMX-COMY+COUNT

where TRANS and FUNC are relocatable symbols, COMX and COMY are common symbols, and COUNT is an absolute symbol.

Discarding the terms involving only absolute elements leaves

3*TRANS-2*FUNC+COMX-COMY

This does not contain any illegal terms, so replacing each symbol by the letter r or k results in

3*r-2*r+k-k

Evaluating this gives r, so the original expression is relocatable.

Example 2.

Consider the expression

4/2*3*TRANS-FUNC+COMX

This reduces to

4/2*3*r-r+k

or

5r+k

This is not r, k, or a number, so the expression is a relocation error.

Example 3.

Consider the expression

N*2*F-N*N*F+N/2*N*K-N/2*K+5

where N is an absolute symbol, F is a relocatable symbol, and K is a common symbol.

It is an absolute expression if the value of N is zero, a relocatable expression if the value of N is 1, a common expression if the value of N is 2, or a relocation error if the value of N is anything else.

In an absolute assembly, all symbols are treated as absolute symbols; hence, all non-Boolean expressions are absolute expressions, and a relocation error is impossible.

Boolean Expressions

An expression is Boolean if:

1. It forms the variable field of a BOOL pseudo-instruction; or
2. It forms an octal subfield of the variable field of a VFD pseudo-instruction; or
3. It forms the variable field of a Type D machine instruction. (The Type D machine instructions are SIL, SIR, RIL, IIL, IIR, LNT, RNT, LFT, and RFT.)

In most cases a Boolean expression is simply an octal integer. The two instructions in Figure 5-7 are equivalent, but the first is more convenient since it has the octal representation of tape A1. Most programmers will not use Boolean expressions other

than octal integers, and may ignore the remainder of this section.

```
01201     TAPEX BOOL     1201
01201     TAPEX EQU      641
```

Figure 5-7

In a Boolean expression, the four operators +, -, *, and / have Boolean meanings rather than their usual arithmetical meanings, as given in Figure 5-8.

+ (or, inclusive or, union)	— (exclusive or symmetric difference)
0+0=0	0-0=0
0+1=1	0-1=1
1+0=1	1-0=1
1+1=1	1-1=0
* (and, intersection)	/ (complement, ones complement, not)
0*0=0	/0=1
0*1=0	/1=0
1*0=0	
1*1=1	

Figure 5-8

Although / is usually an operation involving only one term, by convention A/B is taken to mean A*/B. Thus, the table for / as a two-term operation is given in Figure 5-9.

0/0=0
0/1=0
1/0=1
1/1=0

Figure 5-9

Other conventions are given in Figure 5-10

Expression	
+A=A+=A	one operand missing
-A=A-=A	one operand missing
A=A=0	one operand missing
A/ =A	one operand missing
+=0	both operands missing
-=0	both operands missing
*=0	both operands missing

Figure 5-10

The tables in Figures 5-8, 5-9, and 5-10 define the four Boolean operations for one-bit quantities. The operations are extended to 36-bit quantities since each bit-position is treated independently. The Boolean operator /, with one operand missing, is made equivalent to 777777777777_8. A Boolean expression is evaluated as follows: First, all integers are taken as octal and must be less than 2^{36}. The operations * and / are carried out from left to right, all operands being regarded as having 36 bits, and then the operations + and - are carried out from left to right, all quantities being regarded as having 36 bits. The rightmost 18 bits are preserved and the remaining bits discarded, except in the variable field of a VFD pseudo-instruction, in which case the number of bits preserved may vary from 1 to 36. Any use of a relocatable or common symbol in an octal or Boolean expression will be flagged as a Boolean error.

CHAPTER 6: SYMBOLIC INSTRUCTIONS

Symbolic instructions are composed of a location field, an operation field, and a variable field. Instructions are punched one per card in the format described on page 1.

Location Field

The location field of a symbolic instruction should either be blank or contain a single symbol, possibly preceded or followed by blanks. Blanks that are embedded in a location symbol are ignored. The purpose of using a location symbol is to give a name to the instruction with which the location symbol is associated, so that the instruction may be referred to by this name in other instructions of the program. Except in the case of the symbol-defining pseudo-operations, a symbol in the location field of an instruction has as its value the address assigned to that instruction.

Operation Field

The operation field of a symbolic instruction will normally contain an alphabetic code representing a machine operation, an extended machine operation, a variable-channel operation, a macro-operation, or a pseudo-operation. A blank operation field will be interpreted in the same manner as the extended operation PZE; that is, a word will be assembled whose prefix is zero. In this connection, note that a blank card in the program deck causes a word of zeros to be generated in the program.

Anything appearing in the operation field that is not among the set of recognized instructions (see Appendix A: Combined Operations Table) is an invalid operation code and will be given the error flag O.

Indirect Addressing

The character * may appear in the operation field of a symbolic instruction immediately to the right of the last character of the operation code. The presence of this character indicates indirect addressing and specifies that the assembler is to insert the appropriate bit or bits into the binary word that corresponds to the symbolic instruction. Bits 12 and 13 are used for machine instructions; bit 18 is used for channel commands. If an instruction so designated is not

indirectly addressable, the warning flag I will be given.

Variable Field

When writing a machine instruction in symbolic form, the programmer may, and sometimes must, specify certain combinations of address, tag, decrement (or count), and mask. For example, a TIX instruction requires an address, tag, and decrement; CLM should not have an address, tag, or decrement; and TCM should have an address, decrement, and mask, but should not have a tag. (The requirements for each machine instruction are explained in detail in the machine reference manuals and are tabulated in Appendix A.)

The address, tag, decrement (or count), and mask subfields of an instruction are specified in that instruction's variable field, in that order. Note that this is the reverse of the internal machine order, which is mask, decrement (or count), tag, and address. Any subfields may be absent, provided that the subfields following it are also absent. Any subfield that is present consists of one symbolic expression (but, see below for zero subfields). Adjacent subfields are separated by commas as illustrated in Figure 6-1.

```
TIX       GAMMA,4,1
```

Figure 6-1

Figure 6-1 specifies an address GAMMA, a tag of 4, and a decrement of 1.

The variable field begins with the first non-blank character following the blank character or the comma, immediately following the operation code, which terminates the operation field. The first character of the variable field may not appear before column 12, or after column 16, with the exception of a left parenthesis which may appear in column 11. When a left parenthesis immediately follows the operation code, it is considered part of the variable field. The end of the variable field is indicated by the appearance of a blank character (except in the case of BCI, BCD, and alphameric literals or certain subfields in macro-definitions and macro-instructions). There may be no blanks between subfields or within any subfield of the variable field, except in the cases listed above.

A subfield that is irrelevant may not be absent if it precedes a subfield that is required. Such a subfield may contain a zero, be void, or contain ** to indicate a field that is to be initialized. The equivalent instruction in Figure 6-2 illustrates that when the contents of a subfield are to be zero, the character 0 may be omitted, leaving the subfield void, of course, the separating comma(s) must be present. Also, if one or more subfields at the right-hand end of the variable field are to be zero, these subfields may be omitted entirely, together with their separating commas. However, omitting subfields may cause a warning flag to be given. As an example, the members within each of the three sets of symbolic instructions in Figure 6-3 are equivalent.

```
PXA       0,4
PXA       ,4
```

Figure 6-2

```
   00000  0634 00 0 00011        SXA    BETA,0
T  00001  0634 00 0 00011        SXA    BETA
   00002  0634 00 0 00011        ZSA    BETA

   00003  3 00000 0 77775        TXH    GAMMA,0,0
TD 00004  3 00000 0 77775        TXH    GAMMA
   00005  3 00000 0 77775        BRN    GAMMA

   00006 -0754 00 0 00000        PXD    0,0
T  00007 -0754 00 0 00000        PXD
   00010 -0754 00 0 00000        ZAC
```

Figure 6-3

However, the second member of each triplet is flagged to indicate the omitted field. The third member of each triplet has the same meaning as the first and the second, but is not flagged.

Any valid expression that appears in a subfield of the variable field will be evaluated according to the rules given in the section "Evaluation of Expressions," page 7. However, after the expression in the tag subfield has been evaluated, only the rightmost three bits will be used; that is, the tag is reduced modulo eight. If an instruction is not permitted a decrement, or if a Type C instruction has a decrement larger than eight bits, the entire field will be used, and the warning flag D will be given. In general, if a required address, tag, or decrement of an instruction is omitted, the instruction will be given an A, T, or D warning flag, respectively.

If an expression in a subfield of the variable field contains an undefined symbol, the corresponding field will be left blank in the assembly listing, and the error flag U will be given. If an expression in the variable field of an instruction contains a symbol which is defined more than once, the error flag M will be given. If an instruction has too many fields, the warning flag F will be given and the extra field(s) will be ignored.

The variable field of a disk order consists of three subfields: access and module, track, and record. If a required subfield is omitted, the warning flag A will be given. If a subfield is not required, but is coded, the subfield is assembled, but the warning

flag F will be given.

If a required variable field of a pseudo-instruction is omitted, the warning flag A will be given.

*(Remarks Card)

Any card with an * in column 1 is called a remarks card. When such a card is encountered, columns 2 through 72 are treated as commentary. This commentary is printed out as a single line on the assembly listing, exactly as it is written. A remarks card has no other effect on the processing of the source program.

Any card with a $ in column 1 is also treated as a remarks card.

CHAPTER 7: MACHINE OPERATIONS

The FAP language includes all standard 704, 709, 7090, and 7094 machine operations described in the machine reference manuals. A machine instruction, channel command, or order consists of the following:

1. A symbol or blanks in the location field;
2. The operation code in the operation field; and
3. The address, tag, decrement (or count), and mask subfields of an instruction or command, or the various subfields of an order in the variable field.

The assembly of such an instruction, command, or order involves the following functions:

1. If there is a symbol in the location field, this symbol is defined to be the next location to be assigned by the assembler when the instruction, command, or order is encountered.
2. The operation code of an instruction or command is translated into a 36-bit binary word, which is called the instruction word. The bits which determine the operation may occupy positions in the prefix, decrement, address, and, in the case of certain channel commands, even the tag portions of the binary word. The operation code of a disk or Hypertape order is translated into the first 12 bits (bits S-11) of the first binary word of the order.
3. If indirect addressing has been specified, the appropriate flag bits are inserted.
4. The expression in the first subfield of the variable field is evaluated; if the operation is Type D, this expression is evaluated as a Boolean expression. In the case of a Type D operation, this result is taken as the final binary word; any extraneous subfields are ignored, but they cause the warning flag F to be given. The first subfield of a disk or Hypertape order occupies the second 12 bits (bits 12-23) of the first binary word of the order.
5. If the operation is not Type D, and if a second subfield of the variable field is present, the expression in this subfield is evaluated, reduced modulo 8, and the resulting 3 bits are combined with the tag portion of the instruction word in a "logical or" operation. The second subfield of a disk order occupies the third 12 bits (bits 24-35) of the first binary word of the order and the first 12 bits (S-11) of the second binary word of the order.
6. If the operation is not Type D, and if a third subfield of the variable field is present, the expression in this subfield is evaluated, and the resulting 15 bits are combined with the decrement portion of the instruction word in a "logical or" operation. The third subfield of a disk order occupies the second 12 bits (12-23) of the second binary word of the order.
7. If the operation is ICC or TCM, the mask subfield is evaluated and positioned in the low order three bits of the six-bit operation code in a "logical or" operation.
8. In the case of an order, uncoded subfields are converted to 12_8.
9. The 36-bit instruction which results is assigned to the next location to be assigned by the assembler. A disk order is assigned to the next two available locations.

In general, successive instructions are assigned to successively higher storage locations; a disk order requires two consecutive locations. The assembler has a location counter to keep track of the next location to be assigned to an instruction. At the beginning of the assembly, if there is no transfer vector or linkage director, the next location to be assigned is 00000. If there is a transfer vector, its words are assigned to consecutive locations beginning with location 00000. The linkage director follows the last word of the transfer vector (see "Standard Error Procedure Option," page 15). When the first instruction is encountered, the "next location to be assigned" is the location after the last location assigned to the transfer vector and/or linkage director. The ORG and LOC pseudo-operations may be used to set the "next location to be assigned" to any desired value.

CHAPTER 8: EXTENDED MACHINE OPERATIONS

FAP provides (1) operation codes to enable the programmer to specify both select and sense instructions more conveniently, (2) additional mnemonics to more accurately describe the function of certain machine instructions, and (3) numerical prefix codes for use in forming constants or in subroutine calling sequences.

Sense Operations

The machine operations PSE (Plus Sense) and MSE (Minus Sense) are used to perform a variety of operations ranging from advancing the film on the CRT recorder to testing the status of a sense light. The specific operation performed is determined by the address portion of the binary instruction. The addresses are given in the machine manual in octal; however, FAP assumes that the number in the variable field of a PSE instruction is in decimal form. For example, the instruction that causes the film to be advanced in the CRT recorder is a PSE instruction with an octal address of 00030. This may be indicated to the assembler by converting the address to decimal as in Figure 8-1. To free the programmer from looking up octal addresses and converting them

to decimal, FAP incorporates the extended operation CFF (Change Film Frame). When this code appears in the operation field of an instruction, the appropriate operation and address bit are assembled; see the instruction in Figure 8-2. Note that the variable field of the symbolic instruction is left blank, since the entire address is implied by the operation code.

In a similar manner, the instruction which tests the status of Sense Switch 3 has the octal address 00163; see Figure 8-3. This instruction is represented in the FAP language by the operation code SWT (Sense Switch Test) and the number of the sense switch to be interrogated in the address subfield of the variable field. The SWT instruction is evaluated as follows:

1. The assembler translates the operation code SWT into the 36-bit binary word whose octal equivalent is 076000000160.
2. The expression in the address subfield of the variable field is evaluated, and the result is combined with the address portion of the instruction word in a "logical or" operation. (More than one subfield would not normally be present in the variable field of a sense operation, but, if present, they will be evaluated as tag and decrement as with a machine operation.)

The instruction which interrogates Sense Switch 3 is given in Figure 8-4.

Figure 8-5 gives the extended operation codes and octal equivalents for those PSE and MSE instructions which are not entirely defined by the operation code. The letter x indicates a digit to be specified in the variable field, and the letter n indicates either a channel designation to be specified in the operation field or its corresponding octal designation in the instruction.

Note that the 704 operation ETT will assemble in a different manner from the 709/7090 operation ETTn.

```
00012   0760 00 0 00030            PSE      24
```

Figure 8-1

```
00013   0760 00 0 00030            CFF
```

Figure 8-2

```
00014   0760 00 0 00163            PSE      115
```

Figure 8-3

```
00015   0760 00 0 00163            SWT      3
```

Figure 8-4

Operation Code	Meaning	Octal Instruction
BTTn	Beginning of Tape Test, Channel n	+0760.............n000
SLF	Turn Sense Light Off	+0760.............0140
SLN	Turn Sense Light On	+0760.............014x
SWT	Sense Switch Test	+0760.............016x
SPUn	Sense Punch, Channel n	+0760.............n34x
SPTn	Sense Printer Test Channel n	+0760.............n360
SPRn	Sense Printer, Channel n	+0760.............n36x
ETTn	End of Tape Test, Channel n	-0760.............n000
SLT	Sense Light Test	-0760.............014x
RDCn	Reset Data Channel n	+0760.............n352
CFF	Change Film Frame	+0760.............0030

Figure 8-5

Select and Related Operations

The Read Select instruction that selects tape unit A3 in the BCD mode may be obtained from the assembler by converting the octal tape address to decimal and writing the instruction in Figure 8-6.

The FAP language includes extended operations which greatly simplify the construction of Read Select and Write Select instructions. An extended instruction that selects a tape for reading or writing has a four-letter operation code in which each letter has a meaning, as follows:

1. The first letter of the operation code is R for a read select or W for a write select,
2. The second letter of the operation code is T for tape, and P for printer,
3. The third letter of the operation code is B for a binary-mode select, or D for a decimal-mode (BCD-mode) select, and
4. The fourth letter of the operation code is the data synchronizer channel letter A-H.

The number of the unit, if any, is given in the variable field. The card reader, card punch, or printer in the decimal mode may be specified by the letters CD, PU, or PR, respectively, and may appear as the second and third letters of the operation code.

Thus, a more convenient way to write a Read Select instruction addressing tape A3 in the decimal (BCD) mode is given in Figure 8-7.

Note that, since the value of the expression in the variable field is combined with the address generated by the operation code by means of a "logical or" operation, the instruction could be written as in Figure 8-8. This would not normally be done, however, and is mentioned here merely to illustrate the effect of the "logical or" operation.

Figure 8-9 contains the FAP language extended operations that perform functions related to input or output. For the sake of brevity, the list includes only the extended operations which refer to data synchronizer channel A; extended operations for other channels are formed by replacing the fourth letter of the operations by the appropriate channel letter. The letter x indicates a number to be specified in the variable field (this number may be 8, 9, or 10, since the part of the instruction which designates the tape unit number, printer hubs, etc., actually consists of four bits). Those marked * exist on the 704 with the channel designation omitted.

Additional Mnemonics

FAP provides the mnemonics given in Figure 8-10 for certain machine instructions. The mnemonics more closely describe a possible function of the instruction.

The BRA and BRN instructions are useful for unconditional switching.

Prefix Codes

In writing subroutine calling sequences, it is often necessary to specify parameters in each of the four sections of the binary word: prefix, decrement, tag, and address. The decrement, tag, and address may be specified in the variable field. (Of course, they must be given in the order: address, tag, decrement.) To enable programmers to specify the value of the prefix bits, the extended operation codes in Figure 8-11 have been included in the FAP language.

The operations in Figure 8-12 are regarded by the assembler as being identical, except for the subfields which are required.

```
00016   0762 00 0 01203          RDS      643
```

Figure 8-6

```
00017   0762 00 0 01203          RTDA     3
```

Figure 8-7

```
00020   0762 00 0 01203          RTDA     643
```

Figure 8-8

709/7090/7094 Instructions

Operation Code	Meaning	Octal Instruction
BSFA	Backspace File	-0764......120x
BSRA	Backspace Record	+0764......120x
BTTA	Beginning of Tape Test	+0760......1000
ETTA	End of Tape Test	-0760......1000*
PSLA	Present Sense Lines	+0664......0000
RCDA	Read Card Reader	+0762......1321*
RDCA	Reset Data Channel	+0760......1352
REWA	Rewind Tape	+0772......120x*
RPRA	Read Printer	+0762......1361*
RTBA	Read Tape Binary	+0762......122x*
RTDA	Read Tape Decimal	+0762......120x*
RUNA	Rewind and Unload	-0772......120x
SDHA	Set Density High	+0776......122x
SDLA	Set Density Low	+0776......120x
SPRA	Sense Printer	+0760......136x
SPTA	Sense Printer Test	+0760......1360
SPUA	Sense Punch	+0760......1340
SSLA	Store Sense Lines	+0660......0000
WEFA	Write End of File	+0770......120x*
WPBA	Write Printer Binary	+0766......1362*
WPDA	Write Printer Decimal	+0766......1361*
WPRA	Write Printer (Decimal)	+0766......1361*
WPUA	Write Punch	+0766......1341*
WTBA	Write Tape Binary	+0766......122x*
WTDA	Write Tape Decimal	+0766......120x*

704 Instructions

Operation Code	Meaning	Octal Instruction
BST	Backspace Tape	+0764......020x
IOD	Input/Output Delay	+0766......0333
RDR	Read Drum	+0762......0300
WDR	Write Drum	+0766......0300
WTS	Write Tape Simultaneously	+0766......0320
WTV	Write Cathode Ray Tube	+0766......0030

Figure 8-9

Mnemonic	Function	Assembled As	Required Field
BRA	Branch	TXL	Address
BRN	Branch No-Op	TXH	Address
ZAC	Zero Accumulator	PXD	None
ZSA	Zero Storage Address	SXA	Address
ZSD	Zero Storage Decrement	SXD	Address

Figure 8-10

Operation Code	Meaning	Octal Prefix
***	Zero	0
blank	Zero	0
...	Zero	0
PZE	Plus Zero	0
PON or ONE	Plus One	1
PTW or TWO	Plus Two	2
PTH or THREE	Plus Three	3
MZE	Minus Zero	4
FOR or FOUR	Four	4
MON	Minus One	5
FVE or FIVE	Five	5
MTW	Minus Two	6
SIX	Six	6
MTH	Minus Three	7
SVN or SEVEN	Seven	7

Figure 8-11

BRA
IOST
MTH
SEVEN
SVN
TXL

Figure 8-12

CHAPTER 9: VARIABLE-CHANNEL TAPE OPERATIONS

It is often desirable to refer to a tape unit symbolically. The instruction in Figure 9-1 defines the symbol X as an absolute symbol whose value is the octal number 3204; this number being the address of tape unit C4 in the decimal (BCD) mode (see TAPENO, page 21).

```
X TAPENO  C4
```

Figure 9-1

The programmer may then write the instructions in Figure 9-2 to write information on tape C4.

```
WTDX
RCHX     IOCOM
TCOX     *
```

Figure 9-2

If the programmer wishes to change his program to write this information on tape C6 instead of on tape C4, he may do so by changing only the TAPENO instruction that defines the symbol X and then reassembling his program.

Variable-channel tape instructions enable a programmer to change either the tape number or the channel or both, simply by changing one card and reassembling his program.

A variable-channel tape instruction is an instruction in which the channel letter of the operation code has been replaced by a one-letter symbol referring to a particular channel and tape number. The operation codes in Figure 9-3 are those which may be so used.

BTTA	TRCA
ETTA	BSFA
LCHA	BSRA
PSLA	REWA
RCHA	RTBA
RDCA	RTDA
RICA	RUNA
RSCA	SDHA
SCDA	SDLA
SCHA	WEFA
SSLA	WTBA
STCA	WTDA
TCNA	
TCOA	
TEFA	

Figure 9-3

The following restrictions must be observed:

1. A variable-channel operation code is formed by replacing the letter A by a symbol in one of the operation codes listed above.
2. The symbol must consist of a single letter of the alphabet from I through Z. It must not be one of the letters from A through H.
3. The symbol must be defined as an absolute symbol whose octal value contains a "thousands" digit (the channel number) from 1 through 10 (octal). This digit determines the channel to which the symbol refers.
4. The symbol is affected by the current heading character. If a variable-channel operation appears in a headed region, the symbol must be defined within the same region or within a similarly headed region.
5. The character $ may not be used in the channel designation of an operation code. However, it may be used to reference a symbol, defined in an alien-headed region, appearing in the variable field of an instruction. For example, RTB Y$X is allowed; RTBY$X is meaningless.

Note that the list of operations in Figure 9-3 is divided into two categories: those in the left-hand column refer to a channel but do not involve a tape number; those in the right-hand column refer to a channel and also require a tape number. An instruction containing a variable-channel tape operation is assembled as follows:

1. If the operation code is one which does not involve a tape number (that is, it is derived from a member of the left-hand column of Figure 9-3), then the instruction is assembled just as if the fourth character in the operation code were replaced by the channel-letter implied by the symbol.
2. If the operation code is one which requires a tape number (that is, it is derived from a member of the right-hand column of Figure 9-3), then the instruction is assembled as if the fourth character of the operation code were replaced by the implied channel-letter, and then the value of the symbol is combined with the address portion of the resulting binary instruction word in a "logical or" operation.
3. In either case, the contents of the variable field are evaluated and combined with the binary instruction word in a "logical or" operation.

For example, if the symbol X has been defined by any one of the instructions in Figure 9-4, then the instructions within the sets in Figure 9-5 are equivalent.

If a tape unit is to be read or written in the BCD mode, both the tape address and the select instruction must be of the "decimal" variety. If it is desired to read or write a tape in the binary mode, either the tape address or the select instruction (or both) should be of the "binary" variety. Thus, if all the select instructions in a program are variable-channel instructions of the "decimal" variety, the programmer may change any tape from BCD to binary, or vice

versa, by changing the card which defines the one-letter tape unit symbol. Variable-channel tape operations other than Read Select and Write Select have the same effect in either mode. Thus, the binary mode, specified in either the operation code or the TAPENO definition, overrides the decimal mode.

Similarly, the high density mode overrides the low density mode. For example, if the symbol X has been defined as in Figure 9-4, and the symbol Y has been defined by the instruction in Figure 9-6, then the four instructions in Figure 9-7 are equivalent.

```
X TAPENO  C4
X TAPENO  C4D
X TAPENO  C4L
```

Figure 9-4

```
                          03204       X   TAPENO  C4
00000   0766  00  0  03204                WTDX
00001   0766  00  0  03204                WTDC     4
00002   0766  00  0  03204                WTDX     4
00003   0766  00  0  03204                WRS      X
00004   0766  00  0  03204                WTD      X

00005   0541  00  0  00021                RCHX     IOCOM
00006   0541  00  0  00021                RCHC     IOCOM

00007   0062  00  0  00007                TCOX     *
00010   0062  00  0  00010                TCOC     *

00011   0772  00  0  03204                REWX
00012   0772  00  0  03204                REWC     4
00013   0772  00  0  03204                REW      X
00014   0772  00  0  03204                REWX     4

00015   0776  00  0  03204                SDN      1668
00016   0776  00  0  03204                SDN      X
00017   0776  00  0  03204                SDLX
00020   0776  00  0  03204                SDLC     4
```

Figure 9-5

```
03224          Y TAPENO   C4B
```

Figure 9-6

```
00022   0766  00  0  03224                WTDY
00023   0766  00  0  03224                WTBY
00024   0766  00  0  03224                WTBX
00025   0766  00  0  03224                WTBC     4
```

Figure 9-7

CHAPTER 10: PSEUDO-OPERATIONS

In addition to recognizing all the standard 704, 709, 7090, and 7094 machine instructions, and the 709 and 7090 extended machine instructions described in the various machine manuals, the FAP language recognizes many pseudo-operations. These pseudo-operations are described in detail in the succeeding chapters and are listed in Appendix A: "Combined Operations Table" along with a list of all the instructions allowed in FAP.

First Card Group

Certain pseudo-operations set the mode of assembly and provide the assembler with required information; these pseudo-operations constitute the first card group and must appear at the beginning of the symbolic deck. The first card group includes all list-control and mode-defining pseudo-operations and is terminated by the appearance of either a machine instruction or a symbol-defining, storage-allocating, or data-generating pseudo-operation.

The following pseudo-operations must appear only in the first card group: page title card, COUNT, ENTRY, SST. An absolute assembly must be specified in the first group by the ABS, FUL, or 9LP pseudo-operation.

In addition to list-control and update pseudo-operations, the following pseudo-operations may appear in the first card group: 704, 7090, EXTERN, HEAD, HED, IFF, MACRO (and associated macro definition), MOP, NOCRS, NULL, OPD, OPSYN, OPVFD, ORGCRS, REM, RMT, and TCD.

Previously Defined Symbols

In most cases, it is permissible to refer to a symbol either before or after that symbol is defined. The exceptions to this rule are the symbol-defining and storage-allocating pseudo-operations and the DUP and IFF pseudo-operations. A symbol that appears in the variable field of any of these pseudo-instructions, except IFF (see page 19), must have been defined by a preceding instruction. That is, the symbolic instruction card that defines the symbol must appear nearer the beginning of the symbolic deck than any symbolic instruction card in which the symbol appears in the variable field of one of the above pseudo-instructions.

If a symbol that has not been previously defined appears in the variable field of any of the symbol-defining or storage-allocating pseudo-instructions or in the variable field of a DUP pseudo-instruction, a phase error will be indicated by either the P, U, or F warning flags.

Any symbol that appears in the location field of the symbol-defining or storage-allocating pseudo-instructions will remain undefined if it has not been previously defined. If this undefined symbol appears in the variable field of another instruction, the error flag U will be given.

Phase Relocation Error

The variable field of a storage-allocating pseudo-instruction specifies the number of words of storage to be reserved. This number must be fixed at the time the program is assembled and may not depend on how the program is subsequently relocated. All of the words reserved do not have to be used by the program each time it is executed; typically, the number of words that are reserved is the maximum number which may be required for a given block of information. Hence, the expression in the variable field of a storage-allocating pseudo-instruction must be an absolute expression, that is, an expression whose value is independent of the relocation process.

If a relocatable or common expression appears in the variable field of a storage-allocating pseudo-instruction, the assembler signals a phase relocation error by the R error flag.

When an absolute assembly has been specified by the use of the ABS, FUL, or 9LP pseudo-operations, all symbols are treated as absolute symbols; therefore, a phase relocation error is impossible.

CHAPTER 11: ASSEMBLY INFORMATION PSEUDO-OPERATIONS

The COUNT Pseudo-Operation

The FAP assembly program owes some of its speed of assembly to the fact that it does not keep the computer waiting while a tape rewinds. The intermediate information produced and used during the assembly process is written on two tapes with half of the information on each, so that one of these tapes is in use while the other tape is rewinding. In order to know when half of the information has been processed, the assembler must be given an estimate of the number of cards in the symbolic deck. This estimate must be given at the beginning of the symbolic deck.

The COUNT card gives this estimate. This card must be in the first card group. The constituents of the COUNT card are as follows:

1. Blanks in the location field;
2. The operation code COUNT in the operation field; and
3. A single decimal integer, which is an estimate of the number of cards in the symbolic deck, in the variable field.

The estimated card count is neither a minimum nor a maximum, and if it is grossly inaccurate, the

only result will be wasted computer time during the assembly. If the COUNT card is missing or contains anything but a decimal integer in the variable field, the assembler will assume that the card count is 2000, and the message "CARD COUNT ESTIMATE MISSING" will appear in the pre-processor assembly listing.

The END Pseudo-Operation

The END pseudo-operation is used to signal the end of the symbolic deck. The constituents of the END pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code END in the operation field; and
3. An expression or blanks in the variable field.

The END pseudo-operation performs the following functions:

1. Any unassembled remote sequences (see "The RMT and RMT* Pseudo-Operations," page 50) are inserted into the program;
3. Any binary output waiting in the punch buffer is written out;
4. In an absolute assembly, if there is a variable field, a binary transfer card is produced whose transfer address is the value of the expression in the variable field;
5. If there is a symbol in the location field, it will be defined as the last location used by the program, one location below the program break;
6. The assembly is terminated.

The END pseudo-instruction must be the last card in the symbolic deck. If the variable field is blank, the transfer card will not be produced. An END pseudo-instruction containing a transfer address may be used only in an absolute assembly.

The IFF Pseudo-Operation

The IFF (If Following Instruction) pseudo-operation controls the assembly of the instruction immediately following the IFF pseudo-instruction. The constituents of the IFF pseudo-instruction are:

1. Blanks in the location field;
2. The operation code IFF in the operation field; and
3. An expression and two symbols, separated by commas, in the variable field.

The pseudo-operation IFF with P, A, B in the variable field provides conditional assembly of program segments according to the value of the parameters P, A, and B. P is a FAP expression, and A and B are BCD symbols. The IFF pseudo-operation causes assembly of the next instruction (and all associated ETC cards) only if:

1. P is not 0, and A is identical to B.
2. P is 0, and A is not identical to B.

P will be zero if it has not been previously defined; P will be nonzero if it is relocatable. It is not a serious restriction that IFF controls only one card (and all ETC cards), since the following card may be a macro-instruction which will expand to a sequence of any length, or it may be another IFF. IFF may be used as the last instruction in a macro definition to control assembly of the instruction following the corresponding macro call or macro-instruction. Remarks cards with an * in column 1 following an IFF will be ignored; the instruction immediately following a block of such cards will be conditionally assembled.

An IFF pseudo-instruction, and all cards under its control, will be copied onto the Update Output tape (see Chapter 27, "General Discussion").

The CALLIO macro-operation in Figure 24-1 can be used to demonstrate IFF. Since P = 0, the prototype instruction following IFF will be generated only if field A, which replaces the dummy argument ERRET, is not identical with field B, which is void. Since in the macro-instruction argument list field A is void as well as field B, the prototype instruction following IFF will not be generated.

Consider the macro-definition in Figure 11-1.

```
ADD3 MACRO    A,B,C
     CLA      A
     ADD      B
     IFF      0,C,AC
     STO      C
ADD3 END
```

Figure 11-1

Figure 11-2 illustrates how a macro-instruction could be used to store the result of an operation, whereas, Figure 11-3 illustrates how a macro-instruction can be used to leave the result in the Accumulator.

```
00000                          ADD3   X,Y,Z
00000  0500 00 0 00003         CLA    X
00001  0400 00 0 00004         ADD    Y
                               IFF    0,Z,AC
00002  0601 00 0 00005         STO    Z
```

Figure 11-2

```
00006                          ADD3   X,Y,AC
00006  0500 00 0 00003         CLA    X
00007  0400 00 0 00004         ADD    Y
                               IFF    0,AC,AC
```

Figure 11-3

CHAPTER 12: SYMBOL-DEFINING PSEUDO-OPERATIONS

With the exception of a few pseudo-operations, any operation may be used to define a symbol simply by placing the symbol to be defined in the location field. The pseudo-operations used to define symbols are BOOL, EQU and SYN, MAX and MIN, SET, and TAPENO. Also included in this group, but not actually used to define symbols, are HEAD, HED, and SST pseudo-operations.

These pseudo-operations may be used to equate two symbols, e. g. , when sections written by two different programmers must be combined. Another use of these pseudo-operations is in the definition of program parameters. If a program parameter is referred to symbolically throughout a program, this parameter may be changed by changing one card in the symbolic deck. Thus, the programmer is spared the task of searching through the program to find all the places where the parameter is used. Of course, reassembly is required to change the definition of any symbol.

The EQU and SYN Pseudo-Operations

The pseudo-operations EQU and SYN (Synonymous) are identical; hence, the discussion below applies to both. The constituents of an EQU or SYN pseudo-instruction are:

1. A symbol in the location field;
2. The operation code EQU or SYN in the operation field; and
3. An expression in the variable field.

The purpose of the EQU or SYN pseudo-instruction is to define the symbol in the location field to have the value of the expression in the variable field. The symbol will be absolute, relocatable, or common according as the expression in the variable field is absolute, relocatable, or common. All symbols that are used in the variable field of an EQU or SYN pseudo-instruction must have been previously defined (see Previously-Defined Symbols, page 18).

If the asterisk is used as an element in the variable field of an EQU or SYN pseudo-instruction to denote "the location of this instruction," the value of the element * is the next sequential location not yet assigned by the assembler. Consider the instructions in Figure 12-1. If the CLA instruction is assigned to location 00102, the symbol FSTL would be defined as a relocatable symbol (since * is always a relocatable element) whose value is 00103, and the ADD instruction would be assigned to location 00103. Figure 12-1 also illustrates the fact that the occurrence of an EQU pseudo-instruction between two instructions does not alter the sequence of locations assigned by the assembler.

```
     CLA   TMP1
FSTL EQU   *
     ADD   TMP2
```

Figure 12-1

The BOOL Pseudo-Operation

The BOOL pseudo-operation is similar to EQU, except that the defining expression is evaluated as a Boolean expression (see "The VFD Pseudo-Operation," page 28). The principal use of the BOOL pseudo-operation is to equate a symbol with an octal number. The constituents of a BOOL pseudo-instruction are:

1. A symbol in the location field;
2. The operation code BOOL in the operation field; and
3. A Boolean expression in the variable field.

The result of the BOOL pseudo-instruction is to define the symbol in the location field to be an absolute symbol having the value of the expression in the variable field. No relocatable symbol or common symbol may appear in the variable field of a BOOL pseudo-instruction, or a Boolean error will be signaled by the assembler; in this case, the error-flag B or F will appear in the left margin of the assembly listing opposite the BOOL pseudo-instruction. All symbols that are used in the variable field of a BOOL pseudo-instruction must have been previously defined (see Previously Defined Symbols, page 18).

The SET Pseudo-Operation

In order to define a symbol and yet permit it to be redefined later, the pseudo-operation SET may be used. The constituents of the SET pseudo-instruction are:

1. A symbol in the location field;
2. The operation code SET in the operation field; and
3. An expression in the variable field.

The symbol in the location field is assigned the value of the expression in the variable field. If the symbol had been previously defined by the SET pseudo-operation, it will be redefined. If the symbol had been previously defined, but not by a SET pseudo-operation, the symbol will be redefined, but a warning flag will be given. The SET pseudo-operation will override any prior means of definition. This pseudo-operation is useful for providing, within a macro-operation, a value for a parameter which is not accessible through the argument list.

If the expression in the variable field of a SET pseudo-instruction is a number, it is considered an absolute expression in the decimal mode.

The SET pseudo-operation may not be used to define a relocatable symbol.

The TAPENO Pseudo-Operation

The TAPENO pseudo-operation is used to equate a symbol with a tape address. Its primary use is with the variable-channel tape operations described in Chapter 9, "Variable-Channel Tape Operations." The constituents of the TAPENO pseudo-instruction are:

1. A symbol in the location field; this symbol may consist of from one to six characters but will usually be a single letter;
2. The operation code TAPENO in the operation field;
3. A tape unit designator in the variable field.

The variable field of the TAPENO pseudo-instruction contains a special type of expression called a "tape unit designator." The designator consists of a channel letter, followed by a tape unit number, and optionally followed by one of the following characters that may be used to set the mode in a TAPENO pseudo-instruction.

B	Binary
D	Decimal
H	High Density
L	Low Density

The purpose of the TAPENO pseudo-operation is to define the symbol in the location field to be an absolute symbol whose value is the address of a designated tape unit. The address will be that of the designated unit in the decimal low density mode unless the letter B or H is present following the tape unit number.

Figure 12-2 illustrates the TAPENO pseudo-instruction.

```
01203          T TAPENO   A3L
01203          U TAPENO   A3D
01203          V TAPENO   A3
01223          W TAPENO   A3H
01223          X TAPENO   A3B
02210          Y TAPENO   B8
03212          Z TAPENO   C10
```

Figure 12-2

The MAX Pseudo-Operation

The MAX (Maximum) pseudo-operation defines the symbol in the location field to have the value of the maximum of the expressions in the variable field. The constituents of the MAX pseudo-instruction are:

1. A symbol in the location field;
2. The operation code MAX in the operation field;
3. A series of expressions, separated by commas, in the variable field.

The MIN Pseudo-Operation

The MIN (Minimum) pseudo-operation defines the symbol in the location field to have the value of the minimum of the expressions in the variable field. The constituents of the MIN pseudo-instruction are:

1. A symbol in the location field;
2. The operation code MIN in the operation field;
3. A series of expressions, separated by commas, in the variable field.

The expressions in the variable field must be all absolute, all relocatable, or all common.

The SST Pseudo-Operation

The SST pseudo-operation causes the System Symbol Table to be included in the assembly. The constituents of the SST pseudo-instruction are:

1. Blanks in the location field;
2. The operation code SST in the operation field;
3. Blanks in the variable field.

The System Symbol Table includes definitions of the system. See Appendix C. For the FORTRAN Assembly Program, which is a part of the FORTRAN Monitor operating under IBSYS, also see "System Symbol Table" in Appendix D. If these definitions are to be used, SST must appear in the first card group; otherwise, the System Symbol Table will be made unavailable.

When FAP is used under the Basic Monitor (IBSYS), a nonblank variable field may be used with the SST pseudo-operation. The variable field may contain the symbol FORTRAN, which causes the symbols listed in both Appendix C and Appendix D to be defined, or it may contain the symbol IBSYS, which causes only the symbols listed in Appendix D to be defined. If the variable field is blank, symbol definition will be determined by the assembly mode (see Appendix D).

The HEAD Pseudo-Operation

It is sometimes desirable to combine two or more programs which use the same symbols for different purposes. The HEAD pseudo-operation makes such a combination possible, by prefixing each symbol (of five or fewer characters) by a heading character. Using different heading characters in the sections to be combined removes any ambiguity as to the definition of a symbol. References from one headed region to a differently-headed region may be made by the use of six character symbols or by the use of the character $ as described later in this section.

It is possible to multiply head symbols in a section of a program by prefixing each symbol by more than one heading factor. Such a section of the program is called a multiply-headed region.

The constituents of the HEAD pseudo-instruction are:

1. Blanks in the location field;
2. The operation code HEAD in the operation field;
3. A series of up to ten single characters (a letter or digit, but not a special character), separated by commas, in the variable field.

The character in the first subfield of the variable field of the HEAD pseudo-instruction is considered the prime heading character and will be prefixed to any symbol appearing in the variable field and location field of the instructions that follow, until a subsequent HEAD or HED is given. The heading characters in the second, third, ... subfields of the variable field are prefixed to symbols appearing in the location field of the instructions that follow, for convenience in referencing instructions in this headed region from an alien-headed region. Each symbol in the location field of a headed region is entered in the Symbolic Reference Table prefixed by each heading character. Six character symbols are not headed.

Figure 12-3 illustrates multiple heading.

```
      HEAD    A,B,C
BETA  CLA     ALPHA
```

Figure 12-3

In the multiply headed region, A0BETA, B0BETA, and C0BETA are entered in the Symbolic Reference Table and have the same definition. ALPHA is headed by A and must be defined elsewhere in the program.

To understand the operation of the heading function, it is necessary to know that every symbol is converted by FAP into a six-character symbol by the addition of zeros to the left. Thus, the pairs of symbols in Figure 12-4 are equivalent.

```
TMPX
00TMPX

Z
00000Z

FUNCT
0FUNCT
```

Figure 12-4

When the assembler encounters a symbol in a headed region, it examines the leftmost character of the symbol, and if this character is zero, the assembler replaces it with the heading character. Thus, in a region headed by the character A, the pairs of symbols in Figure 12-5 are equivalent.

```
TMPX
A0TMPX

Z
A0000Z

FUNCT
AFUNCT
```

Figure 12-5

Observe that a five-character headed symbol is equivalent to a six-character symbol. For example, the following two elements are equivalent:

COMMON
C$OMMON

From the above discussion, it should be clear that an unheaded region is the same as a region headed by the character zero. Hence, to discontinue heading, the instruction in Figure 12-6 should be used.

```
HEAD     0
```

Figure 12-6

By convention, a HEAD pseudo-instruction with a blank variable field is taken to mean heading by the character zero.

Since six-character symbols may not be headed, they may be used conveniently for reference between differently-headed regions.

In order to allow the programmer more freedom in cross-referencing, the character $ may be used to denote alien heading. For example, suppose that, in a region headed by the character B, it is desired to refer to the symbol TMPX that is located in a region headed by the character A. The instruction in Figure 12-7 will accomplish this.

```
ALPHA  AXC      A$TMPX,2
```

Figure 12-7

The rules for the use of the character $ in heading are as follows:

1. An element containing the character $ consists of a single character (a letter or digit, but not a special character), followed by the character $, followed by a single symbol.

2. Such an element is taken to refer to the symbol headed by the heading character preceding the $. In an absolute assembly, if no character precedes the $, the element refers to the symbol headed by the character zero (that is, not headed at all). The heading character specified in this way is used regardless of the heading character applied to the region in which the element appears. In a relocatable assembly, if a $ is the first character in the variable field of a machine instruction and is followed by a symbol, this symbol is unheaded and is considered to be a name in the transfer vector.

3. In order to unhead a symbol in a relocatable assembly, a zero must explicitly precede the dollar sign.

4. If an ENTRY is headed (e.g., ENTRY X$ABC), the subroutine may be referred to by means of a headed CALL (e.g., CALL X$ABC,ARGn), or by a doubly-headed machine operation code (e.g.,

CLA XABC); either will cause a proper entry into the transfer vector.

The HED Pseudo-Operation

The HED pseudo-operation is a 704 SAP pseudo-operation which has been included in the FAP language to make it easier to change 704 symbolic programs into 709/7090 symbolic programs. In FAP this pseudo-operation has been supplanted by HEAD.

The constituents of the HED pseudo-instruction are:

1. A single character or letter or digit, but not a special character, in column 1, the first column of the location field;
2. The operation code HED in the operation field; and
3. Up to nine single characters (letters or digits, but not a special character), separated by commas, in the variable field.

The effect of HED is the same as HEAD, except that the symbol in the location field is considered to be the heading character for symbols in the variable field of the following instructions, and the symbols in both the location field and the variable field of the HED pseudo-instruction are used to define symbols in the location fields of the following instructions.

A blank in column 1 of the symbolic card indicates heading by the character zero; that is, suspension of heading.

CHAPTER 13: STORAGE-ALLOCATING PSEUDO-OPERATIONS

The BES, BSS, and COMMON pseudo-operations are used to reserve blocks of core storage for data storage or for working space. The ORG and LOC pseudo-operations are used to set the program and location counters for storage assignment.

Location Counter and Program Counter

During assembly, the location counter keeps track of the next location to be assigned to an instruction; the program counter keeps track of the next location to be assigned to a symbol. The program and location counters operate in the same manner. The program counter may be set separately from the location counter by means of the LOC pseudo-operation. The ORG pseudo-operation sets both the location counter and the program counter. The load address on a binary card is taken from the location counter, whereas symbol definitions are taken from the program counter.

The BSS Pseudo-Operation

The BSS (Block Started by Symbol) pseudo-operation is used to reserve an area of core storage within a program for data storage or for working space.

The constituents of a BSS pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code BSS in the operation field; and
3. An absolute expression in the variable field.

The BSS pseudo-operation performs the following two functions:

1. A symbol in the location field is defined to have the value of the next location to be assigned by the assembler at the time the BSS pseudo-operation is encountered;
2. A block of consecutive storage locations is reserved; the number of locations reserved is the value of the expression in the variable field.

Thus, the BSS pseudo-operation reserves a block of storage whose length is given in the variable field, and if there is a symbol in the location field, this symbol refers to the first location of the block. The location of the block within the program is determined by the location of the BSS card within the program deck.

The BSS pseudo-operation causes an area to be skipped, not cleared; therefore, it may not be assumed that an area reserved by a BSS pseudo-operation contains zeros. Words of zero may be generated by DEC or OCT in such cases.

The effect of a BSS pseudo-instruction on the binary output of the assembler is to cause any binary words in the punch buffer to be written out, and to cause the next output to start with a new card origin. A BSS with a count of zero has no effect on the binary output.

Any symbols in the variable field of a BSS pseudo-instruction must have been previously defined. The expression in the variable field must be an absolute expression (see Previously-Defined Symbols, page 18, and Phase Relocation Error, page 18).

Figure 13-1 illustrates the manner in which BSS affects storage allocation.

```
00000  0 00004 0 00001   ALPHA IOCD    BETA,,4
00001                     BETA BSS     4
00005  0 00004 0 00006   GAMMA IOCD    DELTA,,4
```

Figure 13-1

The BES Pseudo-Operation

The BES (Block Ended by Symbol) pseudo-operation is used to reserve an area of core storage within a program for data storage or for working space. The constituents of the BES pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code BES in the operation field; and
3. An absolute expression in the variable field.

The BES pseudo-operation performs the following two functions:

1. A block of consecutive storage locations is reserved; the number of locations reserved is the value of the expression in the variable field;
2. A symbol in the location field is defined to have the value of the next location to be assigned by the assembler after the block has been reserved.

Thus, the BES pseudo-instruction reserves a block of storage whose length is given in the variable field; if there is a symbol in the location field, this symbol refers to the location after the last location in the block. The locations of the block within the program are determined by the location of the BES card within the program deck. BSS and BES have the same effect if their location fields are blank.

The BES pseudo-instruction causes an area to be skipped, not cleared; therefore, it may not be assumed that an area reserved by a BES pseudo-operation contains zeros. Words of zero may be generated by DEC or OCT in such cases.

The effect of a BES pseudo-instruction on the binary output of the assembler is to cause any binary words in the punch buffer to be written out, and to cause the next output to start with a new card origin. A BES with a count of zero has no effect on the binary output.

All symbols appearing in the variable field of a BES pseudo-operation must have been previously defined. The expression in the variable field must be an absolute expression, (see Previously-Defined Symbols, page 18, and Phase Relocation Error, page 18).

Figure 13-2 illustrates the manner in which BES affects storage allocation.

The COMMON Pseudo-Operation

The COMMON pseudo-operation is used to reserve an area of upper core storage for data storage or for working space. Typically, this pseudo-operation is used when two or more subprograms operate on the same block of information (see the discussion of FORTRAN COMMON usage in "Common Storage," page 59). The constituents of the COMMON pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code COMMON in the operation field;
3. An absolute expression in the variable field.

The COMMON pseudo-operation may be used only in a relocatable assembly. The COMMON pseudo-instruction operates in conjunction with a counter, called the common counter. This counter keeps track of the location of the next block of common storage to be assigned.

The COMMON pseudo-operation performs the following two functions:

1. A symbol in the location field is defined to be the current value of the common counter; and
2. The common counter is decreased by the value of the expression in the variable field.

Thus, the COMMON pseudo-operation reserves a block of storage in upper core storage. The length of the block is given in the variable field; if there is a symbol in the location field, this symbol is a common symbol which refers to the last location of the block (not the location after the last location, as with BES). This usage coincides with the FORTRAN rule that the name of an array refers to the logically first word of the array, since the first word is stored in the highest core storage location required by the array.

All symbols appearing in the variable field of a COMMON pseudo-instruction must have been previously defined. The expression in the variable field must be an absolute expression (see Previously-Defined Symbols, page 18, and Phase Relocation Error, page 18). If the COMMON pseudo-instruction is used in an absolute assembly, it will be flagged as an undefined operation; an invalid instruction will be generated with the prefix digit of the octal portion of the listing left blank.

The address portion of the fourth word of the program card that precedes the binary output will contain the address of the last piece of data assigned downward in common storage, that is, one more than the final contents of the common counter; this is the common break. However, if no COMMON pseudo-instructions occur in the program, this portion of the program card will be blank. In an assembly with COMMON, the common break will be listed as it appears on the program card.

```
00000  0 00000 4 00005   ALPHA PZE      BETA,4
00005                     BETA BES      4
00005  0 00000 4 00006   GAMMA PZE      DELTA,4
```

Figure 13-2

The ORG Pseudo-Operation

In the FAP language, the ORG (Origin) pseudo-operation is used to set the "next location to be assigned by the assembler" to a desired value. In the absence of an ORG pseudo-instruction, the assembler will assign locations beginning with 00000. The constituents of the ORG pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code ORG in the operation field; and
3. An expression in the variable field.

The ORG pseudo-operation performs the following two functions:

1. The symbol in the location field is defined to have the value of the expression in the variable field; and
2. The value of the expression in the variable field is taken by the assembler to be the next location to be assigned.

All symbols appearing in the variable field of an ORG pseudo-instruction must have been previously defined (see Previously-Defined Symbols, page 18).

The effect of an ORG on the binary output of the assembler is to cause any words in the punch buffer to be written out, and to cause the next output to start at the new card origin. This occurs even if the new origin is the location immediately following the last location used, in contrast to a BSS or BES with a count of zero.

The ORG pseudo-operation causes the next instruction to be assembled at the origin given, and, if there is a symbol in the location field, it has the value of the new origin.

```
                     01750    ALPHA ORG       1000
01750   0500 00 0 01751             CLA       BETA
01751                         BETA  BSS       1
```

Figure 13-3

```
BINARY CARD NO. TEST0000
          PROGRAM CARD

                   10000            ORG       4096

BINARY CARD NO. TEST0001
    10000   0020 00 0 40001          TRA       ALPHA
                      40001          LOC       16385
    40001   0500 00 0 40004    ALPHA CLA       BETA
    40002   0601 00 0 40005          STO       DELTA
    40003   0020 00 0 40006          TRA       GAMMA
    40004                       BETA BSS       1
    40005                      CELTA BSS       1

BINARY CARD NO. TEST0002     CARD ORIGIN   10006
    40006   0056 00  000012    GAMMA RNT        12
```

Figure 13-4

Figure 13-3 illustrates the manner in which ORG affects storage allocation.

The LOC Pseudo-Operation

The LOC (Location) pseudo-operation is used to set the program counter. The constituents of the LOC pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code LOC in the operation field; and
3. An expression in the variable field.

All symbols appearing in the variable field of a LOC pseudo-instruction must have been previously defined (see Previously Defined Symbols). The effect of a previous LOC is terminated by the appearance of an ORG pseudo-operation or by the appearance of a LOC pseudo-operation with a blank variable field.

LOC will not cause any binary words in the punch buffer to be written out.

If a portion of the object program is to be loaded at (ORG) 10001_8 but is to be executed from (LOC) 40001_8, the sequence of instructions in Figure 13-4 may be used to permit symbolic addressing.

A LOC or ORG pseudo-instruction may be used in a relocatable assembly in accordance with the following rules. If the expression in the variable field is absolute or relocatable, the new origin, and any symbol in the location field of a LOC or ORG pseudo-instruction is assumed to be relocatable above the transfer vector and the linkage director. If the expression in the variable field is common or is undefined, an assembly error will result.

The EVEN Pseudo-Operation

The EVEN pseudo-operation is used to ensure an even value of the program counter for the data or instruction that follows. It is used primarily with 7094 double-precision instructions. The constituents of the EVEN pseudo-instruction are:

1. Blanks in the location field;
2. The operation code EVEN in the operation field; and
3. Blanks in the variable field.

If the program counter is odd when an EVEN operation is given, a binary word containing the instruction AXT 0,0 is generated. Also, for relocatable assemblies, an indication is given in the program card that relocation of the program should be by an even amount and an extra AXT 0,0 is added following the transfer vector and preceding the linkage director, if present.

The load address (instruction counter) may be made odd by an EVEN pseudo-operation if LOC is in effect.

CHAPTER 14: DATA-GENERATING PSEUDO-OPERATIONS

The FAP language provides five pseudo-operations (OCT, DEC, BCI, BCD, and VFD) which may be used to introduce words of data into a program during assembly. Numbers introduced in this way are often referred to as "constants." A sixth pseudo-operation, DUP, causes a sequence of symbolic instructions to be duplicated a specified number of times. DUP is often used in conjunction with VFD to generate tables of data.

The OCT Pseudo-Operation

The OCT (Octal Data) pseudo-operation is used to generate data expressed in octal form. The constituents of the OCT pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code OCT in the operation field; and
3. One or more subfields, each containing a signed or unsigned octal integer, in the variable field.

The subfields of the variable field are separated by commas; the number of subfields permissible is limited only by the restrictions that the last subfield must be terminated by a blank, and that the entire instruction must fit on one symbolic card. If the variable field includes column 72, a terminating blank is assumed by the assembler. Of course, several OCT instructions may appear in succession.

The OCT pseudo-operation performs the following two functions:

1. If there is a symbol in the location field, this symbol is defined to be the next location to be assigned by the assembler when the OCT pseudo-instruction is encountered; and
2. Each subfield of the variable field is converted to a binary word; these words are assigned to successively higher core storage locations as the variable field is processed from left to right.

Thus, the OCT pseudo-instruction introduces data words into consecutive core storage locations, and if there is a symbol in the location field, this symbol refers to the first of these locations. Consecutive commas in the variable field cause the number zero to be generated, as does a comma followed by a blank; hence, the number of words of data generated is always one more than the number of commas in the variable field.

A subfield may contain any signed or unsigned octal integer less than 2^{35}. If any subfield of the variable field exceeds these limits, or if any character other than

+ - 0 1 2 3 4 5 6 7

appears in any subfield, the error flag B will be given.

Figure 14-1 illustrates the manner in which the OCT pseudo-instruction generates data.

The DEC Pseudo-Operation

The DEC (Decimal Data) pseudo-operation is used to generate words of data expressed as decimal numbers. DEC is identical to OCT, except that the subfields of the variable field are taken to be decimal data items (see Decimal Data Items, page 5). The constituents of the DEC pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code DEC in the operation field; and
3. One or more subfields, each containing a decimal data item, in the variable field.

The subfields of the variable field are separated by commas; the number of subfields permissible is limited only by the restrictions that the last subfield must be terminated by a blank, and that the entire instruction must fit on one symbolic card. Of course, several DEC instructions may appear in succession.

The DEC pseudo-instruction performs the following two functions:

1. If there is a symbol in the location field, this symbol is defined to be the next location to be assigned by the assembler when the DEC pseudo-operation is encountered; and
2. Each subfield of the variable field is converted to a binary word; these words are assigned to successively higher storage locations as the variable field is processed from left to right.

Thus, the DEC pseudo-instruction introduces data words into consecutive core storage locations, and if there is a symbol in the location field, this symbol refers to the first of these locations. Consecutive commas in the variable field cause the number zero to be generated, as does a comma followed by a blank. Thus, the number of words of data generated is always one more than the number of commas in the

```
00000  0 00007 0 00006   ALPHA IOCD    GAMMA,,7
00001 -377777777777      DATA  OCT     777777777777,,77,66,
00002 +000000000000
00003 +000000000077
00004 +000000000066
00005 +000000000000
00006                    GAMMA BSS     7
```

Figure 14-1

variable field.

If the variable field of a DEC instruction contains anything other than valid decimal data items, the error flag E will be given.

Figure 14-2 illustrates the manner in which the DEC pseudo-instruction generates data.

The BCI Pseudo-Operation

The BCI (Binary Coded Information) pseudo-operation is used to generate BCD data into a program. Each data word generated by this pseudo-operation consists of six 6-bit characters in the standard BCD character code (see BCD Character Code, page 68). The constituents of the BCI pseudo-operation are:

1. A symbol or blank in the location field;
2. The operation code BCI in the operation field; and
3. Two subfields in the variable field:
   a. The count subfield, which consists of a single digit, followed by a comma (a comma in column 12 specifies a count of ten);
   b. The data subfield, whose length is determined by the count subfield.

The number in the count subfield specifies the number of six-character words to be generated; the number of characters in the data subfield is the number in the count subfield multiplied by six. Since the count subfield determines the total length of the variable field, the comments field begins immediately following the end of the data subfield, and no blank character is needed to separate the comments field from the variable field.

The data subfield may contain any combination of valid characters, including comma and blank. Thus, the BCI pseudo-operation is an exception to the rule that the variable field is terminated by a blank.

The BCI pseudo-operation performs the following two functions:

1. If there is a symbol in the location field, this symbol is defined to be the next location to be assigned by the assembler when the BCI pseudo-operation is encountered; and

2. The first six characters of the data subfield are converted to BCD, and the resulting binary word is assigned to the next storage location to be assigned by the assembler. If the number in the count subfield is greater than one, the next six characters are converted and assigned to the next storage location, and so on, until the number of words specified by the count subfield have been generated.

Thus, the BCI pseudo-instruction introduces data words into consecutive core storage locations, the number of words generated being equal to the number in the count subfield. If there is a symbol in the location field, it refers to the first word of data generated.

If the count subfield is not 0-9 or a comma in column 12, one word of blanks is generated and the error flag E is given.

Figure 14-3 illustrates the manner in which the BCI pseudo-instruction generates data.

The BCD Pseudo-Operation

The BCD (Binary Coded Decimal) pseudo-operation is a 704 SAP pseudo-operation which has been included in the FAP language for compatibility. This pseudo-operation has been supplanted by BCI in FAP. The BCD pseudo-operation is like BCI, with the following exceptions:

1. The operation code BCD appears in the operation field;
2. The count digit must appear in column 12;
3. No comma separates the count digit from the data subfield; the data subfield always begins in column 13; and
4. A blank or zero in column 12 is used to indicate ten words of data.

If card column 12 does not contain a blank or the digits 0-9, one word of blanks is generated and the error flag E is given.

```
00000   0 00007 0 00007    ALPHA   IOCD    GAMMA,,7
00001  +000000000015        DATA   DEC     13,-22,5B5,1,,
00002  -000000000026
00003  +050000000000
00004  +000000000001
00005  +000000000000
00006  +000000000000
00007                      GAMMA   BSS     7
```

Figure 14-2

```
00000   0 00007 0 00003    ALPHA IOCD      GAMMA,,7
00001   222324604425       DATA   BCI      2,BCD MESSAGE COMMENT
00002   626221272560
00003                      GAMMA BSS       7
```

Figure 14-3

The VFD Pseudo-Operation

The VFD (Variable Field Definition) pseudo-operation is used primarily for the generation of tables for use with the "convert" operations. The constituents of the VFD pseudo-instruction are:

1. A symbol or blank in the location field;
2. The operation code VFD in the operation field; and
3. One or more subfields (described below) in the variable field.

Each VFD pseudo-instruction generates one or more binary words of data. Each subfield of the variable field generates one or more bits of this data; thus, the unit of information for this pseudo-operation is the single bit. Each subfield is one of three types:

Symbolic
Octal (or Boolean)
Alphameric

The constituents of a VFD subfield are:

1. The type-letter:

The letter O signifies that the subfield is octal (Boolean).

The letter H signifies that the subfield is alphameric.

The absence of a type-letter signifies that the subfield is symbolic.

2. The bit count:

An unsigned decimal integer, which specifies how many bits of the data word will be generated by this subfield.

3. The separation character / (slash).
4. The data item.

The form of the data item depends on the type of subfield:

1. In a symbolic subfield the data consists of one expression;
2. In an octal subfield the data item consists of one octal integer or one Boolean expression; and
3. In an alphameric subfield the data consists of a string of characters, none of which is a comma or a blank.

The subfields are separated by commas. Any number of subfields may be used, but the length of each subfield must be 63 bits or less.

The VFD pseudo-operation performs the following two functions:

1. If there is a symbol in the location field, this symbol is defined to be the next location to be assigned by the assembler when the VFD pseudo-instruction is encountered.

2. Successive subfields of the variable field are converted and packed to the left to form generated data words. If n is the bit count of the first subfield, then the data item in that subfield is converted to an n-bit binary number. This n-bit binary number is placed in the leftmost n bit positions of the first data word to be generated; the sign position is here regarded as the first bit position. If n exceeds 36, the leftmost 36 bits of the converted data item form the first generated data word, and the remaining bits are placed in the first n-minus-36 bit positions of the second generated data word. Each succeeding subfield is converted and placed in the leftmost bit positions remaining after the preceding subfields have been processed. The data words generated in this way are assigned to successively higher storage locations. If the total number of bit positions used is not a multiple of 36, then the unused bit positions at the right of the last generated data word will be filled with zeros.

The data item in a symbolic subfield is converted in the same manner as a symbolic expression. Let n be the bit count of the subfield. If the data item, as converted, occupies more than n bits, only the rightmost n bits of the converted data item are used. If the data item, as converted, occupies less than n bits, sufficient zero bits are placed at the left of the converted data item to form an n-bit binary number. Neither condition is regarded as an error by the assembler. If the data item is a relocatable expression or a common expression, then the subfield must be so situated in relation to preceding fields that its rightmost bit coincides with the rightmost bit of a generated data word, or with the rightmost bit of the decrement portion of a generated data word. A violation of this rule will be flagged as a relocation error by the assembler.

The data item in an octal subfield may be an unsigned octal integer of any length. If the bit count of the subfield is 36 or less, the data item may be any valid Boolean expression. Note that an unsigned octal integer is one type of valid Boolean expression. If the bit count of the subfield exceeds 36, then the data item must be an unsigned octal integer. Let n be the bit count of the subfield. If the data item, as converted, occupies more than n bits, only the rightmost n bits of the converted item are used. If the data item, as converted, occupies less than n bits, sufficient zero bits are placed at the left of the converted data item to form an n-bit binary number. Neither condition is regarded as an error by the assembler.

The data item in an alphameric subfield may consist of any combination of characters other than comma or blank. Each character is converted to its six-bit binary code equivalent. Let n be the bit count of the subfield. If the data item, as converted, occupies more than n bits, only the rightmost n bits are used. If the data item, as converted, occupies less than n bits, sufficient six-bit groups of the form 110000 (the BCD code for blank) are placed at the left of the converted data item to form an n-bit binary number; if n is not a multiple of six, the appropriate right-hand

portion of this group will appear at the extreme left of the n-bit result. In other words, the data item is converted as if the leftmost character were preceded by an unlimited number of blanks. If the bit count of the subfield is not a multiple of six, the leftmost character used, or the leftmost blank used, is truncated. None of the conditions discussed in this paragraph is regarded as an error by the assembler.

The bit count of each subfield must be 63 or less. If the bit count of a subfield exceeds 63, it will be taken as 63, and the erroneous instruction will be given the flag E.

The pseudo-operation ETC, described below, may be used to extend the variable field of a VFD pseudo-instruction. Any number of ETC pseudo-instructions may follow a VFD to give an effective variable field of unlimited length. If there is a symbol in the location field of the VFD pseudo-instruction, this symbol refers to the first generated data word.

The asterisk may be used as an element in the variable field of a VFD pseudo-instruction. When so used, the value of this element is the next location to be assigned by the assembler when the subfield containing the asterisk is about to be processed. That is, the value of the asterisk will be the location assigned to the generated data word which contains the leftmost bit of the subfield in which the asterisk appears. Failure to keep this fact in mind may lead to confusion, since the bits generated by one subfield may occupy as many as three different generated data words.

As an example, suppose the programmer would like to break up a single 36-bit word into four parts as follows:

1. Positions S, 1-9: the binary equivalent of the decimal integer 895
2. Positions 10-14: the binary equivalent of the octal integer 37
3. Positions 15-20: the binary code for the character C
4. Positions 21-35: the binary value of the symbol ALPHA

He may proceed as in Figure 14-4.

```
VFD       10/895,O5/37,H6/C,15/ALPHA
```

Figure 14-4

Figure 14-5 is an additional illustration of the VFD pseudo-instruction.

The ETC Pseudo-Operation

The ETC (Etcetera) pseudo-operation is used to extend the variable fields of the VFD and CALL pseudo-instructions and certain macro-instructions. The constituents of the ETC pseudo-instruction are:

1. Blanks in the location field;
2. The operation code ETC in the operation field; and
3. One or more subfields in the variable field.

An ETC pseudo-instruction may appear only in one of the following positions in the symbolic deck:

1. Immediately following a VFD pseudo-instruction;
2. Immediately following a CALL pseudo-instruction;
3. Immediately following another ETC pseudo-instruction;
4. Immediately following certain macro-instructions;
5. Anywhere within a macro-definition.

An additional restriction is that an ETC pseudo-instruction may not appear immediately after the last instruction in the range of a DUP.

The variable field of the VFD, CALL, or ETC pseudo-instruction preceding an ETC pseudo-instruction must contain a number of complete subfields and must be terminated by a comma followed by a blank. That is, if the variable field of a VFD or CALL instruction is divided among several symbolic cards, the divisions must take place between subfields, with the separating comma at the point of division going with the subfield that precedes it.

If a VFD, CALL, or ETC pseudo-instruction is followed by an ETC pseudo-instruction, but does not have a comma immediately preceding the blank which terminates its variable field, or an ETC occurs immediately following the last instruction in the range of a DUP or immediately following any instruction except a VFD, a CALL, or a valid ETC pseudo-instruction, then the assembler will fail to recognize the operation code, and the error flag O will be given.

Each subfield of the variable field of an ETC pseudo-

```
                    01750         ORG    1000

01750  0 00007 0 01756    ALPHA IOCD   GAMMA,,7
                           DATA VFD    1/1,17/,09/77,
01751  400000077212             ETC    H18/ABC,H18/D,
01752  223606024000             ETC    45/ALPHA,54/*
01753  000000001750
01754  000000000000
01755  001754000000
01756                     GAMMA BSS    7
```

Figure 14-5

instruction will be processed in the same way as subfields of the immediately-preceding VFD, CALL, or ETC pseudo-instruction. Thus, for example, the first instruction in Figure 14-6 is equivalent to the set of three instructions which follow it.

The ETC pseudo-instruction may also be used to extend the variable field of a macro-definition heading card, a macro-instruction, or an instruction within a macro-definition prototype. The ETC convention in the Macro-Operation Processor differs from the ETC convention to extend the variable field of a VFD, CALL, or ETC pseudo-instruction. In the Macro-Operation Processor, it is necessary that the variable field of the instruction preceding the ETC conform to the following conventions:

1. An unmatched left parenthesis exists in the variable field; or
2. The variable field is terminated by a $ immediately followed by the character blank or card column 73. This will not be confused with the use of the character $ to signal a heading character or transfer vector name, since, in neither of these cases will the $ be immediately followed by a blank; or
3. The variable field extends to card column 72.

If a card with an unmatched left parenthesis is not immediately followed by an ETC card, an assembly error will be flagged. If a card with a terminal $ is not followed by an ETC, the terminal $ is deleted from the macro-definition and ignored.

If the preceding card does not follow these conventions, an ETC card will be treated as the first card in the prototype and an assembly error will usually result.

Within the macro-definition prototype the variable field of any instruction may be extended by an ETC, following the ETC convention in the Macro-Operation Processor. However, if the variable field of a VFD or CALL is to be extended by ETC, it must follow the ETC convention for VFD or CALL.

The macro-operation compiler will generate ETC cards, recognized by the macro-operation processor only, to follow any generated instruction whose variable field extends beyond card column 72.

The DUP Pseudo-Operation

The DUP (Duplicate) pseudo-operation causes an instruction or sequence of instructions to be duplicated. Its primary use is in the generation of tables. The constituents of the DUP pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code DUP in the operation field; and
3. Two subfields, the instruction count and the iteration count, in that order, in the variable field. Each subfield contains one symbolic expression.

The DUP pseudo-instruction performs the following functions:

1. If there is a symbol in the location field, this symbol is the next location to be assigned by the assembler when the DUP pseudo-operation is encountered.
2. The instruction count and iteration count subfields are evaluated.
3. Duplication is performed, as described below, under control of the instruction count and iteration count.

All symbols that appear in the variable field of a DUP pseudo-instruction must have been previously defined.

Let the letter m stand for the instruction count and the letter n for the iteration count. Then the meaning of the DUP pseudo-instruction is "Duplicate the next m instruction n times." The set of m instructions immediately following the DUP pseudo-instruction is called the range of the DUP. The effect of the DUP pseudo-instruction is as if the set of m symbolic instructions making up the range of the DUP were copied n-1 times (except for the location fields), and these n-1 copies placed in the symbolic deck behind the original set.

The duplication process consists of n iterations. During the first iteration, the instructions in the range of the DUP are assembled normally, just as if the DUP had not occurred. Symbols that are defined within the range of the DUP are defined during the first iteration. Each subsequent iteration is performed by assembling all of the instructions of the range in order, but without defining any symbols except a symbol defined by the SET pseudo-operation. The number of binary words generated by the duplication process is n times the number of words generated by the instructions in the range of the DUP.
appear in the range of a DUP except a macro-instruction and the pseudo-operations ABS, COMMON, COUNT, DUP, END, ENTRY, FUL, IRP, IFF, LOC, MAC, ORG, RMT, SST, and 9LP. The update pseudo-instructions may not appear in the range of a DUP.

The asterisk may be used as an element within the range of a DUP, in which case the value of this

```
00000  677772300002        DATA  VFD   10/895,05/37,H6/C,15/ALPHA
                                 VFD   10/895,
                                 ETC   05/37,H6/C,
00001  677772300002              ETC   15/ALPHA
```

Figure 14-6

element differs during different iterations of the DUP. This provides a very powerful method for generating tables.

Figures 14-7, 14-8, and 14-9 are equivalent; the first two figures illustrate how the asterisk may be used to reduce the amount of coding required.

```
T1 DUP      1,10
   VFD      20/**100-T1*100,16/T1
```

Figure 14-7

```
   DUP      1,10
T1 VFD      20/**100-T1*100,16/T1
```

Figure 14-8

```
T1 VFD      20/0,16/T1
   VFD      20/100,16/T1
   VFD      20/200,16/T1
   VFD      20/300,16/T1
   VFD      20/400,16/T1
   VFD      20/500,16/T1
   VFD      20/600,16/T1
   VFD      20/700,16/T1
   VFD      20/800,16/T1
   VFD      20/900,16/T1
```

Figure 14-9

The uses of DUP in Figure 14-10 are meaningless; they will be given the warning flag F and will be ignored by the assembler.

```
DUP     0,N
DUP     M,0
DUP     M,1
```

Figure 14-10

CHAPTER 15: PROGRAM-LINKING PSEUDO-OPERATIONS

The ENTRY, CALL, IFEOF, and EXTERN pseudo-operations are used within a program to provide communication links between that program and other programs. The character $ may also be used for this purpose. The descriptions that follow assume that the reader is familiar with the use of the program card and transfer vector in FORTRAN; a thorough discussion of these and related subjects appears in Part V of this manual.

The ENTRY Pseudo-Operation

The ENTRY pseudo-operation is used to define an entry point to a relocatable subprogram. A main program is distinguished by the fact that it contains no ENTRY pseudo-instructions or that it contains an ENTRY with an explicit zero in the variable field. The ENTRY pseudo-operation must appear in the first card group. The constituents of the ENTRY pseudo-instruction are:

1. Blanks in the location field;
2. The operation code ENTRY in the operation field; and
3. A single symbol in the variable field.

The symbol in the variable field that must be defined subsequently as a relocatable symbol is a name of the subprogram. The first character of the subprogram name may not be numeric.

The ENTRY pseudo-instruction performs the following two functions:

1. The symbol in the variable field (followed by sufficient blanks to make six characters) is placed in the program card. An explicit zero in the variable field of an ENTRY pseudo-instruction will cause the program card to indicate that the entry point to the main program is the first instruction following the transfer vector and the linkage director.
2. The value of the symbol, which is defined in the program, is placed in the program card following the symbol itself.

Thus, the ENTRY pseudo-instruction establishes the symbol as a name of the program and identifies the associated entry point with it. There may be more than one ENTRY in a subprogram.

For example, a subroutine to compute sines and cosines begins as in Figure 15-1.

The ENTRY pseudo-instruction may also be used to provide secondary entries for subroutines. If the symbol in the variable field of the ENTRY instruction is preceded by a minus sign, the word on the program card which contains the address of this entry point will have a 1 in the sign position. This will cause the loader to ignore the subroutine unless one of its primary entries has also been called. This feature

```
                                 SINE-COSINE ROUTINE
                      00001             ENTRY    SIN
                      00000             ENTRY    COS
00000  0300 00 0 00072      COS         FAD      =1.57079632679
00001  0601 00 0 00071      SIN         STO      ARG
```

Figure 15-1

of FAP is useful when assembling subroutines for inclusion in the library tape.

The ENTRY pseudo-operation is undefined in an absolute assembly.

The CALL Pseudo-Operation

The CALL pseudo-operation is used to produce a subroutine calling sequence of the type generated by the CALL statement in FORTRAN (see Part 5 of this manual). The constituents of the CALL pseudo-instruction are:

1. A symbol or blanks in the location field;
2. The operation code CALL in the operation field; and
3. One or more subfields in the variable field:
   a. The first subfield of the variable field must contain a single symbol that is the name of a subroutine.
   b. Each subsequent subfield (if any) may contain any symbolic expression; these are the arguments of the subroutine.

The subfields are separated by commas; the number of subfields permissible is unlimited, since the ETC pseudo-operation may be used to extend the variable field to any desired length.

The CALL pseudo-instruction performs the following functions in a relocatable assembly:

1. If there is a symbol in the location field, this symbol is the next location to be assigned by the assembler when the CALL pseudo-instruction is encountered.
2. The first subfield of the variable field contains the name of the subroutine called.
   a. If this name is not already present in the transfer vector, it is placed there (followed by sufficient blanks to make six characters), and the name is defined to be a relocatable symbol whose value is the corresponding location in the transfer vector.
   b. A TSX instruction, having a tag of 4 and an address that is the transfer vector location containing the subroutine name, is assembled and assigned to the next location to be assigned by the assembler.
3. Each subsequent subfield of the variable field contains an argument and is assembled as the address of a TSX instruction whose tag is zero. These TSX instructions are assigned to successively higher locations.
4. A CALL pseudo-instruction (or any valid ETC following it) followed by a comma and then a blank, but not followed by an ETC instruction, will generate an additional argument of TSX 0, corresponding to the void field.
5. If there is a symbol in the location field, this symbol refers to the first instruction of the calling sequence.

Caution must be observed when using constants in a calling sequence to a FORTRAN subprogram. A FORTRAN subprogram always regards a calling-sequence argument as the address of the location where the operand is stored. Thus, if it is necessary to communicate the number 3 to a FORTRAN subprogram as an integer, the argument in the CALL pseudo-instruction must be a symbol assigned to a location whose decrement contains the number 3. Note that certain FAP-coded subprograms, notably DUMP, are written to accept either the operand or its address in the calling sequence.

If a CALL pseudo-instruction is used in an absolute assembly, no transfer vector entry will be made, and the name of the subprogram in the first subfield of the variable field must be defined in the same manner as any other symbol.

Figure 15-2 illustrates the assembly of a CALL pseudo-instruction.

Standard Error Procedure Option

The standard error procedure provides information which will enable an error-tracing routine to tabulate the sequence of subroutine calls which led to a given error. The use of the standard error procedure is optional. When used with a FORTRAN program, the error-tracing routine will give the name of the subprogram in which the error occurred, the name of the higher-level subprogram which called it, the external and internal formula numbers of the FORTRAN statement which called the error-producing subprogram, the name of the still-higher-level subprogram which called the higher-level subprogram, and so on, back to a statement in the main program. The standard error procedure in FAP will make it

```
 TRANSFER VECTOR
00000  246444476060      DUMP

00001  0074 00 4 00000             CALL     DUMP,A,A+100,,R,S,3
00002  0074 00 0 77461
00003  0074 00 0 77625
00004  0074 00 0 00000
00005  0074 00 0 77460
00006  0074 00 0 77457
00007  0074 00 0 00003
```

Figure 15-2

possible for the error-tracing routine to give similar information about FAP-assembled programs, and to continue tracing through FAP and FORTRAN programs. Instead of providing the error-tracing routine with external and internal formula numbers, the standard error procedure in FAP gives the octal location of the calling sequence involved.

The standard error procedure will add two binary words to the beginning of each assembled subprogram. These two words will be introduced immediately following the last word of the transfer vector, or at the very beginning if the subprogram has no transfer vector. The first of these words is called the linkage director, because the information it contains when an error occurs will enable the error-tracing routine to find the statement which called the subprogram. Initially the linkage director will contain the number zero. The subprogram should store index register 4 in the decrement of the linkage director every time the subprogram is entered. Note that FAP does not automatically produce the necessary SXD instructions to save index register 4. If the first location of the program proper is assigned a location symbol, then the address of the linkage director may be obtained by subtracting 2 from the symbol.

Immediately following the linkage director in each subprogram, the standard error procedure will introduce a word containing the BCD name of the subprogram; this name is given in the variable field of the first entry instruction. The error-tracing routine will refer to this location to find the name of the subprogram. A symbolic subprogram using the standard error procedure might begin as in Figure 15-3. In this case the linkage director would occupy SYMB-2; SYMB-1 would contain the number 627044226060, which is the BCD equivalent of SYMB.

```
*SYMBOLIC   SUBPROGRAM
       ENTRY    SYMB
       ENTRY    SUBP
SYMB   FAD      =3.14159
SUBP   SXD      SYMB-2,4
       STO      SYMT
```

Figure 15-3

The standard error procedure will lengthen each calling sequence produced by the CALL pseudo-instruction by introducing the two instructions in Figure 15-4 at the end of the calling sequence:

```
NTR     *+2,0,A
PZE     C,0,B
```

Figure 15-4

where A and B together give the octal location, relative to the beginning of the program, of the first word of the calling sequence, and C gives the location of the linkage director in a subprogram. In a main program C is zero. If the location of the first word of the calling sequence is less than 32770_8, relative to the beginning of the program, then A will be zero, and B will be a binary number which, when converted to decimal, will give the correct octal location; otherwise, A will contain the high-order octal digit of the location, and B, when converted to decimal, will give the low-order four octal digits.

An installation desiring to use the standard error detection procedure may do so by removing the card labeled 9F04FLOW from the Editor Deck; any installation not wanting the facility may leave the Editor Deck intact to omit the additional assembled instructions. Subroutine references made by use of the character $ will not be affected.

Subroutine Reference Using the $ Character

In FAP, preceding a symbol that appears in the variable field of a machine instruction by the character $ has the effect of defining that symbol as the name of a subroutine. When such a symbol is encountered, FAP will do the following:

1. If the symbol is not already present in the transfer vector, the symbol is unheaded, is placed in the transfer vector (followed by sufficient blanks to make six characters), and the symbol is defined to be a relocatable symbol whose value is the corresponding location in the transfer vector.
2. The instruction is assembled in the normal manner, as if the character $ were not present.

Since a symbol need be so identified only once in a program, the use of the $ is necessary only when the subroutine name does not appear as the first subfield of the variable field of a CALL instruction, and then it is necessary to prefix the character $ to just one appearance of the subroutine name. No harm is done, however, if the $ is used more than once with the same symbol.

The use of the $ for subroutine reference is subject to the following restrictions:

1. In an absolute assembly, the character $ will not cause a transfer vector entry to be made.
2. No symbol that has been defined as the name of a subroutine (either by use of the $ or by a CALL pseudo-operation) may be used in the variable field

of any pseudo-instruction except CALL. If the character $ is used in the variable field of any other pseudo-instruction, it will be considered a heading character. If a symbol which has been defined as a subroutine name appears in the variable field of a symbol-defining or storage-allocating pseudo-instruction, the instruction will be given the error flag P.

3. When the character $ is used to define a subroutine name, this character must be the first character of the variable field. That is, the character $ may be used only in the address subfield of an instruction. If the subroutine name is established in the transfer vector by its appearance elsewhere in the program (as the first subfield of the variable field of a CALL pseudo-operation, or, preceded by a $, as an address of a machine operation), then expressions involving the subroutine name, but not including the character $, may be used in the address, tag, or decrement subfield of an instruction.

4. A subroutine name may be headed if the heading character is not the first character in the variable field of the instruction; see Figure 15-5.

Suppose several tables have been assembled as subroutines to avoid having to reassemble the tables each time the program is reassembled. The "table" subroutine might begin as in Figure 15-6. The program that uses these tables might contain the sequence of instructions in Figure 15-7.

The IFEOF Pseudo-Operation

The IFEOF (If End of File) pseudo-operation is used to communicate with a library subroutine (EOF), which must be provided by the individual installation.

Figure 15-8 illustrates the assembly of the IFEOF pseudo-instruction.

The IFEOF pseudo-operation is undefined in an absolute assembly.

```
TSX      $ALPHA,4        ALPHA IS A SUBROUTINE NAME
TSX      A$BETA,4        A$BETA IS NOT DEFINED HERE
                         AS A SUBROUTINE NAME
TSX      $B$PSI,4        B$PSI IS A SUBROUTINE NAME
TSX      $0$PSI,4        PSI IS A SUBROUTINE NAME
```

Figure 15-5

```
        COUNT    200
TABLES FOR  CONVERSION
        ENTRY    TBLP
        ENTRY    TBLQ
        ENTRY    TBLR
        ENTRY    TBLS
        ENTRY    TBLT
TBLP    CAQ      A,,6
A       DUP      1,10
        VFD      20/**100000-A*100000,16/B
B       DUP      1,10
        VFD      20/**10000-B*10000,16/C
C       DUP      1,10
        VFD      20/**1000-C*1000,16/D
D       DUP      1,10
        VFD      20/**100-D*100,16/E
E       DUP      1,10
        VFD      20/**10-E*10,16/F
F       DUP      1,10
        VFD      20/*-F
TBLQ    CVR      G,,6
G       DUP      1,10
        VFD      6/*-G,30/G
        DUP      1,10
        VFD      6/*-G-10,30/GH
```

Figure 15-6

```
LDQ      BCDWD
CLM
XEC*     TBLP
ARS      16
SLW      BINWD
```

Figure 15-7

```
TRANSFER VECTOR
C0000  742546263460       (EOF)

C0001 -0625 60 0 00000         A IFEOF    B
C0002  0761 00 0 00003
```

Figure 15-8

The EXTERN Pseudo-Operation

The EXTERN (External Symbol) pseudo-operation is used to insert symbols into the transfer vector. The constituents of the EXTERN pseudo-instruction are:

1. Blanks in the location field;
2. The operation code EXTERN in the operation field; and
3. A list of FAP symbols, of one through six BCD characters, separated by commas, in the variable field.

Each symbol in the EXTERN variable field will be inserted into the transfer vector. If a symbol is already in the transfer vector, it will be ignored. If a symbol has been previously defined in the location field of a machine instruction or a symbol defining pseudo-operation, it will be multiply defined. EXTERN is undefined in an absolute assembly.

CHAPTER 16: OPERATION CODE-DEFINING PSEUDO-OPERATIONS

In setting the mode of an assembly, the 704 and 7090 pseudo-operations define which operation codes in the Combined Operations Table are to be allowed in a program that is to be executed on an IBM 704 or 7090, respectively. The OPD, OPVFD, and OPSYN pseudo-operations define and rename operation codes. These pseudo-operations may appear in the first card group.

The 704 Pseudo-Operation

The 704 pseudo-operation sets the mode of assembly to 704. Instructions unique to 7090 are given the warning flag 9. The constituents of the 704 pseudo-instruction are:

1. Blanks in the location field;
2. The operation code 704 in the operation field; and
3. Blanks in the variable field.

The 7090 Pseudo-Operation

The 7090 pseudo-operation sets the mode of assembly to 7090. Instructions unique to the 704 are given the warning flag 4. Except for the 709 drum instructions, which are in the 704 mode, all 709 instructions are included in the 7090 mode. The constituents of the 7090 pseudo-instruction are:

1. Blanks in the location field;
2. The operation code 7090 in the operation field; and
3. Blanks in the variable field.

7090 is the initial mode of assembly, that is, it is automatically in effect unless the 704 pseudo-instruction is given.

The OPD Pseudo-Operation

The OPD (Operation Definition) pseudo-operation defines a machine operation code. The constituents of the OPD pseudo-instruction are:

1. A symbol in the location field;
2. The operation code OPD in the operation field; and
3. An octal machine operation code definition (see below) in the variable field.

The OPD pseudo-instruction assembles the variable field as an octal number and assigns this number as the machine operation code definition of the symbol in the location field (see "Machine Operation Code Definition," page 36).

The OPVFD Pseudo-Operation

The OPVFD (Operation Variable Field Definition) pseudo-operation defines a machine operation code. The constituents of the OPVFD pseudo-instruction are:

1. A symbol in the location field;
2. The operation code OPVFD in the operation field; and
3. One or more subfields as described for the VFD pseudo-instruction (see page 28), with a bit count of exactly 36, in the variable field.

The OPVFD pseudo-instruction assembles the variable field as an octal number and assigns this number as the machine operation code definition of the symbol in the location field (see Machine Operation Code Definition below).

The OPSYN Pseudo-Operation

The OPSYN (Operation Synonym) pseudo-operation renames machine operation codes and pseudo-operations. The constituents of the OPSYN pseudo-

instruction are:

1. A symbol in the location field;
2. The operation code OPSYN in the operation field; and
3. A machine operation code, which may have been defined by a prior OPD, OPVFD, or OPSYN, in the variable field.

The OPSYN pseudo-instruction obtains from the Combined Operations Table the octal number to be used for definition of the operation code symbol in the location field.

Machine Operation Code Definition

In order to establish an operation code word for OPD or OPVFD, an octal number must be created in accordance with the table in Figure 16-1.

Position	Meaning
S	Sign
1, 2	Type A operation code
3-11	Type B, C, D, or E operation code
12, 13	Indirect address permitted (for Type B operation codes only)
14	Address required
15	Tag required
16	Decrement required
17	Low-order thirteen bits may contain flags, not a portion of the operation code (Type E or Type B I/O instruction)
18	Indirect address permitted (Type A instruction)
19	Non-transmit bit (Type A instruction)
20	Instruction is machine instruction, not pseudo-instruction (bit automatically provided for OPD or OPVFD)
21	Instruction permitted in 704 mode
22	Instruction permitted in 7090 mode. Note: An instruction must have either or both of the above bits or an N warning flag will appear when the instruction is used.
23-35	Part of operation code if bit 17 is zero
33	Type K disk command
34	Type C instruction or type K disk command with a low order (mask) field
35	Type D instruction

Figure 16-1

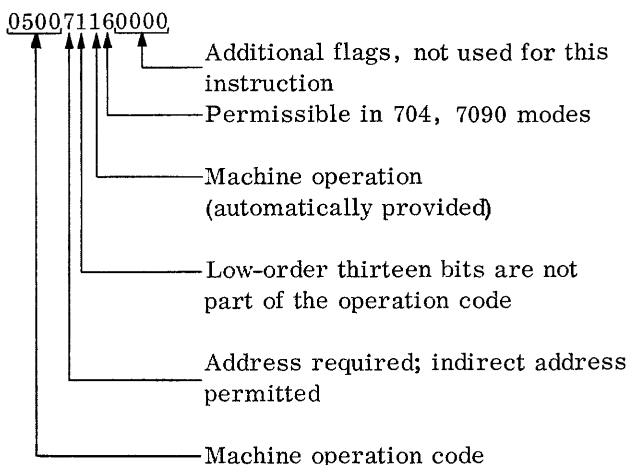
In order to define XYZ as an operation that is synonymous with CLA, it is possible to write either of the instructions in Figure 16-2.

```
XYZ  OPSYN    CLA

XYZ  OPD      050071160000
```

Figure 16-2

The variable field of the second instruction in Figure 16-2 may be analyzed as follows:



CHAPTER 17: CARD FORMAT-CONTROL PSEUDO-OPERATIONS

In addition to assembling relocatable programs for use within the Monitor system, the FAP assembler will also assemble absolute programs. The deck of binary cards produced by an absolute FAP assembly may be loaded and executed with or without the use of the Monitor system. The pseudo-operations used to specify card formats are ABS, FUL, TCD, END, and 9LP. If ABS, FUL, or 9LP are to be used to establish the mode of assembly, the pseudo-operation must appear in the first card group. If an ENTRY pseudo-instruction appears before an ABS, FUL, or 9LP pseudo-instruction, indicating a relocatable assembly, the error flag O will be given for all appearances of ABS, FUL, 9LP, or TCD.

In an absolute assembly:

1. The following pseudo-operations may not be used (and will be regarded as undefined operations): COMMON, ENTRY, IFEOF, and EXTERN.
2. The following pseudo-operations, not otherwise permissible, may be used: 9LP, FUL, ABS, and TCD.
3. A variable field is permitted in an END pseudo-operation.
4. The character $ may not be used for subroutine reference, but it may be used for heading reference (see the description of HEAD, page 21).
5. All symbols (and hence all expressions) are taken as absolute symbols; therefore, relocation errors and phase relocation errors are impossible.
6. Binary output format will depend on the Monitor control cards used. Both row and column binary output, on- and off-line, will be in 22 instructions-per-card format.
7. Binary output will be produced even if errors are detected by the assembler.

The ABS Pseudo-Operation

The ABS (Absolute) pseudo-operation is used to specify an absolute assembly and, within an absolute assembly, to discontinue the FUL or 9LP mode of binary output. The constituents of the ABS pseudo-instruction are:

1. Blanks in the location field;
2. The operation code ABS in the operation field; and
3. Blanks in the variable field.

The ABS pseudo-operation may be used within an absolute assembly, except in the first card group, to cause discontinuance of the FUL or 9LP mode of binary output. When so used, the ABS pseudo-instruction will appear somewhere after the appearance of a FUL or 9LP pseudo-instruction. It effects the binary output by causing any words in the punch buffer to be written out and the next output to start on a new binary card in the appropriate absolute mode of punching. The next location assigned by the assembler becomes the new card origin. If an ABS card is encountered in an absolute assembly when the FUL or 9LP mode is not in force, it is ignored.

The FUL Pseudo-Operation

The FUL pseudo-operation is used to specify absolute assembly and to cause binary output to be in the 24-words-per-card FUL mode (see Chapter 19 for a description of this mode). The constituents of the FUL pseudo-instruction are:

1. Blanks in the location field;
2. The operation code FUL in the operation field; and
3. Blanks in the variable field.

When the FUL pseudo-operation appears within an absolute assembly, any words remaining in the punch buffer are written out and the next output is started at the beginning of a FUL card. Binary output will thereafter be in the FUL mode until the end of the assembly or until an ABS or 9LP pseudo-instruction is encountered.

The 9LP Pseudo-Operation

The 9LP (9 Left Prefix) pseudo-operation specifies an absolute assembly and causes a prefix punch in the 9-left word of an absolute binary card. The constituents of the 9LP pseudo-instruction are:

1. Blanks in the location field;
2. The operation code 9LP in the operation field; and
3. An expression in the variable field.

When 9LP appears within an absolute assembly, any words remaining in the punch buffer are written out and the next output is started at the beginning of a 9LP card. The prefix punch in the first word is based on the low-order three binary digits of the expression in the variable field. In other respects the binary output is identical to that produced by ABS (see above). Binary output will thereafter be in the 9LP mode until the end of the assembly or until an ABS or FUL is encountered.

The TCD Pseudo-Operation

In an absolute program, a binary transfer card directs the loading program to stop loading cards and to transfer control to a designated location. In most cases, a transfer card is required only at the end of the binary deck; in absolute assemblies, the END pseudo-operation may cause a binary transfer card to be punched (see the description of END, page 19). However, it is occasionally desirable to cause a transfer card to be punched before the end of the binary deck; in this case, the TCD pseudo-operation is used.

The constituents of the TCD pseudo-instruction are:

1. Blanks in the location field;
2. The operation code TCD in the operation field; and
3. A symbolic expression in the variable field.

The TCD pseudo-operation performs the following functions:

1. Any binary output waiting in the punch buffer is written out.
2. A binary transfer card is produced whose transfer address is the value of the expression in the variable field.

CHAPTER 18: LIST-CONTROL PSEUDO-OPERATIONS

The list-control pseudo-operations affect the assembly listing but have no effect whatever on the binary program produced by the assembler.

The REM Pseudo-Operation

The REM (Remarks) pseudo-operation is used to enter remarks into the assembly listing. The constituents of the REM pseudo-instruction are:

1. Blanks in the location field;
2. The operation code REM in the operation field; and
3. Remarks in the variable field, starting after column 11.

The contents of columns 8-10 of the operation field will be replaced by blanks, and the contents of the remainder of the card will be copied onto the assembly listing. Otherwise, the remarks card is ignored by the assembler.

The SPACE Pseudo-Operation

The SPACE pseudo-operation is used to generate one or more blank lines onto the assembly listing. The constituents of the SPACE pseudo-instruction are:

1. Blanks in the location field;
2. The operation code SPACE in the operation field; and
3. A symbolic expression in the variable field.

The value of the expression in the variable field is the number of blank lines which will appear in the assembly listing; however, if the value of the expression is zero, one blank line will appear. Also, if the spacing operation would result in the next line being printed within five lines of the bottom of the page, no spacing will occur; instead, the next line of the listing will appear at the top of a new page. Listing of the SPACE pseudo-instruction is controlled by PCC.

The EJECT Pseudo-Operation

The EJECT pseudo-operation causes the next line of the listing to appear at the top of a new page. The constituents of the EJECT pseudo-instruction are:

1. Blanks in the location field;
2. The operation code EJECT in the operation field; and
3. Blanks in the variable field.

Listing of the EJECT pseudo-instruction is controlled by PCC.

The UNLIST Pseudo-Operation

The UNLIST pseudo-operation causes all listing, except instructions that are flagged, to be suspended. The constituents of the UNLIST pseudo-instruction are:

1. Blanks in the location field;
2. The operation code UNLIST in the operation field; and
3. Blanks or comments after column 17.

The UNLIST pseudo-instruction itself is listed (unless a previous UNLIST is in effect), but thereafter only those instructions that are flagged will be listed by the assembler until a LIST pseudo-instruction is encountered. List-control pseudo-operations, other than LIST, will be ignored by the assembler when UNLIST is in effect.

The LIST Pseudo-Operation

The LIST pseudo-operation causes listing that was previously suspended by an UNLIST pseudo-instruction to be resumed. The constituents of the LIST pseudo-instruction are:

1. Blanks in the location field;
2. The operation code LIST in the operation field; and
3. Blanks in the variable field.

Listing of the LIST pseudo-instruction is controlled by PCC. The LIST pseudo-instruction will cause one blank line on the assembly listing whether or not UNLIST is in effect.

The TITLE Pseudo-Operation

Most symbolic instructions generate one binary word; some pseudo-operations generate no binary words; the generative pseudo-operations generate several binary words. The generative pseudo-operations are OCT, DEC, BCI, BCD, DUP, CALL, ETC, VFD, and IFEOF. Initially the assembly listing will contain all the binary words generated by the generative pseudo-operations.

The TITLE pseudo-operation causes the assembly listing to be abbreviated by eliminating all but the first word generated by a generative pseudo-operation. In the case of DUP, iterations after the first are eliminated from the listing. The constituents of the TITLE pseudo-instruction are:

1. Blanks in the location field;
2. The operation code TITLE in the operation field; and
3. Blanks in the variable field.

Listing of the TITLE pseudo-instruction is controlled by PCC.

Following the appearance of TITLE pseudo-instruction and until the appearance of a DETAIL pseudo-instruction, the assembler will exclude from the assembly listing any line that meets the following conditions: the line contains octal information and does not contain the symbolic instruction that generated the octal information, unless this line has been flagged by the assembler.

If TITLE is in effect at the end of the assembly, the literals will not be listed.

The DETAIL Pseudo-Operation

The DETAIL pseudo-operation causes the listing of generated data, that was previously suspended by a TITLE pseudo-instruction, to be resumed. The constituents of the DETAIL pseudo-instruction are:

1. Blanks in the location field;
2. The operation code DETAIL in the operation field; and
3. Blanks in the variable field.

The sole effect of the DETAIL pseudo-operation is to cancel the effect of a previous TITLE pseudo-instruction. If TITLE is not in effect, the DETAIL pseudo-instruction is ignored by the assembler.

Listing of the DETAIL pseudo-instruction is controlled by PCC.

The LBL Pseudo-Operation

The LBL (Label) pseudo-operation causes binary cards to be serialized in card columns 73-80. The constituents of the LBL pseudo-instruction are:

1. Blanks in the location field;
2. The operation code LBL in the operation field;
3. Two subfields, separated by commas, in the variable field: the first subfield contains up to eight alphabetic and numeric characters; the second subfield may contain any nonzero, nonblank character, or the number one.

Serialization begins with the characters appearing in the first subfield of the variable field; the characters are left-justified and filled with terminating zeros. Serialization is incremented until the rightmost non-numeric character is reached, at which time the numeric portion recycles to zero. This subfield may not contain any special characters.

If the variable field of a LBL pseudo-instruction is blank, serialization of binary cards will be discontinued. In order to serialize from 0, an explicit zero must appear in the variable field.

A transfer card will be labeled TRAn, where n is a five octal digit transfer address.

A nonblank, nonzero second subfield of the variable field, if it exists, will cause serialization to be listed. If this second subfield consists of the number one, only the first use of the label will be listed.

If LBL with listing is requested in a relocatable assembly with no ENTRY cards, the message PROGRAM CARD will identify the card so serialized.

If LBL with listing is included for a card governed by LOC, the actual card origin (which may differ from the location on the listing) will be listed. Listing of the LBL pseudo-instruction is controlled by PCC. LBL takes precedence over LABEL (see "Label and Serialization (FORTRAN Monitor)," (Chapter 19).

The PCC Pseudo-Operation

The PCC (Print Control Card) pseudo-operation causes the following pseudo-operations to be listed: COUNT, DETAIL, EJECT, IFF (and cards deleted by IFF), INDEX, IRP, LBL, LIST, NOCRS, ORGCRS, PMC, REF, SPACE, TITLE, and TTL. The constituents of the PCC pseudo-instruction are:

1. Blanks in the location field;
2. The operation code PCC in the operation field;
3. Blanks, ON, or OFF in the variable field.

If any field of the pseudo-operations, whose listing is controlled by PCC, is in error and is flagged, the pseudo-operation will always be listed. PCC will always be listed. Alternate appearances of PCC turn this feature on and off. The initial mode, prior to the appearance of PCC, is off.

The use of ON or OFF in the variable field gives absolute control of the PCC pseudo-operation.

The REF Pseudo-Operation

The REF (Reference Table) pseudo-operation causes the deletion of the Symbolic Reference Table listing. The constituents of the REF pseudo-instruction are:

1. Blanks in the location field;
2. The operation code in the operation field; and
3. Blanks in the variable field.

Multiply-defined and undefined symbols are always listed. REF may occur at any point in the program. Listing of the REF pseudo-instruction is controlled by PCC.

The TTL Pseudo-Operation

The TTL (Subtitle) pseudo-operation generates a subtitle on the listing. The constituents of the TTL pseudo-instruction are:

1. Blanks or a decimal integer in the location field;
2. The operation code TTL in the operation field; and
3. A string of alphameric characters starting in card column 12.

Card columns 11-72 are used in words 4-14 of a subtitle which will appear on each page. The subtitle may be overwritten by another TTL. The subtitle may be deleted by TTL 0.

A decimal integer (from 1-32,767) in the location field will cause a renumbering of pages beginning with that integer. Listing of the TTL pseudo-instruction is controlled by PCC.

The INDEX Pseudo-Operation

The INDEX pseudo-operation is used to list a table of contents of important locations within the assembly. The constituents of the INDEX pseudo-instruction are:

1. Blanks in the location field;
2. The operation code INDEX in the operation field; and
3. A list of FAP symbols, separated by commas, in the variable field.

The first appearance of an INDEX card will cause the message

TABLE OF CONTENTS

to be listed. Each subfield of an INDEX pseudo-instruction will cause the symbol, and its definition, to be listed. The listing of the INDEX pseudo-instruction is controlled by PCC.

INDEX pseudo-instructions may appear anywhere in the source program and need not be grouped. The listing generated by INDEX pseudo-instructions will be inserted where the pseudo-instructions appear. However, the message TABLE OF CONTENTS will appear only once, and, for the most meaningful commentary, INDEX pseudo-instructions should be grouped at the beginning of the source program,

interspersed with appropriate remarks cards.

The PMC Pseudo-Operation

The PMC (Print Macro-Generated Cards) pseudo-operation causes the card images generated by macro-instructions or remote sequences to be listed. The constituents of the PMC pseudo-instruction are:

1. Blanks in the location field;
2. The operation code PMC in the operation field; and
3. Blanks, ON, or OFF in the variable field.

Alternate appearances of PMC turn this feature on and off. The use of ON or OFF in the variable field gives absolute control of the PMC pseudo-operation. Initially, instructions generated through the use of a macro-instruction are not listed, except for those instructions which are flagged by the assembler. In order to cause such instructions and their octal equivalent to be listed, the PMC pseudo-operation can be used.

Control cards generated by macro-instructions are listed only if both PMC and PCC modes are ON. The nesting level of macro-generated cards (see page 46) appears in card columns 81 through 84 of the listing. If, at the end of the assembly, the PMC mode is off and remote sequences exist, literals will not be listed.

The NULL Pseudo-Operation

The NULL pseudo-operation is used to cause an instruction to be listed in full, but has no effect upon assembly.

The constituents of the NULL pseudo-instruction are:

1. Any BCD characters in the location field;
2. The operation code NULL in the operation field;
3. Any BCD characters in the variable field.

A possible use of the NULL pseudo-instruction is given in Figure 18-1 where CODE is a machine operation or pseudo-operation which is not defined in the assembler, the effect of which may be omitted from the assembly.

```
CODE OPSYN    NULL
```

Figure 18-1

The instruction in Figure 18-2 will enable a proper absolute assembly of a subprogram, with ENTRY pseudo-instructions listed, but not affecting the program.

```
ENTRY OPSYN    NULL
```

Figure 18-2

CHAPTER 19: BINARY OUTPUT FROM THE ASSEMBLER

The FAP assembler produces several different forms of binary output, depending upon whether an assembly is relocatable or absolute, and upon whether Monitor control cards appeared before the symbolic deck.

Relocatable Output

In a relocatable assembly (that is, a non-absolute assembly), the binary output form is the same as that produced by the FORTRAN compiler. The binary deck consists of a program card followed by relocatable cards containing the program. If there are more than ten entry names, additional program cards will be punched. Either row binary cards or column binary cards may be specified for either on- or off-line punching. The format of the row binary output is exactly that described in the publication, IBM 7090/7094 Programming Systems, FORTRAN II Operations, Form C28-6066-4. The column binary output is the columnar image of the row binary card format, with the addition of 7-9 punches in column 1; that is, the 9-left word of the row form occupies columns 1-3, the 9-right word occupies columns 4-6, the 8-left word occupies columns 7-9, etc.

The FORTRAN column binary transfer card (12-7-9 punches in column 1, the remainder of the card blank) is not used when operating in the Monitor system. Therefore, FAP produces no transfer card for a relocatable assembly when column binary cards are specified. When on-line row binary cards are specified for the relocatable assembly of a main program, FAP will produce a FORTRAN row binary transfer card (9 punch in column 1, the remainder of the card blank) as the last card of the binary output. If an error is detected during a relocatable assembly, no binary output will be produced.

Absolute Output

In an absolute assembly, binary output is produced even if errors are detected. The binary output is in one of six card formats, depending upon whether the output is in row binary or column binary form, and upon whether 9LP, ABS, or FUL mode is in force. Row binary output is in the standard 704 row binary card format described in the publication, IBM 7090/7094 Programming Systems, FORTRAN II Operations, Form C28-6066-4. The column binary output is the columnar image of the row binary card format, with the addition of 7-9 punches in column 1; that is, the 9-left word of the row form occupies columns 1-3, the 9-right word occupies columns 4-6, etc.

Full Output

When the full mode has been established in an absolute assembly by the use of a FUL pseudo-instruction, binary output will be in one of two formats, depending upon whether row binary or column binary output has been specified. The first word of output occupies columns 1-36 of the 9-row, the second word occupies columns 37-72 of the 9-row, the third word occupies columns 1-36 of the 8-row, and so on, to a maximum of 24 words per card. No control words or check sums are produced by the assembler in full mode.

Column binary full cards contain the first word of output in columns 1-3, the second in columns 4-6, and so on, to a maximum of 24 words per card. No control words or check sums are produced by the assembler in the full mode. There will be 7-9 punches in column 1 only if the programmer arranges for the first word on the card to contain 1's in bit-positions 9 and 11. If these bits are missing, they will not be supplied by the assembler in the "full" mode.

9LP Output

When the 9LP mode has been established in an absolute assembly by the use of a 9LP pseudo-instruction, binary output will be the same as absolute output, with the following addition: on a row binary card, columns 1, 2, and 3 of row 9 are punched; on a column binary card, column 1 of rows 12, 11, and 0 is punched with the appropriate prefix bits.

Label and Serialization (FORTRAN Monitor)

The off-line binary cards may be labeled and serialized by using the FORTRAN Monitor Control Card LABEL. The information in card columns 2-7 of the FAP page title card (see Chapter 3) will appear as the label. This label, with blanks treated as zero, is then placed in columns 73-78 of the binary cards, with columns 79 and 80 used for serialization.

Serialization begins with 00 and recycles when 99 is reached. If, however, the label does not use all of columns 73-78, serialization begins with zero and increases to 99...9, filling all remaining columns through column 80 before it recycles. LBL (see "The LBL Pseudo-Operation," in Chapter 18) takes precedence over LABEL.

FORTRAN Monitor Control Cards

The following FORTRAN Monitor Control Cards affect the FAP assembler: DATE; CARDS ROW; CARDS COLUMN; ROW; PACK; PRINT; and LABEL.

CHAPTER 20: GENERAL DESCRIPTION

A FAP macro-operation is a type of pseudo-operation created by the programmer. The most significant property of an instruction specifying a macro-operation, a macro-instruction, is that it can generate no, one, or more card images. The contents of the generated instructions are virtually unrestricted and may include any machine operation, any pseudo-operation not restricted to the first card group (e.g., COUNT, ENTRY), any macro-operation, and any field permitted on a FAP source card.

A macro-instruction can be regarded as an abbreviation for a sequence of instructions. The sequence of instructions generated by the macro-instruction is determined by the particular macro-definition corresponding to the macro-instruction code. Each macro-operation has its own definition, which consists of a heading card, a sequence of prototype instructions, and an END card.

A prototype instruction has a standard FAP symbolic card format with location field, operation code, and variable field. Remarks may appear on a prototype card; however, the remarks will not normally appear following the variable field on the macro-generated instruction. The fields of a prototype card may consist of the following: text, which will be reproduced as written; substitutable arguments; and (special) punctuation characters, which delimit arguments and, like text, are reproduced.

A field or subfield is text if it is longer than six characters or if it is a string of one through six characters, delimited by punctuation marks, which does not appear in the argument list of the macro-definition heading card. A field or subfield is a substitutable argument if it is a string of one through six characters which appears in the argument list of the macro-definition heading card.

A macro-generated sequence of instructions consists of each of the prototype instructions, with text and punctuation characters as on the prototype, but with substitutable arguments replaced by specific argument strings (of unrestricted length) whose position in the argument list of the macro-instruction corresponds to that in the argument list of the macro-definition heading card.

As an example of a programmer macro-operation, suppose that the programmer has written a source program that includes the sequences of instructions in Figure 20-1.

The pattern of these instructions might be designated by some BCD name, say QSUM, which could then be defined as in Figure 20-2.

```
•••
CLA       FEDTAX
ADD       STATAX
STO       TOTTAX
•••
CLA       XSUB1
ADD       YSUB1
STO       ZSUB1
•••
CLA       PART1
ADD       PART2
STO       TOTAL
•••
```

Figure 20-1

```
QSUM   MACRO    V1,V2,V3
       CLA      V1
       ADD      V2
       STO      V3
QSUM   END
```

Figure 20-2

The sequence of five source cards in Figure 20-2 generates no binary words in the object program, but constitutes the definition of the macro-operation which the programmer has chosen to call QSUM. The first card is the macro-definition heading card. It includes the name of the macro-operation in the location field and the argument list in the variable field. The next three cards are the prototype, in which V1, V2, and V3 are substitutable arguments identified in the heading card argument list. All of the other fields, CLA, ADD, STO, are text, since they do not appear in the argument list. The fifth card (END) marks the end of the range of the macro-definition. It will not terminate assembly.

Once the macro-operation QSUM has been defined in the source program, the sequence of CLA, ADD, STO, instructions need no longer be written. These instructions may be replaced by a macro-instruction card which generates them with their substitutable arguments replaced by the arguments in the variable field of the macro-instruction QSUM as in Figure 20-3.

```
        •••
        QSUM     FEDTAX,STATAX,TOTTAX
        •••
SUBCOM  QSUM     XSUB1,YSUB1,ZSUB1
        •••
        QSUM     PART1,PART2,TOTAL
        •••
```

Figure 20-3

In Figure 20-3, note the following points:

1. The string QSUM that appears in the location field of the pseudo-operation MACRO is not a symbol, but a code for the macro-operation being defined, and as such is entered into the Combined Operations Table. It may be the same as a location symbol appearing anywhere in the assembly, including symbols within the macro-definition.

2. The substitutable arguments V1, V2, and V3, which appear in the variable field of the macro-definition heading card and also in various fields within the prototype, merely characterize the order of the expressions and character strings that may appear in the variable field of a later macro-instruction which uses the given macro-operation. If the order on the macro-operation heading card were changed to V3, V1, V2, then the macro-instruction in Figure 20-4 would generate the same instruction.

```
QSUM     TOTTAX,FEDTAX,STATAX
```

Figure 20-4

Because the substitutable arguments are dummy names, they may be identical to strings used as symbols or operation codes for this or any other macro-operation appearing in location, operation, or variable fields in the program. The programmer should exercise caution in constructing the prototype so that the text will not be confused with substitutable arguments, since every string of six or fewer characters in any field is compared with the argument list. Special care should be taken with alphameric text, or with fields of VFD, DEC, or OCT pseudo-operations.

3. The macro-operation QSUM may be considered an open subroutine, since the generated instructions are inserted into the program at each appearance of the macro-operation QSUM.

In the simple example of the macro-definition given above, the substitutable arguments appeared in address fields in the prototype and were replaced by symbols on the macro-generated cards. In general, substitutable arguments may appear in the location field, the operation field, in any of the subfields of the variable field, or as a heading character in any subfield. The substitutable arguments may be replaced by any valid FAP expression or appropriate alphameric character strings.

If a substitutable argument appears in an operation field, it may be a string of one through six characters; however, the code which replaces it must be a standard FAP operation code of three through six characters. Consider Figure 20-5. In this macro-definition, mnemonic character strings have been chosen to represent the substitutable arguments. Notice that LOOP appears in a location field, that OP appears in an operation field, and that COEFF and DEG appear as symbols within expressions in address subfields. Notice also that GAMMA is text, i.e., a symbol, and not a substitutable dummy argument, and must be defined elsewhere in the program. Any use of the code QPOLY in a macro-instruction should be accompanied by an argument list of appropriate substitutions for the substitutable arguments. In particular, LOOP should be replaced by a symbol, which should not be multiply-defined, and OP should be replaced by a valid operation code.

```
QPOLY MACRO    COEFF,LOOP,DEG,T,OP
      AXT      DEG,T
      LDQ      COEFF
 LOOP FMP      GAMMA
      OP       COEFF+DEG+1,T
      XCA
      TIX      LOOP,T,1
QPOLY END
```

Figure 20-5

A QPOLY macro-instruction would assemble as in Figure 20-6. The symbol X015 is defined as the location in which the first instruction (AXT) appears; each of the substitutable arguments is replaced by the corresponding argument in the macro-instruction argument list. The expression arising from the prototype address COEFF+DEG+1 is equivalent to C1+2.

```
00000                          X015  QPOLY   C1-4,FIRST,5,4,FAD
00000  0774 00 4 00005               AXT     5,4
00001  0560 00 0 00002               LDQ     C1-4
00002  0260 00 0 00007   FIRST       FMP     GAMMA
00003  0300 00 4 00010               FAD     C1-4+5+1,4
00004  0131 00 0 00000               XCA
00005  2 00001 4 00002               TIX     FIRST,4,1
```

Figure 20-6

The use of the macro-operation processor permits simulating the machine instructions of another computer, or extending the machine operation vocabulary of the 709/7090.

Figure 20-7 illustrates how STO can be modified to dump the information that is stored at each execution of STO.

```
STO OPSYN    STO
STO MACRO    A
    .STO     A
    SXA      *+2,4
    TSX      DUMP,4
    AXT      **,4
STO END
```

Figure 20-7

In this case, the flag M will be given to the macro-definition heading cards (STO MACRO A) to indicate redefinition of an existing operation code.

CHAPTER 21: MACRO-DEFINITION HEADING CARD

The MACRO Pseudo-Operation

The MACRO pseudo-operation is used in the macro-definition heading card to name a macro-operation and to identify the arguments in the prototype that follows. The constituents of the MACRO pseudo-instruction are:

1. A FAP symbol of 3-6 characters (not all zeros) in the location field;
2. The operation code MACRO in the operation field; and
3. A list of substitutable arguments in the variable field.

The character string in the location field is the macro-operation code and not a location symbol. If it is the same as any other machine operation, pseudo-operation, or macro-operation, the pseudo-operation will be flagged, the code will be redefined within the Combined Operations Table, and the former definition will be lost.

The substitutable arguments in the variable field of the macro-definition heading card argument list may be any valid FAP symbols, or may consist of all numeric characters (excluding all zeros). The substitutable arguments in a macro-definition may be separated by any one of the following punctuation characters:

= + - * / ( ) $, '

The argument list is terminated by the character blank. After a punctuation character, succeeding punctuation characters or an explicit zero are ignored and do not result in a substitutable argument of zero.

Meaningful notation may be used in the argument list of a macro-definition heading card. For example, the two macro-definition heading cards in Figure 21-1 are identical.

```
ALPHA MACRO    A(B+C)-D$E

ALPHA MACRO    A,B,C,D,E
```

Figure 21-1

Extending the Argument List

The argument list of a macro-definition heading card may be extended by the use of the ETC pseudo-operation. See page 29 for a description of the ETC convention in the Macro-Operation Processor.

Alternative Format of MACRO

An alternative format of the MACRO pseudo-instruction has the following constituents:

1. Blanks in the location field;
2. The operation code MACRO in the operation field; and
3. Blanks in the variable field.

This pseudo-instruction must be immediately followed by an instruction with these constituents:

1. A FAP symbol in the location field;
2. A FAP symbol of 3-6 characters, which is the macro-operation code in the operation field; and
3. A list of substitutable arguments in the variable field.

The symbol that appears in the location field of the second instruction is a substitutable argument and not a location symbol. It is replaced by the corresponding argument in the location field of the macro-instruction card. If it appears in the variable field of an instruction, it must be defined elsewhere in the program.

The MOP Pseudo-Operation

The MOP pseudo-operation is also used in the macro-definition heading card to name a macro-operation. The constituents of the MOP pseudo-instruction are:

1. Blanks in the location field;
2. The operation code MOP in the operation field; and
3. The macro-operation code, followed by the argument list, in the variable field.

MOP is identical to MACRO except that the macro-operation code being defined appears as the first subfield in the variable field, followed by a punctuation character and the argument list.

The prototype consists of a sequence of FAP instructions that use the substitutable arguments listed in the variable field of the preceding macro-definition heading card. The prototype must be followed by an END card with the macro-operation code in its location field or variable field, or with both the location field and the variable field blank.

To lend greater flexibility to macro-operations, parentheses and the apostrophe have been included in the list of special characters that may be used within the various fields of the macro-definitions. However, neither character may be used as part of a substitutable argument.

Remarks cards, with an asterisk in column 1, appearing within a macro-definition, will not appear in the expansions.

Heading characters in effect within the region in which the macro-definition appears do not apply to the definition.

The Location Field of a Prototype Instruction

A substitutable argument or any FAP symbol may appear in the location field of a prototype instruction.

The Operation Field of a Prototype Instruction

The operation field of a prototype instruction may contain any substitutable argument, any machine operation, any pseudo-operation not restricted to the first card group, or any macro-operation code.

The Variable Field of a Prototype Instruction

A blank that is encountered before card column 72 on a prototype card (except the cards BCD, BCI, REM, TTL, and those instructions whose variable fields begin with an equal sign) is considered to terminate the variable field, and any information or commentary to the right of the blank will not be included in the macro-definition. If the blank follows an unmatched left parenthesis, however, the blank does not terminate the scan; blanks within parentheses are considered to be punctuation. If a matching right parenthesis is not encountered before card column 73, an ETC card must follow. Unmatched parentheses cause an assembly error.

Alphameric cards are scanned in full for substitutable arguments. If the variable field of either a BCD card (beginning in card column 12) or a BCI card (beginning in card column 12-16) begins with a non-blank, non-numeric character, the first subfield should be a substitutable argument for which a count will be substituted in the macro-instruction argument list.

The apostrophe (8-4 punch) may be used to concatenate (link) partial subfields in the operation field or in the variable field. It is possible to create a single subfield from a combination of arguments and text, since the apostrophe delimits an argument in the macro-definition prototype, but is itself not included in the macro-definition. The character apostrophe cannot be used to concatenate subfields of lower level nested macro-definitions. For example, to delimit the count on a BCD prototype card, the coding in Figure 22-1 may be used. Figure 22-2 illustrates how a BETA macro-instruction will assemble.

```
BETA MACRO   A,B,C
     BCD A'0  'B' ERROR. CONDITION'C' IGNORED.
BETA END
```

Figure 22-1

```
00000                          BETA    7,FIELD,,
00000  006060263125            BCD 70  FIELD ERROR. CONDITION IGNORED.
00001  432460255151
00002  465133602346
00003  452431633146
00004  456031274546
00005  512524336060
00006  606060606060
```

Figure 22-2

```
                   01226          J TAPENO  A6B
                   02201          K TAPENO  B1L

                           NAME4 MACRO    A,B,C,D,E,F,G
                           A     B'T'C'D  E
                                 SD'F     G
                           NAME4 END

C0000                            NAME4    AX,W,D,J,,L,**
C0000  0766 00 0 01226     AX    WTDJ                                          .001
C0001  0776 00 0 00200           SDL      **                                   .001

C0002                            NAME4    AY,R,B,,5,N,K
C0002  0762 00 0 00225     AY    RTB      5                                    .001
C0003  0776 00 0 02201           SDN      K                                    .001
```

Figure 22-3

Figure 22-3 illustrates the macro-generation of an operation code by concatenation.

If parentheses are to be used as part of text, the enclosed string must not appear in the macro-definition heading argument list, or the string will be considered a substitutable argument.

If a heading character of the form A$B is required, either A or B, or both may be substitutable arguments.

If a transfer vector name of the form $NAME is required, NAME may be a substitutable argument.

If a literal of the form =A is required, A may be a substitutable argument for which a valid form of a literal must be substituted.

The variable field of any card in the prototype may be extended by the use of the ETC pseudo-operation. In order that a following ETC card be recognized, it is necessary to follow the conventions stated for extending a variable field in the Macro-Operation Processor (page 30). The programmer should exercise caution so that a macro-generated card image which overflows column 72 has a variable field which is properly extended by an ETC card. Such variable fields are limited to those of a macro-instruction card, or a nested macro-definition heading card.

The macro-operation compiler will generate ETC cards, recognized by the macro-operation processor only, to follow any generated instruction whose variable field overflows card column 72.

CHAPTER 23: NESTING MACRO-DEFINITIONS

Macro-definitions may be nested by including a macro-definition heading card within a macro-definition prototype. A macro-instruction using the outermost code will generate the nested macro-definition heading cards and prototypes of each of the macro-operations nested one level below, with all outer level arguments properly substituted. Lower level macro-operations will not be defined until all higher level macro-operations within which they are nested have been expanded. Thus the expansion of a nested macro-definition may be affected by the use of SET to redefine an argument.

If macro-definitions are nested, the ends of the lower level prototypes must be marked with END cards bearing the name of the macro-operation in the location field or the variable field. If no name appears in either field of the END card, the outermost macro-definition is terminated.

An example of nested macro-definitions is given in Figure 23-1.

```
NEST1 MACRO    A,B,C
NEST2 MACRO    A,D,E
NEST3 MACRO    B,D,F
      ...
      ...
NEST3 END
NEST2 END
NEST1 END
```

Figure 23-1

The prototype of a macro-definition may include macro-instructions, the macro-operations of which have not yet been defined; however, such lower level macro-operations must be defined prior to an appearance of the higher level macro-instruction. Circular definitions that will result in a loop within the macro-operation processor must be avoided by the programmer.

CHAPTER 24: MACRO-INSTRUCTIONS

A macro-instruction is used to generate, in line, the sequence of instructions given by the prototype, with substitutions for the arguments. The constituents of a macro-instruction are:

1. A FAP symbol in the location field;
2. A previously defined macro-operation in the operation field; and

3. A list of FAP symbols, expressions, alphameric character strings, or operation codes in the variable field.

The symbol in the location field of the macro-instruction will be defined as the location of the next instruction. A macro-instruction should not appear within the range of a DUP.

The variable field contains the specific arguments to be substituted for the substitutable arguments in the prototype.

The MAC Pseudo-Operation

The MAC pseudo-operation may be used for a macro-instruction. The constituents of the MAC pseudo-instruction are:

1. A symbol in the location field;
2. The operation code MAC in the operation field; and
3. The name of the macro-operation, followed by the argument list, in the variable field.

The only difference between the MAC pseudo-instruction and a macro-instruction is that the macro-operation code appears in the first subfield of the variable field in the MAC pseudo-instruction, whereas the macro-operation code appears in the operation field of a macro-instruction.

In a MAC pseudo-instruction, a comma or left parenthesis must separate the operation code from the argument list.

Punctuation in Macro-Instructions

Only commas and parentheses may be used to separate arguments in the macro-instruction argument list. A single comma following a right parenthesis, or a single comma preceding a left parenthesis, is redundant and may be omitted. Consecutive commas define a null argument string; an explicit zero, if desired, must appear in the argument list. A blank not within parentheses terminates the argument list. Parentheses surrounding a string of characters in a macro-instruction argument string signify that everything within the parentheses is to be substituted for the corresponding argument in the macro-definition prototype. Within such parentheses, nested parentheses, commas, and blanks are considered to be part of the argument string to be substituted. If a matching right parenthesis is not encountered before card column 73, an ETC card must follow. Unmatched parentheses cause an assembly error.

Parentheses that are to be included as part of an argument string to be substituted in a prototype must be enclosed within an outer pair of parentheses that will be deleted in the macro-instruction expansion.

Figure 24-1 illustrates the definition of the CALLIO macro-operation and an example of its use.

Note that TAPE should not be a substitutable argument; that (RBEP) must be enclosed in an outer pair of parentheses; and that, in the macro-instruction argument list, an explicit null argument bounded by a comma appears corresponding in position to the substitutable argument ERRET in the macro-definition argument list. This will cause the fourth word of the calling sequence to be omitted (see IFF on page 19).

Since the character $ does not delimit an argument in the macro-instruction argument list, it may be used freely to indicate a heading character or a transfer vector symbol that is to replace a substitutable argument. If the character $ is at the end of the argument list, a comma must be used to distinguish this from the character used to flag a following ETC card. Since the character = does not delimit an argument in the macro-instruction argument list, it may be used freely to indicate a literal that is to replace a substitutable argument.

Argument Strings

The specific argument strings to be substituted must be given in the same order in the macro-instruction argument list as the substitutable arguments that appear in the macro-definition heading argument list.

```
                                 CALLIO MACRO    IOCOM,T1,OP,LABEL,T2,UNIT,PFX,ERRET
                                        TSX      (TAPE),4
                                        PZE      IOCOM,T1,OP
                                        PZE      LABEL,T2,UNIT
                                        IFF      0,ERRET
                                        PFX      ERRET
                                 CALLIO END

00000                                   CALLIO   CITIO,2,((RBEP)),CITLB,,CITTAP,,,
00000  0074 00 4 73406                  TSX      (TAPE),4                               .001
00001  0 40004 2 01204                  PZE      CITIO,2,(RBEP)                         .001
00002  0 01205 0 01203                  PZE      CITLB,,CITTAP                          .001
                                        IFF      0,                                     .001
```

Figure 24-1

It is not necessary to restrict to six characters the length of an argument string to be substituted into a location field or to seven characters the length of an argument string to be substituted into an operation field. An entire card image may be inserted into any field. No blank will be inserted following a location field longer than six characters, and the operation field, if any, will follow immediately. See Figure 24-2.

Nested Macro-Instructions

It is possible to nest macro-instructions by including either a macro-operation code or a substitutable argument that will be replaced by a macro-operation within an operation field in the prototype. See Figure 24-3.

The null macro-operation XXX will cause the generation of no instructions.

The assembler will assume that a comma or a left parenthesis immediately following the operation code, as early as card column 11, is the end of the operation field or the beginning of the variable field, respectively. Hence, meaningful notation, such as that given in Figure 24-4, may be used as a string to replace a substitutable argument.

The argument list and subargument list (see page 50) of a macro-instruction may be extended by the use of the ETC pseudo-operation. In order that a following ETC pseudo-instruction be recognized, it is necessary that the preceding instruction follow the conventions stated for extending a macro-definition heading card argument list (page 44).

```
COS(ALPHA)
```

Figure 24-4

CHAPTER 25: THE GENERATED INSTRUCTIONS

The generated instructions are similar to the prototype instructions, except that the substitutable arguments in the prototype will be replaced with the argument appearing in the macro-instruction argument list. The heading characters in effect within the region in which the macro-instruction appears will prefix all symbols of less than six characters in the location and variable fields.

In the 709/7090 mode, the macro-compiler normally generates the variable field beginning in card column 16; in the 704 mode, it normally begins in card column 12. Exceptions to this rule are the pseudo-operations BCD, BCI, REM, and TTL, whose variable fields always begin in card column 12. When the variable field of a prototype card is separated from the operation field by a comma or a left parenthesis, the fields of the generated card are separated in the same way.

```
                                           NAME9 MACRO    XXX
                                                 XXX      REMARK
                                           NAME9 END
00004                                            NAME9    (CLA      B       )
00004    0500 00 0 77461                         CLA      B        REMARK
                   77461                       B COMMON   1
```

Figure 24-2

```
                                      XXX MACRO
                                      XXX END

                                      COS MACRO     OP
                                          OP
                                          TSX       $COS,4
                                      COS END

C0005                                     COS(COS(XXX))
C0005                                     COS(XXX)
                                          XXX
C0005    0074 00 4 00000                  TSX       $COS,4
C0006    0074 00 4 00000                  TSX       $COS,4
C0007                                     COS(XXX)
                                          XXX
C0007    0074 00 4 00000                  TSX       $COS,4
                                          XXX
```

Figure 24-3

Created Symbols

If arguments are missing from the end of the argument list of a macro-instruction, symbols will be created to fill the vacancies. These symbols take the form of ..001, ..002, to ..nnn, throughout the program. An explicitly null argument terminated by a comma will be treated as null; created symbols will be supplied only at the end of the argument string.

For example, given the macro-definition heading card and the macro-instruction card in Figure 25-1, each appearance of the substitutable argument A will be replaced by X; each appearance of the substitutable argument B will be omitted, since the argument is explicitly void; and each appearance of the substitutable arguments C and D will be replaced by the symbols ..nnn that are created to replace the omitted arguments at the end.

```
ALPHA MACRO     A,B,C,D
      ALPHA     X,,
```

Figure 25-1

If more than 9999 symbols are to be created, the programmer must define a new origin for created symbols using the ORGCRS pseudo-operation, or the assembly will be terminated.

The NOCRS Pseudo-Operation

The NOCRS (No Created Symbols) pseudo-operation is used to suppress the creation of symbols that replace specific argument strings missing from the end of a macro-instruction argument list. The constituents of the NOCRS pseudo-instruction are:

1. Blanks in the location field;
2. The operation code NOCRS in the operation field; and
3. Blanks in the variable field.

The ORGCRS Pseudo-Operation

In order to alter the form of created symbols, the ORGCRS (Origin Created Symbols) pseudo-operation may be used. This pseudo-operation also reinstates the creation of symbols, if they have been suppressed by NOCRS. The constituents of the ORGCRS pseudo-instruction are:

1. Blanks in the location field;
2. The operation code ORGCRS in the operation field; and
3. Blanks or one BCD character, followed by three digits, in the variable field.

The BCD character in the variable field will replace the second dot (e.g., .Annn); the digits, if any, will be the origin of a new set of created symbols. This origin will be one number lower than the first symbol actually created. If the BCD character is desired, the three digits must also be stated explicitly; if it is not desired, the three digits are sufficient.

In the macro-definition in Figure 25-2, the transfer address S and the storage address R must be unique for each appearance of the macro-operation in a macro-instruction. However, neither address is required outside of the resulting expansion. Hence, the assembler may be permitted to assign a location symbol by omitting the corresponding arguments in the macro-instruction argument list. The ORGCRS pseudo-instruction may be used to change the format of the created symbol to .Nnnn as in Figure 25-3.

```
MNO MACRO   D,A,R,S
    CAL     D
    TZE     S
    STA     R
    ALS     18
    ORA     R
  S SLW     A
    RMT
  R BSS     1
    RMT
MNO END
```

Figure 25-2

```
                                      ORGCRS  N150
00000                                 MNO     M,H
00000 -0500 00 0 77461                CAL     M                        .001
00001  0100 00 0 00005                TZE     .N152                    .001
00002  0621 00 0 00006                STA     .N151                    .001
00003  0767 00 0 00022                ALS     18                       .001
00004 -0501 00 0 00006                ORA     .N151                    .001
00005  0602 00 0 77460      .N152     SLW     H                        .001
                                      RMT                              .001
                            .N151     BSS     1                        .001
                                      RMT                              .001
```

Figure 25-3

The pseudo-operation BSS will be assembled later in the program (see "The RMT and RMT* Pseudo-Operations" below) as in Figure 25-4.

```
00006                          RMT     *
00006                  .N151   BSS     1
```

Figure 25-4

CHAPTER 26: ADDITIONAL PSEUDO-OPERATIONS

The IRP Pseudo-Operation

The IRP (Iterate within Prototype) pseudo-operation is used within a prototype to iterate a series of instructions within the set of generated instructions. The constituents of the IRP pseudo-instruction are:

1. Blanks in the location field;
2. The operation code IRP in the operation field; and
3. A symbol in the variable field.

The symbol in the variable field must be the name of a single substitutable argument appearing in the macro-definition argument list. An IRP pseudo-instruction must precede the instructions to be iterated, and another IRP pseudo-instruction, with blank location and variable fields, must follow the instructions. Both IRP cards must be within the range of the prototype.

The argument to be substituted may be any legal macro-instruction argument, including an explicitly null argument. However, it will normally be a series of subarguments enclosed in parentheses. The subarguments are punctuated according to the same rules for punctuating a macro-instruction argument list. The number of these subargument strings will be the number of iterations of the enclosed instructions, and each iteration will be made with the corresponding subargument string substituted for the dummy argument. If no argument was given in the variable field of the first IRP, no iterations will be made; one argument causes one iteration, two arguments cause two iterations, etc.

For example, to compute the sum of squares, the SUMSQ macro-definition in Figure 26-1 can be written. The four instructions between IRPs are to be iterated. Figure 26-2 illustrates the computation $A=X^2+Y^2+Z^2$, using the SUMSQ macro-operation.

An IRP pseudo-instruction cannot occur explicitly within the range of an IRP; the first nested IRP will cause the termination of the range, and the second nested IRP will cause the reopening of another range. However, a macro-instruction within the range of IRP pseudo-instructions may itself cause pairs of IRP pseudo-instructions to be generated at a lower level.

```
SUMSQ MACRO   T,B
      STZ     T
      IRP     B
      LDQ     B
      FMP     B
      FAD     T
      STO     T
      IRP
SUMSQ END
```

Figure 26-1

The macro-compiler will not generate an ETC card for an IRP pseudo-instruction whose subargument string does not fit on one card, but will process the string internally.

The IRP pseudo-operation is undefined outside of a macro-operation; hence, it should not occur explicitly in the range of a RMT pseudo-operation. If it should appear in this way, it will not be flagged, but will be treated as if it had no argument, i.e., it will cause omission of the following cards, until the matching IRP is encountered.

The RMT and RMT* Pseudo-Operations

Macro-instructions may require assignment of temporary storage, definitions of constants, closed

```
00007                          SUMSQ   A,(X,Y,Z)
00007   0600 00 0 77777        STZ     A
                               IRP     X,Y,Z
00010   0560 00 0 77776        LDQ     X
00011   0260 00 0 77776        FMP     X
00012   0300 00 0 77777        FAD     A
00013   0601 00 0 77777        STO     A
00014   0560 00 0 77775        LDQ     Y
00015   0260 00 0 77775        FMP     Y
00016   0300 00 0 77777        FAD     A
00017   0601 00 0 77777        STO     A
00020   0560 00 0 77774        LDQ     Z
00021   0260 00 0 77774        FMP     Z
00022   0300 00 0 77777        FAD     A
00023   0601 00 0 77777        STO     A
```

Figure 26-2

subroutines, or other allocations of memory. If such storage is assigned within the macro-operation, either it must be bypassed by transfer instructions or the programmer can keep track of the storage requirements and define the necessary symbols whenever convenient. The pseudo-operation RMT (Remote Sequence) provides a means by which such storage may be automatically assigned later in the assembly, at any point the programmer may specify by a RMT* pseudo-operation. The constituents of the RMT pseudo-instruction are:

1. Blanks in the location field;
2. The operation code RMT in the operation field; and
3. Blanks in the variable field.

A remote sequence is defined as all the source instructions, or macro-generated instructions, bracketed by a pair of RMT pseudo-instructions. Remarks cards with an * in column 1 appearing within a remote sequence will not appear in the expansion. An RMT pseudo-instruction cannot appear explicitly within the range of a remote sequence; the nested pair will cause the termination of the range, and the reopening of another range. However, a macro-instruction within the range of a remote sequence may itself cause pairs of RMT pseudo-instructions to be generated at a lower level. A remote sequence can be nested in a macro-operation. and macro-operation prototypes not including remote sequences may be nested to any desired depth within a remote sequence. When the remote sequence is assembled, any macro-definitions nested within will be defined and any macro-instructions nested within will be expanded.

Remote sequences may be defined outside of macro-operations, but should be used sparingly, since they may result in termination of assembly owing to macro-definition table overflow. Little is gained by the use of this pseudo-operation outside of macro-operations.

Figure 26-3 illustrates a typical remote sequence.

```
    RMT
    CLA    *
XX  DEC    1E1
YY  DUP    1,2
    PZE
ZZ  BSS    10
    PZE    7
    RMT
```

Figure 26-3

Subsequently, an RMT* pseudo-instruction will cause all waiting remote sequences to be assembled. The constituents of an RMT* pseudo-instruction are:

1. Blanks in the location field;
2. The operation code RMT in the operation field;
3. An asterisk in the variable field.

If the remote sequence in Figure 26-3 has been defined, the coding in Figure 26-4 would be assembled following an RMT*.

If any remote sequences remain at the end of an assembly, they will be assembled following the program END card. Since remote sequences may include macro-definitions, and macro-definitions may generate remote sequences, a significant amount of coding may follow the END card. This generated coding will precede any literals.

Heading characters for remote sequences are those in effect at the time of definition, and not at the time of assembly. A remote sequence, defined or assembled in a multiply-headed region, may be improperly headed, and caution should be exercised.

```
00000                                RMT    *                                .001
00000   0500 00 0 00000              CLA    *                                .001
00001  +204500000000          XX     DEC    1E1                              .001
                     00002    YY     DUP    1,2                              .001
00002   0 00000 0 00000              PZE
00003   0 00000 0 00000                                                      .001
00004                         ZZ     BSS    10                               .001
00016   0 00000 0 00007              PZE    7
```

Figure 26-4

CHAPTER 27: GENERAL DISCUSSION

The FAP assembler has two objectives: the first is translating symbolic language to machine language, assigning storage locations, and performing other operations necessary to produce the final object program - this is known as assembly; the second is updating a symbolic tape by changing, deleting, or adding instructions. The update may occur concurrently with the assembly of the updated symbolic language program.

Updating involves up to three tapes: one tape, the System Input tape, is used by all assemblies, compilations, and executions; and two tapes, the Update Input tape and the Update Output tape, are used only for updating.

The System Input tape contains all the jobs to be processed by a monitored system. A job may have many elements: compilations, assemblies, and executions. In the case of an update, the corrections to be made to a specified symbolic tape constitute a job element and are contained in a job on the System Input tape.

The Update Input tape is a blocked or unblocked symbolic tape which requires updating. All additions, changes, and deletions are made on the basis of the serialization in card columns 73-80. Thus, instructions on the Update Input tape must be properly serialized. An end of file on the Update Input tape is ignored.

The Update Output tape is an updated, blocked or unblocked, symbolic tape which includes all the requested corrections. This tape may be assembled at a later date; if it includes the necessary control cards, it may be a System Input tape.

The monitored system requires two tapes which deserve note. The System Listing tape contains the listing output for all successful elements of all jobs on the System Input tape. The System Punch tape contains row or column binary card images for the processed jobs.

Uses of the Update Facility

FAP's Update facility may be used for: maintaining the 709/7090 FORTRAN System; multi-reel input and output; subdividing of, or extracting from, an input tape; spacing tape; and sequence checking. See examples of some of these uses below (Figures 30-1 through 30-5).

Blocked Update Output Tape

If the programmer requests a blocked Update Output tape, this symbolic tape will be blocked 14 words per card, 16 cards per block. Control cards and END cards are always unblocked.

Cards with an * or $ in column 1 are considered to be control cards and are unblocked if they follow another such card, or any update pseudo-operation, or an END card. They are considered to be remarks cards and are blocked if they follow any other card.

Sequence Checking and Serialization

Card images on the System Input and Update Input tapes will be checked for proper sequencing by the serialization in card columns 73-80. If the cards are determined to be out of sequence, an error message is printed on- and off-line, and assembly continues.

The Update Output tape and the System Listing tape may be reserialized by using the NUMBER pseudo-operation (see description below). In the case of the Update Output tape, this provides for proper serialization for a subsequent assembly.

Corrections to the Update Input tape are made on the basis of the serialization in card columns 73-80 of the instructions on the System Input and Update Input tapes. A serialized instruction on the System Input tape will replace an instruction with matching serialization on the Update Input tape; an instruction with nonmatching serialization will be inserted in sequence. In the instruction serialization, a BCD blank is sequenced following the character * and preceding the character /. If columns 75-80 of a card image on the System Input or System Update tapes are all blank, the serialization is taken to be all zeros for sequencing, and the instruction is used immediately. Instructions that are inserted or replaced appear, so labeled, on the pre-processor assembly listing.

Serialization on an update pseudo-instruction causes an instruction on the Update Input tape with identical serialization to be deleted before the pseudo-operation is interpreted. Thus, there will not be an instruction with this serialization on the Update Output tape or on the System Listing tape. An update pseudo-operation may be serialized to indicate at what point the pseudo-operation should be interpreted.

Tape Positioning

Update pseudo-instructions, except SKIPTO, do not require matching serialization. To ensure proper spacing of tapes, it is necessary to have symbolic instructions on the System Input tape whose serialization matches those on the Update Input tape. These

"spacer" cards will position the Update Input tape to the next instruction to be updated.

It is also necessary for the System Input tape to include a serialized END or ENDUP pseudo-instruction to properly position the Update Input tape at the end of the update. The END pseudo-instruction serialization must match that of the END pseudo-instruction on the Update Input tape.

The ENDUP pseudo-instruction is used for an update without assembly. If used, it must immediately follow an instruction (usually an END) whose serialization indicates where the Update Input tape is to be positioned. The ENDUP pseudo-instruction must be properly serialized or have no serialization.

If a multi-reel update has been terminated because of source program or machine error, the Update Input and Update Output tapes may be positioned incorrectly. Exercise caution in restarting or continuing. The Source and Machine Error Records will attempt to reposition a single input tape or single output tape correctly.

Illegible Input Instructions

All instructions on an input tape which are either redundant or in the wrong mode are considered illegible. If there are illegible instructions on the System Input tape, the assembly is terminated. If an instruction or block of instructions on the Update Input tape is illegible, the instruction(s) is omitted, a message is printed on- and off-line, and the job is continued. It is possible to insert lost instructions during a later update. The omission of such instructions will usually result in an assembly error, but the Update Input tape will be properly positioned for the next assembly, unless the illegible instruction is an END pseudo-instruction or spacer card.

Updating a FORTRAN Source Program Deck

In order to update a FORTRAN source program deck, the Update Output tape must be unblocked and assembly must be deleted. The ENDUP pseudo-operation is used to terminate the job.

Listing Update Pseudo-Instructions

Update pseudo-instructions are listed in the preprocessor assembly listing and normally do not appear on the Update Output and System Listing tapes. However, if the pseudo-instruction is in error and is flagged, it is written on these tapes.

CHAPTER 28: UPDATE PSEUDO-OPERATIONS

The UPDATE Pseudo-Operation

The UPDATE pseudo-operation is used to initiate the updating mode, to assign Update Input and Update Output tapes, and to specify whether or not blocking and assembly are required. The constituents of the UPDATE pseudo-instruction are:

1. Blanks in the location field;
2. The operation code UPDATE in the operation field;
3. Two symbolic expressions and two strings of characters, all separated by commas, in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

The first subfield of the variable field is an expression designating the logical system tape number of the Update Input tape. This subfield may be void or zero, implying that there is no Update Input tape, or it may be the logical number of the tape containing the symbolic input to be updated. If the FORTRAN Monitor System is used, tapes 1 through 8 may not be used as update tapes.

The second subfield of the variable field is an expression designating the logical tape number of the Update Output tape. This subfield may be void or zero, implying that there is no Update Output tape, or may be the logical number of the tape that is to contain the updated symbolic output.

The third subfield of the variable field is a string of characters which indicates whether or not the Update Output tape is to be blocked. If this subfield is void or zero, the Update Output tape will be blocked; otherwise, the Update Output tape will be unblocked.

The fourth subfield is a string of characters which indicates whether or not assembly is required. If this subfield is void or zero, assembly is required; otherwise, assembly is not required. The UPDATE pseudo-instruction containing this subfield must appear in the first card group. Having once appeared, neither this subfield nor its preceding comma may appear on another UPDATE card. If assembly is not required, FAP is reduced to an updating or blocking routine, or both, since no table entries are made and Pass 2 is omitted; the only instructions that are recognized are Update pseudo-instructions, END pseudo-instructions, and those instructions with an * in card column 1.

Serialization in card columns 73-80 of the UPDATE pseudo-instruction may be useful for multi-reel input and output.

The NUMBER Pseudo-Operation

The NUMBER pseudo-operation is used to reserialize columns 73-80 of the symbolic instructions that are output on the System Listing and Update Output tapes. The constituents of the NUMBER pseudo-instruction are:

1. Up to six alphameric characters in the location field;
2. The operation code NUMBER in the operation field;
3. A number less than 32768 in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

The location field and the variable field define the serialization to be inserted in the instructions on the output tapes; the alphameric characters will be left-justified in card columns 73-78, with blanks omitted, and the number in the variable field will be right-justified in card columns 75-79.

A zero is automatically supplied in column 80. The programmer must ensure that the alphameric and numeric fields do not overlap in the program. Reserialization begins with the serial indicated in the location and variable fields of the NUMBER pseudo-instruction. Regardless of the contents of the location field, if the variable field is omitted, reserialization is suspended and old serial numbers (if any), in card columns 73-80, will be maintained. In order to reserialize from zero, an explicit zero must appear in the variable field, in addition to any characters which may appear in the location field.

The DELETE Pseudo-Operation

The DELETE pseudo-operation causes the deletions of one or more instructions on the Update Input tape. Deleted cards will appear on the pre-processor assembly listing labeled as deleted. The constituents of the DELETE pseudo-instruction are:

1. Blanks in the location field;
2. The operation code DELETE in the operation field;
3. THRU, when required, in the variable field; and
4. A serialization number of eight characters in card columns 73-80.

If the variable field is blank, the instruction with matching serialization will be deleted; if the variable field contains THRU, all instructions starting at the current position of the Update Input tape, up to and including the instruction with matching serialization, will be deleted. In the latter case, if matching serialization does not exist, deletions will be made up to, but not including, the next instruction of higher serialization.

The IGNORE Pseudo-Operation

The IGNORE pseudo-operation causes the deletion of one or more instructions on the Update Input Tape. Deleted cards will not appear on the pre-processor assembly listing. The constituents of the IGNORE pseudo-instruction are:

1. Blanks in the location field;
2. The operation code IGNORE in the operation field;
3. THRU, when required, in the variable field; and
4. A serialization number of eight characters in card columns 73-80.

If the variable field is blank, the instruction with matching serialization is deleted; if the variable field contains THRU, all instructions, starting at the current position of the Update Input tape, up to and including the instruction with matching serialization, will be deleted. In the latter case, if matching serialization does not exist, deletions will be made up to, but <u>not</u> including, the next instruction of higher serialization.

The SKIPTO Pseudo-Operation

The SKIPTO pseudo-operation will cause the deletion of one or more instructions up to, but <u>not</u> including, an instruction with matching serialization. The constituents of the SKIPTO pseudo-instruction are:

1. Blanks in the location field;
2. The operation code SKIPTO in the operation field;
3. Blanks in the variable field; and
4. A serialization number of eight characters in card columns 73-80.

Instructions deleted will not appear on the pre-processor assembly listing. Instructions of higher serialization will <u>not</u> cause this operation to be terminated.

The ENDFIL Pseudo-Operation

The ENDFIL (End of File) pseudo-operation will cause an end of file to be written on the addressed update tape. The constituents of the ENDFIL pseudo-instruction are:

1. Blanks in the location field;
2. The operation code ENDFIL in the operation field;
3. A FAP expression or logical tape number in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

If the logical tape number in the variable field is that of the current Update Output tape, assigned by an UPDATE pseudo-instruction, the last partial

block of instructions waiting to be written will be written on the Update Output tape. The end of file will then be written. If the variable field is blank, the end of file is written on the Update Output tape after the last partial block of instructions waiting to be written has been written.

The REWIND Pseudo-Operation

The REWIND pseudo-operation causes the addressed update tape to be rewound. The constituents of the REWIND pseudo-instructions are:

1. Blanks in the location field;
2. The operation code REWIND in the operation field;
3. A FAP expression or logical tape number in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

If the logical tape number in the variable field is that of the current Update Output tape assigned by an UPDATE pseudo-instruction, the last partial block of instructions waiting to be output will be written on the Update Output tape before it is rewound. If the variable field is blank, the Update Output tape is rewound. If the Update Input or Update Output tape is rewound, it is logically disconnected. No update operation referring to it will be executed unless it is reassigned by a subsequent UPDATE pseudo-instruction.

The UNLOAD Pseudo-Operation

The UNLOAD pseudo-operation causes the addressed update tape to be rewound and unloaded. The constituents of the UNLOAD pseudo-instruction are:

1. Blanks in the location field;
2. The operation code UNLOAD in the operation field;
3. A FAP expression or logical tape number in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

If the logical tape number in the variable field is the current Update Output tape assigned by an UPDATE pseudo-instruction, the last partial block of instructions waiting to be output will be written on the Update Output tape before it is rewound and unloaded. If the variable field is blank, the Update Output tape is rewound and unloaded. If either the Update Input or Update Output tape is rewound and unloaded, it is logically disconnected. It may be reassigned by a subsequent UPDATE pseudo-instruction; however, the operator should be informed to ready the tape by a PRINT pseudo-instruction sufficiently in advance to avoid delaying the assembler.

The SKPFIL Pseudo-Operation

The SKPFIL (Skip File) pseudo-operation causes the addressed update tape to be spaced forward until an end of file is passed. The constituents of the SKPFIL pseudo-instruction are:

1. Blanks in the location field;
2. The operation code SKPFIL in the operation field;
3. A FAP expression or logical tape number in the variable field; and
4. Serialization, which is optional, in card column 73-80.

If the logical tape number in the variable field is the current Update Output tape assigned by an UPDATE pseudo-instruction, the last partial block of instructions waiting to be output will be written on the Update Output tape before it is spaced. If the variable field is blank, the Update Input tape is spaced forward past an end of file.

If an Update Input tape contains a binary file to be skipped over, the SKPFIL pseudo-operation should appear outside the range of an UPDATE pseudo-operation which pertains to that tape. This will prevent a redundancy error message from occurring because of look-ahead reading initiated by an UPDATE pseudo-operation.

The UMC Pseudo-Operation

The UMC (Updated Macro-Instruction Cards) pseudo-operation is used to write the instructions generated by a macro-instruction on the Update Output tape and to delete macro-definitions and macro-instructions. The constituents of the UMC pseudo-instruction are:

1. Blanks in the location field;
2. The operation code UMC in the operation field;
3. Blanks in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

Initially, macro-definitions and macro-instructions are written on the Update Output tape, but macro-generated instructions are not written. The UMC pseudo-operation reverses this procedure. Successive appearances of UMC cause switching from writing to not writing macro-generated instructions and, therefore, from deleting to writing macro-definitions and macro-instructions. The programmer should exercise caution in the use of this pseudo-operation, since, under alternate modes, a macro-definition may be deleted, while a macro-instruction for the same code may remain. An IRP pseudo-instruction in a macro-generated sequence is never written on the Update Output tape.

The serial numbers associated with deleted macro-definitions will be deleted. All instructions generated by the macro-instruction will be serialized with the

serial number associated with the macro-instruction.

Since remaining remote sequences (see RMT, page 50) will be expanded after the END card is written, when UMC is in effect, they must be inserted before the END card by the use of the RMT* pseudo-operation.

If UMC is to be used, all macro-definitions for the program should be written in the alternate form. In this form, the location field is an argument and is defined within the macro-definition.

The ENDUP Pseudo-Operation

The ENDUP (End Update) pseudo-operation signals the termination of an update with assembly deleted. The constituents of the ENDUP pseudo-instruction are:

1. Blanks in the location field;
2. The operation code ENDUP in the operation field;
3. Blanks in the variable field; and
4. Serialization, which is optional, in card columns 73-80.

If assembly is deleted, END cards will be unblocked, but will not terminate the update. This makes it possible to update more than one program at a time. If assembly is not deleted, ENDUP will be undefined. Blank serialization on the ENDUP card will cause immediate termination of the update.

The PRINT Pseudo-Operation

The PRINT pseudo-operation first causes card columns 14-72 of the PRINT card to be printed on-line during the first pass over the input deck; then it causes a machine halt. The constituents of the PRINT pseudo-instruction are:

1. Blanks in the location field;
2. The operation code PRINT in the operation field; and
3. A string of alphameric characters starting in card column 14.

CHAPTER 29: OPTIONAL SERIALIZATION IN UPDATE PSEUDO-INSTRUCTIONS

Only the DELETE, IGNORE, and SKIPTO pseudo-instructions require serialization in card columns 73-80. Without serialization they have no effect except that of inhibiting sequence checking (see Chapter 1).

UPDATE, NUMBER, ENDFIL, REWIND, UNLOAD, SKPFIL, UMC, and ENDUP may have serialization in card columns 73-80. If these pseudo-instructions are serialized on the System Input tape, the following occurs:

1. Either the Update Input tape is positioned at an instruction with matching serialization or, if there is no instruction with matching serialization, the Update Input tape is positioned after the instruction of lower serialization and before the instruction of higher serialization;
2. The instruction on the Update Input tape with matching serialization is deleted; and
3. The pseudo-instruction is interpreted.

If the pseudo-instruction is not serialized, steps 1 and 2 are omitted; thus, the pseudo-instruction is interpreted at the current position of the Update Input tape.

CHAPTER 30: UPDATE EXAMPLES

Figures 30-1 through 30-5 illustrate the use of some of the update pseudo-operations.

```
UPDATE  9,10
```

Figure 30-1

The instruction in Figure 30-1 may be used to merge the correction cards that appear on the System Input tape after the UPDATE pseudo-instruction, with those on logical tape 9, the Update Input tape; then assemble.

```
UPDATE  9,10         ...          F0213750
```

Figure 30-2

The instruction in Figure 30-2 may be used to position logical tape 9, the Update Input tape, to an instruction with serialization F0213750; delete F0213750; merge the correction cards on the System Input tape with those on logical tape 9, the Update Input tape, write a blocked symbolic tape on logical tape 10, the Update Output tape; then assemble.

```
*    FAP
     UPDATE    ,10,,D
     ENDFIL
*    END TAPE
     ENDFIL
     REWIND
     UNLOAD    9
     ENDUP
```

Figure 30-3

The instructions in Figure 30-3 may be used to write an end of file, an END TAPE card, another end of file, rewind logical tape 10, the Update Output tape; and unload logical tape 9.

```
*       DATE      6/18/62
*       JOB IDENTIFICATION
*       PACK
*       FAP
*       PAGE TITLE CARD
        UPDATE    ,10
   F01  NUMBER  0
*       DATE      6/18/62
*       JOB IDENTIFICATION
*       PACK
*       FAP
*       PAGE TITLE CARD
        COUNT     5000
        ABS
        ***
        ***
        ***
        END
*       FAP
        UPDATE    ,10
        ENDFIL
*       END TAPE
        ENDFIL
        UNLOAD
        END
   (END OF FILE)
*       END TAPE
   (END OF FILE)
```

Figure 30-4

The instructions in Figure 30-4 may be used to assemble the cards on the system input unit while simultaneously generating a blocked, serialized Update Output tape, logical tape 10, for a later assembly.

```
*       FAP
        UPDATE  9,10,,D
   F01  NUMBER  0
        CAL     ERASE                                              F0101010
        DELETE                                                     F0101200
        NZT     SYMBL                                              F0101461
        TRA     ERROR
   F02  NUMBER  0                                                  F0199999
        DELETE                                                     F0201730
        DELETE  THRU                                               F0201840
        DELETE                                                     F0202040
        IGNORE  THRU                                               F0204170
   F03  NUMBER  0                                                  F0299999
        END                                                        F0302880
        ENDUP
```

Figure 30-5

The instructions in Figure 30-5 may be used to update and reserialize the instructions on the Update Input tape, logical tape 9; to delete assembly; and to write the updated symbolic output on logical tape 10, the Update Output tape.

CHAPTER 31: SUBROUTINES

A subroutine is a set of program steps, taken as a unit, which performs a task and which forms part of a program. Subroutines are useful in many applications and for many reasons. One reason for using a subroutine is to utilize existing coding. Another reason is that the effort of debugging a program is substantially reduced if subroutines are used which have been debugged previously. Finally, by entering a subroutine from several points in a program, one set of instructions can do the work of many, thus saving core storage space.

Open and Closed Subroutines

Two types of subroutines are used: open subroutines and closed subroutines. An open subroutine is a set of instructions inserted into the body of a program and encountered in the normal path of flow. An open subroutine is most effectively embedded in coding by use of the macro-operation. Open subroutines require no special linkage. A closed subroutine is, in a sense, a separate program. The main routine using a closed subroutine transfers control to it, and when the subroutine has accomplished its functions, it returns control to the appropriate point in the main routine. Closed subroutines require special communication facilities and offer flexibility and economy of storage. The remainder of this discussion will be devoted to closed subroutines.

Linkages

Since a closed subroutine may be entered from different points in the main routine, the main routine must enter the subroutine in such a way that the subroutine will be able to return to the correct point. The instruction or instructions which make this possible form what is called the "linkage" between the main routine and the subroutine. In the 704, 709, 7090, and 7094 the standard linkage is a Transfer and Set Index instruction which uses Index Register 4. The instruction in Figure 31-1 transfers control to the subroutine SUBR and places the 2's complement of the location of the TSX instruction in Index Register 4. In order to return to the location following the TSX, the subroutine may use the instruction in Figure 31-2.

```
TSX       SUBR,4
```

Figure 31-1

```
TRA       1,4
```

Figure 31-2

Calling Sequences

Often it is necessary for a main routine and a subroutine to communicate with one another. The main routine may have to furnish the subroutine with numbers on which to operate; these numbers are called arguments. Also, the subroutine may produce one or more numbers which are needed by the main routine. For example, a mathematical routine such as SIN will have one argument and will produce one number as its result. In such a case, it is usual for the main routine to place its argument in various registers of the computer, and also for the subroutine to leave its results in one or more of the registers. However, often there are more arguments, or more results, than there are registers. In this case, the subroutine will usually be written so that the argument may appear in successive locations in the main routine following the location of the linkage. Then the beginning and ending instructions of the subroutine might appear as in Figure 31-3.

```
00000  0500 00 4 00001    SUBR CLA   1,4
00001  0601 00 0 77461         STO   ARGA
00002  0500 00 4 00002         CLA   2,4
00003  0601 00 0 77460         STO   ARGB
00004  0500 00 4 00003         CLA   3,4
00005  0601 00 0 77457         STO   ARGC
00006  0634 00 4 00012         SXA   END,4
00007  0 00000 0 00000         ***
00010  0 00000 0 00000         ***
00011  0 00000 0 00000         ***
00012  0774 00 4 00000    END  AXT   0,4
00013  0020 00 4 00004         TRA   4,4
```

Figure 31-3

The linkage and the locations containing parameters together form what is called the calling sequence. Typically, the subroutine returns control to the location following the last word of the calling sequence. In FORTRAN, an argument appears in the address portion of each location of the calling sequence; the argument is usually the address of a location containing the argument needed by the subroutine. The operation code of a FORTRAN calling-sequence word contains a TSX instruction; the tag bits are zeros. Thus, a FORTRAN calling sequence appears as in Figure 31-4.

```
TSX     SUBR,4
TSX     ARGX
TSX     ARGY
TSX     ARGZ
```

Figure 31-4

The number of tagless TSX instructions in the calling sequence is determined by the subroutine. For some subroutines, the length of the calling sequence is variable. The end of the calling sequence is signaled by the occurrence of something other than a tagless TSX. The use of the calling sequence is a powerful tool; the parameters may be the locations of entire arrays, some used by the subroutine, and some produced by it.

FORTRAN Linkage and Calling Sequences

Five types of subroutines may be used in FORTRAN. These are built-in functions, arithmetic statement functions, library functions, FUNCTION subprograms, and SUBROUTINE subprograms. The built-in functions are compiled as open subroutines. Arithmetic statement functions are closed subroutines that are compiled as an integral part of the program in which they appear. A further discussion of built-in functions and arithmetic statement functions is given in the IBM Reference Manual, 709/7090 FORTRAN Programming System, Form C28-6054-2.

The argument of a library function is placed in the Accumulator by the main routine before control is transferred to the subroutine. If there is a second argument, it is placed in the MQ register. The calling sequence for a library function consists solely of the linking TSX, which has a tag of 4. A library function produces a single number as its result. This number is left in the Accumulator, and control is returned to the instruction following the one-word calling sequence. The FORTRAN compiler is directed to use this calling procedure by the presence of the letter F at the end of the function's name. The compiler removes this terminal F, since it is not present in the corresponding routine in the library.

A FUNCTION subprogram is compiled or assembled separately from the program which calls it. The calling program utilizes the FORTRAN calling sequence described in the preceding section. Thus, the number of arguments is practically unlimited. Like a library function, the FUNCTION subprogram produces a single number as its result. This number is left in the Accumulator, and control is returned to the instruction following the last word of the calling sequence. The FORTRAN compiler is directed to use this calling procedure by the absence of the letter F at the end of the function's name. Further information regarding FUNCTION subprograms is given in the 709/7090 FORTRAN reference manual referenced above.

Each of the four types of function subroutines discussed above is called in the same manner in a FORTRAN source program. The name of the function is used implicitly to form a part of some statement, and the compiler automatically constructs the required open subroutine or calling sequence. A SUBROUTINE subprogram is called explicitly from a FORTRAN source program by the use of the CALL statement. The CALL statement produces a FORTRAN calling sequence. In this way, a SUBROUTINE subprogram is like a FUNCTION subprogram. Unlike the FUNCTION subprogram, the SUBROUTINE subprogram does not leave a result in the Accumulator. Instead one or more arguments in the calling sequence are used to specify where the results should be stored.

Segmentation

It is often advantageous for a programmer to divide a program into segments, one segment being the main program and other segments being subprograms. The segments are linked together by CALL statements in FORTRAN and the CALL pseudo-instructions in FAP. The segment may be compiled separately, and, by supplying each segment in turn with test data, each segment may be debugged separately.

Common Storage

Often a subprogram or segment will require the values of a large number of variables computed by another subprogram or segment. If the names of all these variables are placed in their respective calling sequences, the task of writing these calling sequences becomes difficult, and time is lost interpreting these calling sequences in the execution of the resulting program.

By using the COMMON statement in FORTRAN programs and the COMMON pseudo-instruction in FAP programs, the programmer may reserve locations in upper core storage for the storage of certain variables that are used in more than one subprogram or segment. If the assignment of common storage is performed identically in all subprograms or segments, then references to a given variable in different subprograms or segments will result in reference being made to the same location in core storage.

In FORTRAN, the COMMON statement is used in conjunction with the DIMENSION statement to determine storage assignments. The first variable named in a COMMON statement is assigned to a block of storage whose highest location is the octal address 77461. In FORTRAN, arrays are stored backward in core storage, and the name of the array refers to the highest address in the block of storage allocated. The size of the block is determined by a DIMENSION statement; if the variable does not appear in a

DIMENSION statement, one location is reserved. The location assigned to the second variable in a COMMON statement is computed by subtracting the length of the first block from the octal number 77461. Assignment proceeds by the allocation of successively lower addresses to successive variables in common storage.

Assignment of common storage in FAP is accomplished by the use of the COMMON pseudo-operation (see page 24). The length of the block to be reserved is given in the variable field of each COMMON instruction, while the order of these instructions determines the order of the locations assigned.

Relocatable Binary

The various subprograms or segments which make up a complete job are compiled or assembled individually. The library subroutines are present on the library tape in binary form. In order to execute a job, all the component programs must be loaded into the computer in binary form. The loading operation is performed by BSS (Binary Symbolic Subroutine) Control, which requires that all programs to be loaded be in the relocatable binary format. This format is so named because it enables the loader to relocate the program, that is, to place the program into whatever area of core storage is available for it.

The process of relocation involves three operations:

1. Each word of the program is moved to a new location.
2. A number, the relocation constant, is added to the address portion of certain words of the program.
3. The relocation constant is added to the decrement portions of certain words of the program.

Occasionally, a program requires some other form of modification for relocation. For instance, use of complemented locations may require that the relocation constant be subtracted from the address or decrement portion of some words. BSS Control is not capable of performing this sort of modification, thus, such a program must be recoded before being used with FORTRAN programs. If the FAP assembler is instructed to produce coding which cannot be relocated by the BSS Control, a relocation error will be indicated.

A program will consist of a sequence of card images on tape, each card image being the tape equivalent of one binary card. The unique feature of the relocatable binary format is that each card image contains within it a sequence of relocation indicator bits. These bits indicate to BSS Control which of the address and decrement portions of words on the card are to be modified. A discussion of the relocation indicator bits and of the relocatable binary card format is given in the IBM Reference Manual, 709/7090 FORTRAN Operations, Form C28-6066-3.

Transfer Vector

In addition to performing relocation, BSS Control makes possible references between different subprograms. The mechanism by which this reference is made is called the transfer vector. The BCD representation of the name of each subprogram referenced in the program is placed in the transfer vector, which appears at the beginning of the program.

When the programs are loaded, BSS Control determines the location of the first instruction to be executed in the subprogram and replaces the transfer vector word by a transfer to this location. Thus, the linking TSX in the main program transfers control to the transfer vector word, which is also located in the main program, and the transfer vector word transfers control to the subprogram. Since Index Register 4 is set by the linking TSX and is not altered by the transfer vector instruction, the subprogram may use Index Register 4 to locate its arguments and return point.

In general, there will be as many words in the transfer vector as there are different subroutine names referenced in the program. Each transfer vector name will be replaced by an appropriate transfer instruction during loading.

BSS Control is guided in this replacement process by program cards. Each binary deck begins with a program card. The program card gives the number of locations in the transfer vector, the number of locations in the program, the number of locations of common storage used by the program, and every entry-point name used by the program together with the corresponding entry address. A detailed description of the program card is given in the IBM Reference Manual, 709/7090 FORTRAN Operations, Form C28-6066-3.

The FAP assembler, guided by the ENTRY instruction in the symbolic program, produces a program card. The transfer vector is also produced automatically by FAP and contains every subroutine name used in a CALL statement or referenced by use of the character $.

CHAPTER 32: A BRIEF DESCRIPTION OF THE ASSEMBLY PROCESS

The user of FAP will be better able to keep in mind the capabilities and limitations of the assembler if he is familiar with the basic structure of the assembly program. However, Parts 1 through 4 of this manual are complete, and the programmer need not be familiar with the material covered here in order to make full use of FAP.

The assembly process consists of two passes.

During the first pass, the symbolic cards are read from the input tape and are copied onto the intermediate tape, and values are assigned to all location symbols. These values are tabulated in a dictionary called the "symbol table." If a BCD source deck is serialized, it will be checked for cards out of sequence. Updating occurs during Pass 1. During the second pass, the symbolic cards are read from the intermediate tape, values are substituted for operation codes and symbols, the assembly listing is written on the output tape, and off-line row or column binary output is written on the output tape. Only if on-line cards are requested will a binary intermediate tape be written.

The first pass is devoted to the construction of the symbol table, to updating, to the definition of remote sequences and macro-definitions, and to the definition of operations defined by OPD, OPSYN, and OPVFD. In addition, macro-instructions are expanded and remote sequences are generated in Pass 1. The assembler uses two counters called the "location counter" and "program counter" to keep track of the "next location to be assigned." Initially, both counters are set to zero. When an instruction, which is not a pseudo-operation, is processed, its location symbol, if any, is entered into the symbol table together with the current value of the program counter. Then both counters are increased by one or, for certain orders, by two. Pseudo-operations affect both counters in different ways. ORG sets both counters to the given value; LOC sets only the program counter. BSS and BES increase the contents of both counters by the given value. EQU, BOOL, and several other pseudo-operations have no effect on either counter. Whenever a symbol is encountered in the location field of an instruction, it is defined. Some pseudo-operations require reference to be made to the symbol table in Pass 1. When such reference must be made, the symbol referred to must already have been entered in the symbol table. This is the reason for the statement that symbols must have been previously defined. In most instructions, the symbols in the variable field are not evaluated until Pass 2, and hence need not have been previously defined.

Macro-definitions and remote sequences are reduced to skeleton form and inserted into the macro-definition table. Each macro-operation code and a pointer to its definition in the macro-definition table is inserted into the combined operations table. Remote sequences are chained.

When a macro-instruction is encountered, its arguments are tabulated. When a macro-definition is encountered, the arguments are tabulated, and each appearance of an argument is replaced by a flag indicating the position of the argument in the argument list.

The macro-definition skeleton is expanded, and each appearance of a flag is replaced by a specific argument. If an RMT* is encountered, each remote sequence in the chain is expanded.

Halfway through Pass 1 (as decided on the basis of the COUNT card) the assembler rewinds the first intermediate tape and begins writing on the second intermediate tape. If the assembler used only one intermediate tape, time would be lost while the assembler waited for this tape to rewind. Using two intermediate tapes, FAP works with each intermediate tape in turn while the other is rewinding.

The intermediate tapes contain 16 words per card image, 16 card images per block. The first 80 characters are the symbolic instruction being processed. Certain flags are transmitted from Pass 1 to Pass 2 in the last 24 bits of the fourteenth word. The fifteenth word contains either a literal or a pseudo-operation definition, which was evaluated in Pass 1. The sixteenth word contains either the operation code or the pseudo-operation transfer address; if an operation is undefined, this word is all zeros. The intermediate tapes are written and read in binary because of the information in the fifteenth and sixteenth words.

At the end of Pass 1, the assembler sorts the symbol table to make searching more efficient in Pass 2. During this sort, any symbols which have been defined more than once are detected and flagged in the symbol table. The assembler then begins the second pass. The assembler reads the first intermediate tape, which is now rewound, and fully processes all instructions, using the symbol table constructed in Pass 1. By the time the assembler has finished processing the instructions on the first intermediate tape, the second intermediate tape should be rewound.

The symbol-defining pseudo-operations are interpreted in the first pass in order to make entries to the symbol table. The definitions are saved as the fifteenth word of the intermediate record. They are interpreted again in the second pass in order to verify the value in the symbol table.

The storage-allocating pseudo-operations are interpreted in the first pass to define the symbol in the location field and to alter the location and program counters in the case of BSS and BES, or common counter in the case of COMMON. In the second pass, the effect of these pseudo-operations is recorded in the assembly listing, and, in the case of BSS and BES, the binary output is also affected.

The data-generating pseudo-operations are interpreted in the first pass to the extent necessary to define the symbol in the location field and to determine the number of words generated (which number is added to the location and program counters). The generated data words are developed when the pseudo-operations are interpreted in the second pass.

Instructions which are not pseudo-operations are not interpreted in the first pass, except to define the symbol, if any, in the location field, and to increase the location and program counter by one or two. The operation, address, tag, and decrement bits are all assembled in the second pass.

Subroutine names defined by CALL or EXTERN instructions or by use of the character $ are tabulated in the first pass. The number of different subroutine names so tabulated gives the length of the program's transfer vector. At the end of the first pass the length of the transfer vector is added to the value of every relocatable symbol in the symbol table except those symbols which are themselves subroutine names. The length of the transfer vector, the length of the program, and the lowest address of common storage used are placed in the program card image at the beginning of the second pass. Then the ENTRY instructions are processed to produce the completed program card (or program cards).

When a DUP pseudo-operation is encountered in the first pass, the assembler processes each instruction in the range of the DUP just once, computes the amount by which the program counter has been increased, multiplies this amount by the DUP count, and uses this product to compute the new values of the location and program counters. In the second pass, the assembler backspaces the intermediate tape and reprocesses the range of the DUP the correct number of times.

Literals are processed in the first pass. Each literal is evaluated to yield a 36-bit data item; these data items are tabulated to form the literal table. As each literal is encountered in the first pass, the literal table is searched to see if it already contains the corresponding data item. If the data item is not present, the literal table is expanded and the data item is added. The literal table is kept in sorted order at all times. When a symbolic instruction containing a literal is written on the intermediate tape, the data item generated by the literal is written on the intermediate tape as an extra word in the record. In the second pass, the data item is retrieved from the intermediate tape, and the literal table is then used to compute the address to be assigned to the data item. At the end of the second pass, the data items in the literal table are published in the assembly listing and in the binary output. This rather elaborate procedure for handling literals is used because, at the beginning of the second pass, the assembler must have already determined how many different literal data items are used in the program in order to compute the total length of the program; this length must be entered into the program card image.

Following the occurrence of a relocatable assembly in error, the binary instructions are erased from the Monitor binary output, and a labeled "FAILED" card is inserted in their place.

PSEUDO-OPERATIONS

The following is a list of pseudo-operations in the combined operations table.

Operation Code	Purpose
704	Set mode of assembly
7090	Set mode of assembly
9LP	Set card format
ABS	Set card format
BCD	Generate data
BCI	Generate data
BES	Allocate storage
BOOL	Define symbols
BSS	Allocate storage
CALL	Link programs
COMMON	Allocate storage
COUNT	Assembly information
DEC	Generate data
DETAIL	Control listing
DUP	Generate data
EJECT	Control listing
END	Assembly information
ENDFIL	Update information
ENDUP	Update information
ENTRY	Link programs
EQU	Define symbols
EVEN	Allocate storage
EXTERN	Link programs
FUL	Set card format
HEAD	Define symbols
HED	Define symbols
IFEOF	Link programs
IFF	Assembly information
INDEX	Control listing
IRP	Control macro-operations
LBL	Label binary cards
LIST	Control listing
LOC	Allocate storage
MAC	Macro-instruction
MACRO	Macro-definition
MAX	Define symbols
MIN	Define symbols
MOP	Macro-definition
NOCRS	Control macro-operations
NULL	Control listing
NUMBER	Update information
OCT	Generate data
OPD	Define operation code
OPSYN	Define operation code
OPVFD	Define operation code
ORG	Allocate storage
ORGCRS	Control macro-operations
PCC	Control listing
PMC	Control listing
PRINT	Update information
REF	Control listing
REM	Control listing
REWIND	Update information
RMT	Defer assembly
SET	Define symbols
SKPFIL	Update information
SPACE	Control listing
SST	Define symbols
SYN	Define symbols
TAPENO	Define symbols
TCD	Set card format
TITLE	Control listing
TTL	Control listing
UMC	Update information
UNLIST	Control listing
UNLOAD	Update information
UPDATE	Update information
VFD	Generate data

The following pseudo-operations are recognized by FAP, but do not appear in the combined operations table.

DELETE	Update information
ETC	Continue variable field
IGNORE	Update information
SKIPTO	Update information

MACHINE AND EXTENDED MACHINE INSTRUCTIONS

The following is a list of machine instructions and extended machine instructions, type, and permissible fields in the combined operations table.
Each is described in detail in the various machine manuals.

In the machine instruction list, the description of the machine instruction, the fields permitted and/or machine required, the mode of assembly, and other information is coded as follows:

Type	Instructions Instruction Format	Usage
A	0 00000 0 00000	15-bit decrement field
B	0000 xx 0 00000	No decrement field
C	0000 00 0 00000	8-bit decrement field
D	0000 xx 000000	18-bit address field
E	0760 xx 0 x0000	13-bit address field
K	00 0000 0 00000	4-bit prefix field

Code	Other Codes Interpretation
N	Not significant, or may change operation
P	Permissible
R	Required
T	Operation code defined or required address satisfied by use of TAPENO character
(I/O)	I/O unit address defined by operation mnemonic
4	Permissible in 704 mode only
9	Permissible in 7090 mode only
94	Permissible in 7090 mode only; no flag given for a 7094 instruction in a 7090 program
8	Decrement field not longer than 8 bits required

An x as the fourth character of an operation code indicates a variable channel operation; the channel designation A to H, or a properly defined TAPENO character, must be substituted for this character. This character should not be used with unit record equipment.

Violation of these rules will cause the instruction to be flagged. No 709 mode exists; hence, while assembling in the 7090 mode, drum instructions will be flagged, but indirect addressing I/O commands will not.

Operation codes which are on a "Request Price Quotation" basis are indicated by RPQ in the mode column.

The following notes pertain to the combined operations table:

(1) A count field in the low-order subfield position of the operation code is assembled from the fourth subfield of the variable field; for example,

```
ICC ,,,4
```

(2) The following extended machine operations are included in the combined operations table (listed above). They differ from the assembled machine operation codes only in that no flag will appear in the left margin of the listing for certain uncoded tag and decrement fields.

Mnemonic	As Assembled	Unfl. Omitted Fields
BRA	TXL	T, D
BRN	TXH	T, D
ZAC	PXD	T
ZSA	SXA	T
ZSD	SXD	T

(3) Alternate forms for PZE

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
ACL	B	R	P		P	
ADD	B	R	P		P	
ADM	B	R	P		P	
ALS	B	R	P		P	
ANA	B	R	P		P	
ANS	B	R	P		P	
ARS	B	R	P			

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
AXC	B	R	R			9
AXT	B	R	R			9
BRA	A	R	(2)	(2)		
BRN	A	R	(2)	(2)		
BSF	B(I/O)	R	P			9
BSFx	B(I/O)	RT	P			9
BSR	B(I/O)	R	P			9
BSRx	B(I/O)	RT	P			9
BST	B(I/O)	R	P			4
BTT	E	R	P			9
BTTx	E	NT	N			9
CAD	B	R	P			4
CAL	B	R	P		P	
CAQ	C	R	P	8		9
CAS	B	R	P		P	
CFF	E	N	N			
CHS	E	N	N			
CLA	B	R	P		P	
CLM	E	N	N			
CLS	B	R	P		P	
COM	E	N	N			
CPY	B	R	P			4
CPYD	A	R	N	R	P	9
CPYP	A	R	N	R	P	9
CRQ	C	R	P	8		9
CTL	K	R	N		P	9
CTLN	K	R	N		P	9
CTLR	K	R	N		P	9
CTLRN	K	R	N		P	9
CTLW	K	R	N		P	9
CTLWN	K	R	N		P	9
CVR	C	R	P	8		9
DCT	E	N	N			
DFAD	B	R	P		P	94
DFAM	B	R	P		P	94
DFDH	B	R	P		P	94
DFDP	B	R	P		P	94
DFMP	B	R	P		P	94
DFSB	B	R	P		P	94
DFSM	B	R	P		P	94
DLD	B	R	P		P	94
DRS	B	R	P			RPQ
DST	B	R	P		P	94
DUAM	B	R	P		P	94
DUFA	B	R	P		P	94
DUFM	B	R	P		P	94
DUFS	B	R	P		P	94
DUSM	B	R	P		P	94
DVH	B	R	P		P	
DVP	B	R	P		P	
EAD	B	R	P		P	RPQ
EAXM	E	N	N			RPQ
ECA	B	R	P		P	RPQ
ECQ	B	R	P		P	RPQ
ECTM	E	N	N			9

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
EDP	B	R	P		P	RPQ
EFTM	E	N	N			9
ELD	B	R	P		P	RPQ
EMP	B	R	P		P	RPQ
EMTM	E	N	N			94
ENB	B	R	P		P	9
ENK	E	N	N			9
ERA	B	R	P		P	9
ESB	B	R	P		P	RPQ
ESNT	B	R	P		P	9
EST	B	R	P		P	RPQ
ESTM	E	N	N			9
ETM	E	N	N			
ETT	E	N	N			4
ETTx	E	NT	P			9
EUA	B	R	P		P	RPQ
FAD	B	R	P		P	
FAM	B	R	P		P	9
FDH	B	R	P		P	
FDP	B	R	P		P	
FIVE	A	P	P	P		9
FMP	B	R	P		P	
FOR	A	P	P	P		
FOUR	A	P	P	P		9
FRN	E	N	N			9
FSB	B	R	P		P	
FSM	B	R	P		P	9
FVE	A	P	P	P		
HPR	B	N	N			
HTR	B	R	P		P	
ICC	K(1)	N	N			9
IIA	B	N	N			9
IIL	D	R				9
IIR	D	R				9
IIS	B	R	P		P	9
IOD	B(I/O)	N	N			4
IOT	E	N	N			9
*IOXY(N)	A	R	P	R	P	9
. . . (3)	A	P	P	P		
LAC	B	R	R			9
LAR	K	R	N	N	P	9
LAS	B	R	P		P	9
LAXM	E	N	N			RPQ
LBT	E	N	N			
LCC	K	R	N	N	P	9
LCHx	BT	R	P		P	9
LDA	B	R	P		P	4
LDC	B	R	R			9
LDI	B	R	P		P	9
LDQ	B	R	P		P	
LFT	D	R				9
LFTM	E	N	N			9
LGL	B	R	P			
LGR	B	R	P			9
LIP	K	N	N	N		9

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
LIPT	K	R	N	N	P	9
LLS	B	R	P			
LMTM	E	N	N			94
LNT	D	R				9
LRS	B	R	P			
LSNM	E	N	N			9
LTM	E	N	N			
LXA	B	R	R			
LXD	B	R	R			
MON	A	P	P	P		
MPR	B	R	P		P	
MPY	B	R	P		P	
MSE	E	P	P			
MTH	A	P	P	P		
MTW	A	P	P	P		
MZE	A	P	P	P		
NOP	B	N	N			
NTR	A	R	P	P		4
NZT	B	R	P		P	9
OAI	B	N	N			9
OFT	B	R	P		P	9
ONE	A	P	P	P		
ONT	B	R	P		P	9
ORA	B	R	P		P	
ORS	B	R	P		P	
OSI	B	R	P		P	9
PAC	B	N	R			9
PAI	B	N	N			9
PAX	B	N	R			
PBT	E	N	N			
PCA	B	N	R			94
PCD	B	N	R			94
PDC	B	N	R			9
PDX	B	N	R			
PIA	B	N	N			9
PON	A	P	P	P		
PSE	E	R	P			
PSLx	BT	R	P		P	RPQ
PTH	A	P	P	P		
PTW	A	P	P	P		
PXA	B	N	R			9
PXD	B	N	R			
PZE	A	P	P	P		
RCD	B(I/O)	P	P			
RCDx	B(I/O)	N	N			9
RCHx	BT	R	P		P	9
RCT	E	N	N			9
RDC	E	R	P			9
RDCx	E	NT	P			9
RDR	B(I/O)	R	P			4
RDS	B	R	P			
REW	B(I/O)	R	P			
REWx	B(I/O)	RT	P			9
RFT	D	R				9
RIA	B	N	N			9

*IOXY(N) symbolizes all I/O commands.

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
RICx	E	NT	P			9
RIL	D	R				9
RIR	D	R				9
RIS	B	R	P		P	9
RND	E	N	N			
RNT	D	R				9
RPR	B(I/O)	P	P			
RPRx	B(I/O)	N	N			9
RQL	B	R	P			
RSCx	BT	R	P		P	9
RTB	B(I/O)	R	P			
RTBx	B(I/O)	RT	P			9
RTD	B(I/O)	R	P			
RTDx	B(I/O)	RT	P			9
RTT	E	N	N			4
RUN	B(I/O)	R	P			9
RUNx	B(I/O)	RT	P			9
***(3)	A	P	P	P		
bbb(3)	A	P	P	P		
SAR	K	R	N	N	P	9
SBM	B	R	P		P	
SCA	B	R	R			94
SCD	B	R	R			94
SCDx	BT	R	P		P	9
SCHx	BT	R	P		P	9
SDH	B(I/O)	R	P			9
SDHx	B(I/O)	RT	P			9
SDL	B(I/O)	R	P			9
SDLx	B(I/O)	RT	P			9
SDN	B(I/O)	R	P			9
SEVEN	A	P	P	P		9
SIL	D	R				9
SIR	D	R				9
SIX	A	P	P	P		
SLF	E	N	N			
SLN	E	R	P			
SLQ	B	R	P		P	
SLT	E	R	P			
SLW	B	R	P		P	
SMS	K	R	N		P	9
SNS	K	N	N			9
SPR	E	R	P			
SPRx	E	R	P			9
SPT	E	N	N			
SPTx	E	N	N			9
SPU	E	R	P			
SPUx	E	R	P			9
SSLx	BT	R	P		P	RPQ
SSM	E	N	N			
SSP	E	N	N			
STA	B	R	P		P	
STCx	BT	N	N			9
STD	B	R	P		P	
STI	B	R	P		P	9
STL	B	R	P		P	9
STO	B	R	P		P	

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
STP	B	R	P		P	
STQ	B	R	P		P	
STR	A	N	N	N		9
STT	B	R	P		P	9
STZ	B	R	P		P	
SUB	B	R	P		P	
SVN	A	P	P	P		
SWT	E	R	P			
SXA	B	R	R			9
SXD	B	R	R			
TCH	A	R	P	N	P	9
TCM	K(1)	R	N	R	P	9
TCNx	BT	R	P		P	9
TCOx	BT	R	P		P	9
TDC	K	R	N	N	P	9
TEFx	BT	R	P		P	9
THREE	A	P	P	P		9
TIF	B	R	P		P	9
TIO	B	R	P		P	9
TIX	A	R	R	R		
TLQ	B	R	P		P	
TMI	B	R	P		P	
TNO	B	R	P		P	
TNX	A	R	R	R		
TNZ	B	R	P		P	
TOV	B	R	P		P	
TPL	B	R	P		P	
TQO	B	R	P		P	
TQP	B	R	P		P	
TRA	B	R	P		P	
TRCx	BT	R	P		P	9
TRS	B	R	P			RPQ
TSX	B	R	R			
TTR	B	R	P		P	
TWO	A	P	P	P		
TWT	K	R	N	N	P	9
TXH	A	R	R	R		
TXI	A	R	R	R		
TXL	A	R	R	R		
TZE	B	R	P		P	
UAM	B	R	P		P	9
UFA	B	R	P		P	
UFM	B	R	P		P	
UFS	B	R	P		P	
USM	B	R	P		P	9
VDH	C	R	P	8		9
VDP	C	R	P	8		9
VLM	C	R	P	8		9
WDR	B(I/O)	R	P			4
WEF	B(I/O)	R	P			
WEFx	B(I/O)	RT	P			9
WPB	B(I/O)	P	P			
WPBx	B(I/O)	N	N			9
WPD	B(I/O)	P	P			
WPDx	B(I/O)	N	N			9

Op Code	Type	Addr	Tag	Dec	Ind Addr	Mode
WPR	B(I/O)	P	P			
WPRx	B(I/O)	N	N			9
WPU	B(I/O)	P	P			
WPUx	B(I/O)	N	N			9
WRS	B	R	P			
WTB	B(I/O)	R	P			
WTBx	B(I/O)	RT	P			9
WTD	B(I/O)	R	P			
WTDx	B(I/O)	RT	P			9
WTR	A	R	N	N	P	9
WTS	B(I/O)	N	N			4
WTV	B(I/O)	N	N			
XCA	B	N	N			9
XCL	B	N	N			9
XEC	B	R	P		P	9
XIT	B	R	P		P	
XMT	A	R	N	R	P	9
ZAC	B	N	(2)			
ZET	B	R	P		P	9
ZSA	B	R	(2)			9
ZSD	B	R	(2)			

Disk File Orders

The following disk file orders are included in the combined operations table. The symbolic order should be written as follows:

LOC DORD A,T,R

The constituents of the symbolic disk order are:

1. A symbol or blanks in the location field;
2. The disk order code in the operation field;
3. One through three subfields in the variable field.

The first subfield of the variable field is a two-digit number representing the access and module (A).

The second subfield of the variable field is a four-digit number representing the track (T).

The third subfield of the variable field is two alphabetic or numeric characters representing the record (R).

The variable field will assemble as two, six, or eight consecutive BCD characters. The operation code is two BCD characters. Thus, the disk order consists of up to ten BCD characters. Since each BCD character consists of six-bits, a disk order requires two consecutive binary words in core storage. Unused portions of the binary words will contain 12_8, the disk BCD representation for zero.

Mnemonic	Order Code	Access Module	Track	Record	Definition
DNOP	00				No Operation
DREL	04				Release
DEBM	08				Eight-Bit Mode
DSBM	09				Six-Bit Mode
DSEK	80	R	R		Seek
DVSR	82	R	R	R	Prepare to Verify Single Record
DWRF	83	R	R		Prepare to Write Format
DVTN	84	R	R	R	Prepare to Verify Track, No Address
DVCY	85	R	R	R	Prepare to Verify Cylinder Operation
DWRC	86	R	R	R	Prepare to Write Check
DSAI	87	R			Set Access Inoperative
DVTA	88	R	R	R	Prepare to Verify Track With Address
DVHA	89	R	R		Prepare to Verify Home Address

Hypertape Orders

The following Hypertape orders are included in the combined operations table. The symbolic order should be written as follows:

LOC HORD A

The constituents of the symbolic Hypertape order are:

1. A symbol or blanks in the location field;
2. The Hypertape order code in the operation field; and
3. The Hypertape drive number in the variable field.

The variable field will be blank in all orders except HSBR and HSEL.

FAP will assemble Hypertape orders left-justified, one order per word, and the unused portion will contain 12_8, the Hypertape BCD representation for zero.

Mnemonic	Order Code	Drive	Definition
HNOP	00		No Operation
HEOS	01		End of Sequence
HSEL	06	R	Select
HSBR	07	R	Select for Backward Reading
HCCR	28		Change Cartridge and Rewind
HRWD	30		Rewind
HRUN	31		Rewind and Unload Cartridge
HERG	32		Erase Long Gap
HWTM	33		Write Tape Mark
HBSR	34		Backspace
HBSF	35		Backspace File
HSKR	36		Space
HSKF	37		Space File
HCHC	38		Change Cartridge
HUNL	39		Unload Cartridge
HFPN	42		File Protect On

APPENDIX B: THE FAP BCD CHARACTER CODE

The FAP BCD character code is identical to the FORTRAN character code except that the FAP apostrophe is replaced by a redundant minus sign, or dash, in the FORTRAN character code.

Character	BCD Code (Octal)	Card Code
(blank)	60	(blank)
0	00	0
1	01	1
2	02	2
3	03	3
4	04	4
5	05	5
6	06	6
7	07	7
8	10	8
9	11	9
A	21	12-1
B	22	12-2
C	23	12-3
D	24	12-4
E	25	12-5
F	26	12-6
G	27	12-7
H	30	12-8
I	31	12-9
J	41	11-1
K	42	11-2
L	43	11-3
M	44	11-4
N	45	11-5
O	46	11-6
P	47	11-7
Q	50	11-8
R	51	11-9
S	62	0-2
T	63	0-3
U	64	0-4
V	65	0-5
W	66	0-6
X	67	0-7
Y	70	0-8
Z	71	0-9
+ (plus)	20	12
- (minus)	40	11
/ (slash)	61	0-1
= (equals)	13	8-3
' (apostrophe)	14	8-4
. (period)	33	12-8-3
) (right parenthesis)	34	12-8-4
$ (dollar sign)	53	11-8-3
* (asterisk)	54	11-8-4
, (comma)	73	0-8-3
((left parenthesis)	74	0-8-4

APPENDIX C: SYSTEM SYMBOL TABLE, FORTRAN MONITOR

The following is a list of symbols that can be defined with the use of the SST pseudo-operation for the assembler operating under the FORTRAN Monitor.

Since the octal location of the following symbols will change when FAP is reassembled, these locations will not be included here. Other symbols which may be included in the System Symbol Table are given in Appendix D.

Core Allocation Symbols

TOPMEM	Top of available core storage
BOTIOP	Bottom of I/O package
BOTTOM	Bottom of available core storage
(PCBK)	COMMON break,, object program program break
DATEBX	Monitor date cell
LINECT	Monitor job line count,, FORTRAN page number, label flag in prefix
FLAGBX	Monitor flag cell

Tape Assignment Symbols

SYSTAP	00001	System tape
FINTAP	00002	First intermediate tape
SINTAP	00003	Second intermediate tape
TINTAP	00004	Third intermediate tape
MINTAP	00005	Monitor input tape
MLSTAP	00006	Monitor listing tape
MBNTAP	00007	Monitor punch tape
MCHTAP	00010	Monitor intermediate chain tape
SNPTAP	00011	Snap tape
LIBTAP	00001	System library tape

Entry Points to I/O Package

(LOAD)	Call next record on system tape
(DIAG)	Call diagnostic record, source or machine error
(TAPE)	Initiate tape operation
(PRNT)	Initiate on-line print
(PNCH)	Initiate on-line punch
(READ)*	Initiate on-line card read
(STAT)	Locate tape statistic tables
(REST)	Restore console
(STDN)	Set tape density
(SECL)	Call source program error record
(MECL)	Call machine error record
(DGLD)	Restore memory and halt

* FAP under the independent FORTRAN Monitor only.

** FAP under the FORTRAN Monitor operating under IBSYS only.

Parameters Variable at Edit Time

(ES1S)*	Physical sense switch corresponding to END Card Setting 1
(ES2S)*	Physical sense switch corresponding to END Card Setting 2
(ES3S)*	Physical sense switch corresponding to END Card Setting 3
(ES4S)*	Physical sense switch corresponding to END Card Setting 4
(ES5S)*	Physical sense switch corresponding to END Card Setting 5
(ES6S)*	Physical sense switch unassigned (See FORTRAN Reference Manual)
(PGCT)	Listing page dimensions
(LIBT)	System library tape assignment

Common Communications Region

(FGBX)	Location of Monitor flag cell
(LNCT)	Location of Monitor line count
(DATE)	Job date
(SNCT)	Snapshot tape file count
(MSLN)	Flag for diagnostic record
(ENDS)	End card setting
(SCHU)	Data transmission error information
(LODR)**	Entry point and record number of current record
(LBLD)**	Load address for relocatable library

Definitions of Operation Mnemonics to Initiate Tape Operation

	Definition	Operation	Information	End File	Check
(WROW)	40031	Write	Row Binary		Immediate
(RDEC)	40016	Read	Decimal	Permitted	Immediate
(WEFC)	40015	Write	End File		Immediate
(RBEC)	40014	Read	Binary	Permitted	Immediate
(WDNC)	40013	Write	Decimal		Immediate
(RDNC)	40012	Read	Decimal	Prohibited	Immediate
(WBNC)	40011	Write	Binary		Immediate
(RBNC)	40010	Read	Binary	Prohibited	Immediate
(RDEP)	40006	Read	Decimal	Permitted	Later
(WEFP)	40005	Write	End File		Later
(RBEP)	40004	Read	Binary	Permitted	Later
(WDNP)	40003	Write	Decimal		Later
(RDNP)	40002	Read	Decimal	Prohibited	Later
(WBNP)	40001	Write	Binary		Later
(RBNP)	40000	Read	Binary	Prohibited	Later
(SKDC)	20012	Skip	Decimal		Immediate
(SKBC)	20010	Skip Backspace	Binary		Immediate for BTT
(SKDP)	20002	Skip	Decimal		Later
(SKBP)	20000	Skip Backspace	Binary		Later Not significant
(CHKU)	10000	Delay and check last information transmitted			
(TPER)	04000	Error return for transmitted information found improper			
(SNAP)	01000	Dump panel and core storage selectively on SNPTAP			
(SUAV)**		Set unit(s) available			
(SUNV)**		Set unit(s) unavailable			

APPENDIX D: THE ASSEMBLE ONLY MODE OF FAP OPERATING UNDER THE BASIC MONITOR

The FORTRAN Assembly Program (FAP) is available in two modes when operating under the Basic Monitor (IBSYS). In addition to the normal FORTRAN Monitor mode, a special mode called IBSFAP is available which permits assembly and updating, but not execution. Complete IBSYS specifications are given in the reference manual IBM 7090/7094 Operating Systems, Basic Monitor (IBSYS), Form C28-6248.

IBSFAP OPERATIONS

Control Cards

The following Basic Monitor card sets the IBSFAP mode of the FORTRAN Monitor:

```
1             16
$EXECUTE      IBSFAP
```

The FORTRAN Monitor operating in the IBSFAP mode recognizes all standard Monitor control cards except those pertaining to execution. The asterisk, which defines a card as being a Monitor control card, may be in card column 1 or card column 7 when the FORTRAN Monitor is in the IBSFAP mode.

Complete details regarding FORTRAN Monitor operations are in the publication IBM 7090/7094 Programming Systems: FORTRAN II Operations, Form C28-6066-4.

System Symbol Table

When FAP is operating under the Basic Monitor, the SST pseudo-operation provides the symbols listed in items 1, 2, and 3 in the following text. In the FORTRAN Monitor mode, these symbols are provided in addition to those symbols given in Appendix C; in the IBSFAP mode, they are provided instead of those symbols in Appendix C. The exception, which overrides this rule, is when SST has a variable field (see the section entitled "The SST Pseudo-Operation" in Chapter 12). The additional symbols provided are the IBNUC and IOEX symbolic definition entries, mentioned in the reference manual IBM 7090/7094 Operating Systems, Basic Monitor (IBSYS), Form C28-6248, as follows:

1. All the one-word entry points starting with SYSTRA through SYSTWT in IBNUC, plus the current value of SYSEND and SYSORG.
2. All the SYSUNI functions.
3. The communication region to IOEX, from ACTIV to TRPSW.

APPENDIX E: FIELD MODIFICATIONS TO FAP

Certain modifications to FAP as distributed may be desired at individual installations. The following are among those which may be useful. While these are indicated for the convenience of installations, it should be emphasized that there is no assurance that the ability to make such modification will be continued. If at any time Programming Systems finds it necessary to make modifications to FAP as distributed which will prevent these modifications, it will do so with no prior notice.

Since the octal locations of the following symbols will change when FAP is reassembled, these locations will not be included here.

1. To permit on-line monitoring of the off-line listing tape:

```
ORG     WRITS
SWT     X
```

Sense Switch X may be toggled.

2. To permit operator termination, rather than continuation of assembly, following an on-line non-critical error message and programmed pause:

```
ORG     SPACS
SWT     Y
```

The machine error record will be called. Sense Switch Y may be toggled.

3. To avoid printing sequence error messages on-line:

```
ORG     ORDRS
PZE     ORDBF,,18
```

4. To permit HTR to be assembled as a Type A instruction:

```
ORG     IFL01+1
LFT     077700
```

5. To force binary output for relocatable assembly in error:

```
ORG     ERRRS
NOP
```

6. To add to or amend the Combined Operations Table:

Reassemble source card numbers F0D70000-F0D82900 to obtain binary card numbers 9F04OP00-9F04OP56. Note that definition cards for pseudo-operation transfer cards and ORG will have to be provided. Replace the distributed binary cards so serialized.

An alternate method would be to insert the required correction cards using an update and assemble run of the entire FAP assembler.

7. To add to or amend the System Symbol Table:

Reassemble all source cards serialized with FOD9 in columns 73-76 to obtain binary cards with serialization beginning with 9FO4SS00. Note that definition cards and ORG will have to be provided. Replace the distributed binary cards so serialized.

An alternate method would be to insert the required correction cards using an update and assemble run of the entire FAP assembler.

8. To provide Flow Trace Facility:

Remove and discard binary card 9F04FLOW. See Standard Error Procedure, page 32.

9. With regard to available tape units and channels, the following are assembly parameters:

T - The maximum number of logical tapes in the system.

C - The maximum number of data channels in the system.

T and C are contained in the addresses of SYSAST and CHANS, respectively.

The number of tapes reserved for the Monitor is contained in the address of symbolic location SYSTPS.

10. Currently an on-line message is given if there are more than 40 sequence errors (card columns 73-80). To increase the allowable limit, change the addresses of ORDR2 and ORDR3-1 from 40 to the desired limit.

APPENDIX F: TABLE LIMITS

The tables used by the FAP assembler are subject to the following limitations:

1. The number of different data words generated by literals may not exceed 1,000.
2. The number of transfer vector names may not exceed 250.
3. The number of symbol names permitted is approximately 10,000.
4. Only 150 undefined symbols will be accumulated in the symbolic reference table.

An overflow of the undefined symbol table, or of the symbolic reference table, will cause the cessation of collection of such symbols or references.

5. The length of the Combined Operations Table must not exceed 1,024 operation codes. Approximately 400 names are available to the programmer to insert macro-operation codes, or codes defined by OPD, OPSYN, or OPVFD pseudo-operations.
6. Macro-definitions share core storage with the symbol table; entries in one reduce the space available for the other.
7. There may be no more than 63 arguments in a macro-definition heading card argument list.
8. The depth of nesting of remote sequences is limited as follows: each macro-instruction within a remote sequence, or conversely, each remote sequence generated by a macro-instruction, requires three locations in the Level Table. The total length of this table is 112 locations, and the most severe use of this table, an alternate nesting of macro-instructions and remote sequences, will terminate assembly if there are more than 37 such nestings.

INDEX

ABS 37
Absolute
  assembly 2
  output 40
  symbols 4
Additional mnemonics 14
Additional pseudo-operations 50
Alphameric
  data items 6
  literals 6
Apostrophe (') 45
Argument list 42
  extending the 44
Argument strings 47
Assembly
  absolute 2
  listing 2, 61
  process 60
  relocatable 1
Assembly information pseudo-operations 18
Asterisk (*)
  as an element 4, 29
  in indirect addressing 9
  remarks 11

BCD 27
BCI 27
BES 24
Binary-place part of a fixed-point number 5
Binary output from the assembler 40
Blocked update output tape 52
BOOL 8, 20
Boolean expressions 8, 28
BSS 23
BSS Control 31, 60

CALL 32, 59, 62
Card format-control pseudo-operations 36
Closed subroutines 58
Column binary
  output 40
  transfer card 40
Combined operations table 63
Comments field 1
COMMON 24
  statement 59
Common
  break 24
  counter 24
  storage 59
  symbols 4
COUNT 18
Created symbols 49

Data-generating pseudo-operations 26, 61
Data items 5, 28
  alphameric 6
  decimal 5, 26
  octal 5
DEC 26
Decimal data items 5, 26
Decimal literals 6
Decimal integers 5
  fixed-point 5
  floating-point 5

DELETE 54
DETAIL 38
Disk file orders 67
  variable field of 10
Dollar sign ($) 21, 22, 30, 31, 33, 34, 47
DUP 30

EJECT 38
Elements 4
  of the language 1
END 19
ENDFIL 54
ENDUP 56
ENTRY 31
EQU 20
Equal sign (=) 6, 47
Error-tracing routine 32
ETC 29
EVEN 25
Excessive fields 3
Exponent part
  of a fixed-point number 5
  of a floating-point number 5
Expressions 7
  Boolean 8
  evaluation of 7
  symbolic 4
  types of 7
Extending the argument list 44
EXTERN 35, 62

FAP BCD character code 68
FAP operating under Basic Monitor (IBSFAP) 70
Field modifications to FAP 71
Fields
  comments 1
  excessive 3
  identification 1
  location 1, 9
  operation 1, 9
  variable 1, 10
First card group 3, 18
Fixed-point numbers 5
Flags 2
Floating-point numbers 5
FORTRAN linkages and calling sequences 59
FORTRAN Monitor control cards 41
FUL 37
Full output 41
FUNCTION subprograms 59

Generated instructions 48

HEAD 21
HED 23
Heading character 21
Hypertape orders 67

IBSFAP 70
Identification field 1
IFEOF 34
IFF 19
IGNORE 54
Illegible input instructions 53

INDEX 39
Indirect addressing 9
IRP 50

LABEL 39, 41
Label (FORTRAN Monitor) 41
LBL 39
Linkage director 2, 12, 33
Linkages 58
LIST 38
List-control pseudo-operations 37
Listing
  assembly 2
  update pseudo-instructions 53
Literals 62
  alphameric 6
  decimal 6
  octal 6
  table 62
Load address 1
LOC 25
Location counter 12, 23, 61
Location field 1, 9
  of a prototype 45

MAC 47
Machine instructions 12, 63
  extended 12, 63
Machine operation code definition 36
MACRO 44
  alternative form of 44
Macro-definition 42
  heading card 42, 44
  nesting 46
Macro-instruction 42, 46
  punctuation in 47
  nested 48
Macro-operation 42
MAX 21
MIN 21
MOP 44
Multiply-defined symbols 3
Multiply-headed region 21

Nesting
  macro-definitions 46
  macro-instructions 48
9LP 37
9LP output 41
NOCRS 49
NULL 40
NUMBER 54

OCT 26
Octal data items 5
Octal literals 6
OPD 35
Open subroutines 58
Operation code-defining pseudo-operations 35
Operation field 1, 9
  of a prototype 45
Operators 4, 7
  Boolean 9
OPSYN 35
Optional serialization in update pseudo-instructions 56
OPVFD 35
ORG 25
ORGCRS 49
Output
  column binary 40
  full 41
  9LP 41
  row binary 40

Page
  heading 3
  title card 3
PCC 39
Phase error 3, 18
Phase relocation error 3, 18
PMC 40
Prefix codes 14
Previously defined symbols 18
Principle part
  of a fixed-point number 5
  of a floating-point number 5
PRINT 56
Program
  break 1, 2
  card 31, 61
  counter 23, 60
Prototype 45
  instruction 42
  location field of 45
  operation field of 45
  variable field of 45
Pseudo-operations 18
  additional 50
  assembly information 18
  card format-control 36
  data-generating 26
  list-control 37
  operation code-defining 35
  program-linking 31
  storage-allocating 23
  symbol-defining 20
  update 53

Read Select 14
REF 39
Relocatable
  assembly 1
  binary format 60
  output 40
  symbols 4
Relocation
  constant 60
  error 3
  indicator bits 2, 7, 60
REM 37
Remarks card 11
Remote sequence 40, 51
REWIND 55
RMT and RMT* 50
Row binary
  output 40
  transfer card 40

Segmentation 59
Select and related operations 14

Read Select 14
Write Select 14
Sense operations 12
Sequence checking 1,52,56
Serialization 39, 52
FORTRAN Monitor 41
in update pseudo-instructions 56
SET 20,30
704 35
7090 35
SKIPTO 54
SKIPFIL 55
SPACE 38
SST 21
Standard error procedure option 32
Storage-allocating pseudo-operations 23, 61
Subroutine reference using the $ character 33
SUBROUTINE subprograms 59
Subroutines 58
Symbolic
expressions 4
instructions 1, 9
Symbolic card format 1
Symbolic Reference Table 2, 61
Symbol-defining pseudo-operations 20, 61
Symbols 4
absolute 4
common 4
created 49
definition 4
multiply-defined 3
relocatable 4
types of 4
SYN 20
System Symbol Table 4, 21
FORTRAN Monitor 69

Table limits 72
TAPENO 21
Tape positioning 52
Tape unit designator 21
TCD 37
Terms 4, 7
TITLE 38
Transfer card 40
Transfer vector 2,12, 31, 60
TTL 39

UMC 55
Undefined
operation code 3
symbol 3
UNLIST 38
UNLOAD 55
UPDATE 53,55
Update examples 56
Update pseudo-operations 53
optional serialization in 56
Updating a FORTRAN source program deck 53
Updating symbolic tapes 52
Uses of the update facility 52

Variable-channel tape operations 16
Variable field 1, 10
of a disk order 10
of a prototype 45
VFD 28

Write Select 14

XXX 48

C28-6235-3

Printed in U.S.A. C28-6235-3

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601