IBM 7090/7094 IBSYS Operating System

Version 13

Macro Assembly Program (MAP) Language

This publication provides detailed information for writing source programs in the
7090/7094 Macro Assembly Program (MAP) Language.

Users of the MAP symbolic programming language are provided with an exten-
tive set of pseudo-operations, as well as all 7090/7094 machine operations.

The Macro Assembly Program (7090-SP-804), IBMAP, is a component of the
7090/7094 ibsys Operating System: Version 13, iPjob Processor and operates under
the ijob Monitor.
The MAF language and its use in writing 7090/7094 programs are described in this publication. This symbolic language encompasses all 7090/7094 machine operations, extended machine operations, and special operations. In addition, MAF provides more than sixty pseudo-operations, including the powerful macro-facility, all of which are described in this publication.

MAF language programs are processed by the 7090/7094 assembly program, IBM/MAF, which is a component of the 7090/7094 IBDP Processor and which operates under the IBDP Monitor. The facilities of ICS, FORTRAN, and COBOL are accessible to the MAF user.

To assist the user in making the most effective use of this flexible programming tool, basic information about the MAF language is provided in Part I of this publication. Its main features and capabilities are outlined, and the constituents of MAF symbolic instructions are explained.

The pseudo-operations provided by MAF have been divided into classes according to function. Most of the pseudo-operations are described in Part II, where their formats are shown and their use in programs is explained and demonstrated.

The macro-facility is described separately in Part III.

Five appendixes following Part III provide supplementary information related to the MAF language.

It has been assumed that the reader is familiar with the contents of one of the following publications:
- IBM 7090 Principles of Operation, Form A22-6528
- IBM 7094 Principles of Operation, Form A22-6703

The following related publications may also be useful, depending on individual interests and requirements:
- IBM 7090/7094 IBSYS Operating System: Version 13, System Monitor (IBSYS), Form C28-6248
- IBM 7090/7094 IBSYS Operating System: Version 13, IBJOBO Processor, Form C28-6389
- IBM 7090/7094 IBSYS Operating System: Version 13, Input/Output Control System, Form C28-6345
- IBM 7090/7094 IBSYS Operating System: Version 13, FORTRAN IV Language, Form C28-6390
- IBM 7090/7094 IBSYS Operating System: Version 13, COBOL Language, Form C28-6391

Machine requirements for MAF language programs are given in the publication IBM 7090/7094 IBSYS Operating System: Version 13, IBJOBO Processor, Form C28-6389.

This publication, Form C28-6392-0, updates the material in Forms C28-6311-3 and N28-0128 for users of Version 13 of the IBSYS Operating System.

The principal changes in this edition are the addition of the LOC pseudo-operation to the Location-Counter group and the addition of a new group, Literal-Positioning Pseudo-Operations, which comprises the LORG pseudo-operation and a new LITORP pseudo-operation. Several miscellaneous changes have also been made. All changes are identified by a vertical line to the left of the changed material.

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, 1271 Avenue of the Americas, New York, N. Y. 10020

© 1963 by International Business Machines Corporation
## Contents

### Symbolic Programming Using MAP

7090/7094 MAP Language Features

- Operations .................................................. 5
- Macro-Operations ........................................ 6
- Location Counters ....................................... 6
- Absolute and Relocatable Assemblies ............... 6
- Error Checking ........................................... 6
- Forming Symbolic Instructions ..................... 6
- Instruction Fields ................................. 6
- Symbols .................................................. 8
- Defining Symbols .................................... 8
- Ordinary Symbols ................................... 8
- Immediate Symbols ................................ 9
- Relocation Properties of Symbols ............... 9
- Literals .................................................. 9
- Writing Expressions ................................ 11
- Evaluating Expressions .............................. 11
- Rules for Forming Expressions ................. 11
- Boolean Expressions ................................. 12
- Using Symbols in Expressions ................. 12
- Relative Addressing ................................ 12

### Map Pseudo-Operations

- Location-Counter Pseudo-Operations ............... 14
- The USE Pseudo-Operation .......................... 14
- The BEGIN Pseudo-Operation ....................... 14
- The ORG Pseudo-Operation .......................... 15
- The LOC Pseudo-Operation ........................ 15
- Storage-Allocation Pseudo-Operations ........... 15
- The ISS Pseudo-Operation .......................... 15
- The IES Pseudo-Operation .......................... 16
- The EVEN Pseudo-Operation ......................... 17
- The EDIR Pseudo-Operation ......................... 17
- The COMMON Pseudo-Operation ................. 17
- Literal-Positioning Pseudo-Operations .......... 18
- The LONG Pseudo-Operation ....................... 18
- The LITORG Pseudo-Operation .................... 18
- Date-Generating Pseudo-Operations ............... 18
- The OCT Pseudo-Operation .......................... 18
- The DEC Pseudo-Operation .......................... 19
- The RCI Pseudo-Operation .......................... 19
- The VFD Pseudo-Operation .......................... 19
- The LIT Pseudo-Operation ........................... 20
- The DUP Pseudo-Operation ........................... 21
- Symbol-Defining Pseudo-Operations ............... 21
- The EQV and SYM Pseudo-Operations .............. 22
- The NULL Pseudo-Operation .......................... 22
- The MAX Pseudo-Operation .......................... 23
- The MIN Pseudo-Operation ........................... 23
- The SET Pseudo-Operation ........................... 23
- Boolean Pseudo-Operations ......................... 24
- The BQUAL Pseudo-Operation ....................... 24
- The BCOORD and BCOORD Pseudo-Operations ....... 24
- Conditional-Assembly Pseudo-Operations ........ 24
- The IFT and IFF Pseudo-Operations ............. 24
- The CTOR Pseudo-Operation ......................... 25
- Symbol-Qualifying Pseudo-Operations ............. 25
- The QUAL Pseudo-Operation ......................... 25
- The ENDO Pseudo-Operation ......................... 26
- Control-Section Pseudo-Operations ............... 26
- The CONTRL Pseudo-Operation ..................... 27
- The ENTRY Pseudo-Operation ....................... 27
- File-Description Pseudo-Operations .......... 28
- The FILE Pseudo-Operation .......................... 28
- The LABEL Pseudo-Operation ....................... 31
- Operation-Defining Pseudo-Operations ............ 32
- The ORP Pseudo-Operation .......................... 32
- The OPVFD Pseudo-Operation ....................... 32

### Miscellaneous Pseudo-Operations

- The OPVFD Pseudo-Operation .......................... 32
- Miscellaneous Pseudo-Operations ................. 33
- The END Pseudo-Operation .......................... 33
- The ETC Pseudo-Operation .......................... 33
- The REM Pseudo-Operation .......................... 33
- The EEP Pseudo-Operation .......................... 34
- Absolute-Assembly Pseudo-Operations ............ 34
- The AHS Pseudo-Operation .......................... 34
- The FUL Pseudo-Operation .......................... 34
- The PUNCH and unpunch Pseudo-Operations ....... 34
- The TCD Pseudo-Operation .......................... 34

### List-Control Pseudo-Operations

- The FCC Pseudo-Operation .......................... 35
- The UNLIST Pseudo-Operation ...................... 35
- The LIST Pseudo-Operation .......................... 35
- The TITL Pseudo-Operation .......................... 35
- The DETAIL Pseudo-Operation ....................... 35
- The EJECT Pseudo-Operation ....................... 35
- The SPACE Pseudo-Operation ....................... 35
- The LBL Pseudo-Operation ........................... 36
- The INDEX Pseudo-Operation ....................... 36
- The PMC Pseudo-Operation ............................ 36
- The TTL Pseudo-Operation .......................... 37
- The FCG Pseudo-Operation ............................ 37
- Special Systems Pseudo-Operations ............... 37
- The CALL Pseudo-Operation ......................... 37
- Expansions of the CALL Pseudo-Operation ....... 38
- The SAVE Pseudo-Operation ........................... 38
- Expansions of the SAVE Pseudo-Operation ....... 38
- The SEVEN Pseudo-Operation ......................... 39
- The RETURN Pseudo-Operation ....................... 39

### The Macro-Operation Facility

- The MACRO Pseudo-Operation ......................... 41
- Defining Macro-Operations ......................... 41
- Prototypes in Macro-Definitions ................. 42
- The ENDM Pseudo-Operation .......................... 43
- Calling Macro-Operations ........................... 43
- The MACRO-Instruction ............................... 43
- Inserting Instructions into Macro-Expansions ... 45
- Conditional Assembly in Macro-Operations ....... 45
- Combining Substitutable Arguments and Text ..... 45
- Nested Macro-Operations ............................. 46
- Macro-Instructions in Macro-Definitions ......... 46
- Qualifications Within Macro-Operations .......... 46
- Macro-Related Pseudo-Operations ................. 47
- The HP Pseudo-Operation .............................. 47
- Created Symbols ............................ 47
- The NOCRS Pseudo-Operation ......................... 48
- The ONCRS Pseudo-Operation ......................... 48
- The PUBLISH Pseudo-Operation ...................... 48

### Appendixes

- Appendix A: Machine Operations .................... 49
  - 7090 Machine Operations ........................... 49
  - Extended Operations ............................... 50
  - Special Operations ................................. 51
  - 7094 Machine Operations ........................... 51
  - IBM 7090 Data Channel Commands ................. 51
  - IBM 1301 Disk and 7320 Drum File Orders ....... 52
  - IBM 7340 Hypertape Orders ....................... 52
- Appendix B: 7090 Macro-Expansions of 7094 Instructions .... 52
- Appendix C: Operation Code Formats ............... 54
- Operations ................................. 54
- Pseudo-Operations .............................. 54
- Adjective Codes ............................... 54
- Appendix D: IBMMAP-FAP Incompatibilities .......... 56
- Appendix E: The MAP BCAD Character Code ....... 57
Programmers can communicate instructions to computers at three general language levels. The language of the computer itself is the most basic. At the highest level are scientific and commercial programming languages, such as FORTRAN and COBOL, respectively. Assembly-program languages like the MAP (Macro Assembly Program) language are at the intermediate level.

Because the computer executes instructions at the machine-language level, a source program written at either of the other two levels must be reduced to a machine-language object program before it can be executed. Machine-language programming is theoretically the most efficient, since no translation from source program to object program is required. For the programmer, however, programming in machine language is tedious and time consuming, and programming errors are more likely.

A source program written in the FORTRAN language closely resembles the mathematical notation used to state a problem to be solved by traditional methods. The COBOL language is based on English statements much like those that would be used to explain a procedure. These languages are relatively easy to learn and to use because of their similarity to the ordinary languages of business and science.

Source programs written in these languages are translated into machine-language programs within the computer by a compiler program. By using a compiler, the computer can produce an efficient machine-language program from a FORTRAN or COBOL source program faster and more accurately than a programmer can. Such compiler languages thus offer marked advantages over machine-language programming. However, compiler languages are somewhat restrictive. Some programming features that are available when using machine language cannot be included in any present-day compiler.

An assembly-program language is similar in structure to machine language. However, mnemonic symbols are substituted for each binary instruction code, and symbols provided by the programmer are substituted for the other fields of an instruction. An assembly-program language can also provide additional advantages beyond machine-language programming. For example, pseudo-operations can be provided, which often permit the coding of one instruction instead of many instructions. Thus, an assembly program provides the programmer with all the flexibility and versatility of machine language but with greatly reduced programming effort. In addition, error checking can be included to facilitate source program debugging.

**7090/7094 MAP Language Features**

**Operations**

The 7090/7094 MAP (Macro Assembly Program) language can be used for all of the 7090/7094 machine operations, the extended machine operations, and the special machine operations. (All such operations recognized by MAP are listed in Appendix A with supplementary information about them.) In addition, the MAP language provides an extensive set of pseudo-operations that supplement machine instructions.

A pseudo-operation is any operation included in the MAP language that is not an actual machine operation, extended machine operation, or special machine operation. Pseudo-operations are used by the programmer in much the same way as machine operations. MAP provides more than sixty such pseudo-operations to meet a variety of programming needs. These pseudo-operations, which are described in detail in Parts II and III, have been divided into classes according to function.

**Location-Counter Pseudo-Operations** enable the programmer to establish symbolic location counters and control their operation.

**Storage-Allocation Pseudo-Operations** reserve areas of core storage.

**Data-Generating Pseudo-Operations** introduce data into a program in any of a variety of formats. They are also used in combination to generate tables of data.

**Symbol-Defining Pseudo-Operations** are used to assign specific values to symbols.

**Boolean Pseudo-Operations** define symbols as Boolean quantities.

**Conditional-Assembly Pseudo-Operations** base assembly of an instruction on programmer established criteria.

**Symbol-Qualifying Pseudo-Operations** qualify symbols within sections of a program.

**Control-Section Pseudo-Operations** delimit sections of a program, facilitating cross-referencing among programs and among program segments.

**File-Description Pseudo-Operations** define the requirements of input/output files used by the program.

**Operation-Defining Pseudo-Operations** define or redefine symbols as operation codes.
Miscellaneous Pseudo-Operations indicate the end of a program, extend the variable field of an operation, and permit remarks to be entered into the assembly listing.

Absolute-Assembly Pseudo-Operations specify the punched output format of an absolute assembly.

List-Control Pseudo-Operations specify the contents and format of an assembly listing.

Special Systems Pseudo-Operations generate subroutine calling sequences. They may also be used to save and restore the index registers and indicators.

Macro-Defining Pseudo-Operations are used to define programmer macro-operations. They are used in conjunction with the macro-related pseudo-operations, which extend the facilities of macro-operations.

Macro-Operations
The programmer macro-operation facility is a very flexible and powerful programming tool. Many programming applications involve a repetition of a pattern of instructions, often with parts of the instructions varied at each iteration. Using the macro-defining pseudo-operations, a programmer can define the pattern as a macro-operation.

In defining the pattern, the programmer gives it a name that becomes the operation code used to generate the pattern of instructions. Thus, the coding of a single instruction can cause the pattern of instructions to be repeated as often as desired. Moreover, parts of the instruction can be varied each time the sequence is repeated. The contents of any field of any instruction within the pattern may be varied, and even entire instructions can be inserted in the sequence. The macro-operation facility is described in Part III.

Location Counters
During assembly, a location counter registers the next location to be assigned to an instruction. For most machine instructions processed by the assembly program, the contents of the location counter in effect at that time (the “current” location counter) is increased by 1. Some pseudo-operations may result in no increase, an increase of 1, or an increase of more than 1.

MAP enables a programmer to create and control as many symbolic location counters as he needs by using the location-counter pseudo-operations. Control can be transferred back and forth among them as often as desired.

This feature permits instructions coded in one sequence to be loaded in another, the establishment of constant tables, etc.

Absolute and Relocatable Assemblies
The control routines of the operating system occupy lower core storage. Therefore, a program may not be loaded into this area but must be loaded into the first unused machine location. However, the programmer need not know the address of this location, since the loader (MLDR) can automatically relocate each program segment to be loaded.

The first address of a program segment to be executed is called the load address, and each succeeding instruction is loaded relative to that address. Thus, the address of an instruction at load time is the address assigned to it during assembly plus the load address of the program segment in which the instruction appears.

In a relocatable assembly, the assembly program produces an object deck that is automatically relocated at execution time by MLDR. However, it may sometimes be desirable to load a program beginning at a certain fixed location in core storage. A program loaded in this way is said to have an absolute origin. The programmer specifies a certain location as the load address for that deck. (An absolute origin may also be specified within a relocatable assembly. See the section “Relocation Properties of Symbols.”)

In absolute assemblies, output is in the standard 22-word-per-card format, which is specified on the SIMAP control card and by the ABS pseudo-operation. Output in this format cannot be handled by MLDR. Whether the assembly is absolute or relocatable is specified by the programmer on the SIMAP card. (See the publication IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C28-6389.)

Error Checking
Source programs written in the MAP language are checked for a variety of errors, including format errors, table overflow errors, input/output errors, improper references, and incorrectly coded operations. In addition, the severity of the error is indicated.

In a normal assembly, messages are printed just after the assembly listing. All messages for a given card are printed together, and the card groups are printed in ascending sequence. Correlation with the listing is accomplished by printing the line number, which is assigned by the assembly program, in the left margin of the listing for each card that requires a message.

A list of MAP error messages and an explanation of the severity code used are included in the publication IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C28-6389.

Forming Symbolic Instructions
Instruction Fields
In the MAP language, each symbolic instruction is punched on a separate standard IBM card. A single
instruction may have as many as five parts, occupying five fields on the card.

**THE NAME FIELD**

An instruction may be given a symbolic name by the programmer, so that references may be made in other instructions to the named instruction. (Other methods are also available for referring from one instruction to another. For example, see the section "Relative Addressing.")

The use of a name is generally optional. However, some pseudo-operations do require a symbol in the name field. Name-field and other requirements of each of the pseudo-operations are explained in Parts II and III. Also, the specifications for symbols used in the name field are given in the section "Symbols."

The name given to a symbolic instruction is from 1 to 6 characters long, and it occupies columns 1 through 6 on the card.

**THE OPERATION FIELD**

The machine operation code, pseudo-operation code, programmer macro-operation code, or an operation code previously defined by one of the operation-defining pseudo-operations appears in the operation field.

The operation field is punched beginning in column 8. Column 7 separates the name field from the operational field. Column 7, which is usually left blank, is ignored by the assembly program.

The character asterisk (*) may be used immediately to the right of some operation codes to indicate indirect addressing. Those machine instructions that are indirectly addressable are indicated in Appendix A. If indirect addressing is specified for an instruction in which it is not permitted, the asterisk (*) is ignored and a low-severity error message is issued.

The operation field is usually restricted to a maximum of six characters. However, an operation code of six characters defined by one of the operation-defining pseudo-operations may be followed by an asterisk (*) to indicate indirect addressing.

**THE VARIABLE FIELD**

The variable field of a symbolic instruction may contain subfields, separated by commas.

In machine instructions, these subfields contain the address, tag, and/or decrement (or count) parts of instruction, depending on the requirements of the particular instruction. These parts of the variable field are supplied in the order: address, tag, decrement.

The subfields that are required, optional, or not permitted in the variable fields of all 7090/7094 machine instructions, extended machine operations, and special operations are indicated in Appendix A.

In pseudo-operations, the subfields of the variable field may contain symbols, symbolic expressions, and literals. The contents of the variable field specified for each of the MAP pseudo-operations is given in Parts II and III.

A null subfield is indicated as being present but as having no value. If a null subfield is at the beginning of the variable field, it is indicated by a single comma. If it is between two other subfields, it is expressed by two consecutive commas. A null subfield at the end of the variable field is represented by a single comma followed by a blank.

If a subfield that is not used in the variable field (an irrelevant subfield) is to be followed by a subfield that is required (a relevant subfield), the irrelevant subfield must be indicated. Irrelevant subfields at the end of the variable field may be indicated as null or may be omitted entirely. For example, the following pairs of instructions are equivalent:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TXH</td>
<td>.0,1</td>
<td></td>
</tr>
<tr>
<td>TXH</td>
<td>.1</td>
<td></td>
</tr>
<tr>
<td>IORB</td>
<td>ALPHA,0,1</td>
<td></td>
</tr>
<tr>
<td>IORB</td>
<td>ALPHA,1</td>
<td></td>
</tr>
<tr>
<td>CLA</td>
<td>ALPHA,0</td>
<td></td>
</tr>
<tr>
<td>CLA</td>
<td>ALPHA</td>
<td></td>
</tr>
<tr>
<td>TXH</td>
<td>ALPHAN,0,0</td>
<td></td>
</tr>
<tr>
<td>TXH</td>
<td>ALPHAN,</td>
<td></td>
</tr>
<tr>
<td>FXA</td>
<td>.0</td>
<td></td>
</tr>
<tr>
<td>FXA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the last two pairs, the commas may not be omitted, since the assembly program checks for a minimum number of subfields. The TXH instruction requires three subfields, while the FXA instruction requires two. These subfields are not irrelevant and must be included.

The variable field is separated from the operation field by at least one blank column. The variable field may begin in column 12 but may never begin after column 16. The variable field cannot extend beyond column 72. An instruction having a variable field extending into column 72 may not have a comments field. However, the variable field of most instructions may be extended over more than one card, each having its own comments field, by using the EOR pseudo-operation.

**THE COMMENTS FIELD**

The comments field is included for the convenience of the programmer and does not affect execution of the program. This field is generally used for explanatory remarks. (See also the section "The Asterisk (*) Remarks Cards.")

A blank precedes the comments field to separate it from the variable field. This field extends through column 72 on the card. If there is a blank variable field, the comments field may begin as soon as column 17. An example of the use of the comments field is shown in Figure 1.
THE SEQUENCE FIELD

Symbolic instructions may be numbered for identification in the sequence field, which includes columns 73 through 80 on the card.

THE ASTERISK (*) REMARKS CARD

The remarks card is a special source card with an asterisk (*) in column 1 and any desired information in the rest of the card.

Any card with an asterisk in column 1 is treated by the assembly program as a remarks card, and its contents are printed out in the assembly listing. It has no other effect on the assembly.

Remarks cards may be grouped and may appear anywhere in a program except in macro-operations or between ETU cards. They are frequently used at the beginning of a program to state the problem to be solved, to describe the technique used, etc. (MAP also makes a remarks card available by using the pseudo-operation REM, which is described in the section “The REM Pseudo-Operation.”)

Examples of remarks cards are shown in Figure 1, as well as methods of coding a variable field.

Symbols

The symbolic names used in the name and variable fields of symbolic instructions consist of one to six nonblank BCD characters (see Appendix E). At least one nonnumeric character must be included, but none of the following ten characters may be used:

+ (plus sign) = (equal sign)
- (minus sign) ; (comma)
* (asterisk) ' (apostrophe)
/ (slash) ( left parenthesis)
$ (dollar sign) ) (right parenthesis)

Parentheses in a symbol cause a low-severity warning message to be printed, but assembly is not affected. However, the ()OK option specified on the SHMAP control card (see the publication IBM 7090/7094 IBSYS Operating System: IBMOP Processor, Form: C28-6389) indicates that parentheses in symbols are desired, and no message will be printed.

Examples of valid symbols are:

A 3.2XY
37B2 DECLOC
DELTA

Conversely, the following are not valid symbols for the reasons indicated:

A+B (invalid character)
3921 (no nonnumeric characters)
A2B4C6D (more than six characters)

Defining Symbols

When a symbol has been assigned a value, it is said to be defined. The assigned value can be the address of a location within core storage, an arbitrary quantity specified by the programmer, or a dependent value assigned by the assembly program. Values are assigned to symbols during and after the first of the two passes made by the assembly program over the source program. Further information about the assembly program is provided in the publication IBM 7090/7094 IBSYS Operating System: IBMOP Processor. Form C28-6389.

Ordinary Symbols

Several types of symbols are used in the variable fields of machine instructions and in most of the pseudo-operations.

1. Location symbols are so called because of their appearance in the name field of an instruction. During the first pass of the assembly program, location symbols in the variable field of an instruction are immediately assigned a value called an S-value. The S-value is 1 if the symbol has previously appeared in the name field of an instruction and 0 if it has not. After the first pass has been completed, these symbols are assigned
the value of the address of the instruction in which they appeared as a name-field symbol.

Absolute symbols are location symbols having fixed values that are independent of any relocation of the program segment.

2. Virtual symbols are used in the variable field of an instruction and never appear in any name field. Virtual symbols, which have special functions in MAP, are defined at load time. The S-value is 0. (For further information, see the publication IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C28-6389.) Virtual symbols are permitted only in a relocatable assembly. In an absolute assembly, virtual symbols are flagged as undefined.

3. System symbols from the IBSYS, IOEX and IJMON communication regions are predefined by the assembler if a System Symbol Option (MNSYM or JBSYM) appears on the SBMAP control card. If symbols thus defined are used in the variable field of an instruction in the source program, they will be treated as absolute symbols appearing in the qualification section "S.S." The S-value is 1. (For further information concerning the use of the System Symbol options, and the symbols involved, see the publication, IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C28-6389.)

**Immediate Symbols**

Immediate symbols are created by using them in the name field of the SET pseudo-operation. Immediate symbols are assigned a value (S-value) during the first pass of the assembly program. Immediate symbols may also be redefined throughout a program by using additional SET pseudo-operations. (See the section "The SET Pseudo-Operation.") The final value of an immediate symbol is used in evaluating variable fields of pseudo-operations affecting location counters.

**Relocation Properties of Symbols**

An absolute origin may be specified in a relocatable assembly, which should not be confused with an absolute assembly. If an absolute origin is given in a relocatable assembly, any symbols whose definitions depend on that origin are absolute. However, instructions under the absolute origin may refer to symbols elsewhere in the program. The assembly can be returned to the relocatable mode by subsequently specifying a relocatable origin.

Under the following conditions, symbols are absolute even if they appear within a relocatable assembly:

1. Symbols whose values depend on an absolute origin (as a result of using the ORG or BEGIN pseudo-operations).

2. Symbols defined by the BOOL, RBOOL, and LBOOL pseudo-operations.

3. Symbols defined by the EQU or SYN pseudo-operations and whose values reduce to a constant.

4. Symbols defined by a MAX or MIN pseudo-operation that yield a constant.

5. Symbols used in the variable field of type D instructions.

6. Symbols defined by the SET pseudo-operation.

**Literals**

Literals provide a simple means for introducing data words and constants into a program. For example, if a programmer wishes to add the number 1 to the contents of the accumulator, he must have the number 1 at some location in storage.

In contrast to other types of subfields, the contents of a literal subfield is itself the data to be operated on. The appearance of a literal directs the assembly program to prepare a constant equal in value to the content of the literal subfield. The assembly program replaces the subfield of the variable field of the instruction containing the literal with the address of the constant thus generated.

All the literals used in a program are placed in the Literal Pool. Unless the programmer specifies another location (see "Literal-Positioning Pseudo-Operations"), the Literal Pool begins one location beyond the final value of the location counter in use at the end of the program.

There are three types of literals — decimal, octal, and alphabetic.

**Decimal Literals**

A decimal literal consists of the character = followed by a decimal data item. Three types of decimal data items are recognized by MAP:

- *Decimal Integers*. A decimal integer is one or more of the digits 0 through 9, and it may be preceded by a plus or minus sign. Maximum size of the decimal integer is $2^{35} - 1$.

- *Floating-Point Numbers*. A floating-point number has two components.

  1. The principal part is a signed or unsigned decimal number, which may be written with or without a decimal point. The decimal point may appear at the beginning, at the end, or within the decimal number. If an exponent part is present, the decimal point may be omitted, in which case it is assumed to be at the right end of the decimal number. The principal part cannot exceed twenty digits. If it does, the number will be truncated and only the first twenty digits will be used.

  2. The exponent part consists of the letters E or EE followed by a signed or unsigned decimal integer. (The letters EE indicate a double-precision floating-point number.) The exponent part may be omitted. If the exponent part is used, it must follow the principal
part. The exponent part cannot exceed two digits. If it does, it will be truncated and only the first two digits will be used.

A floating-point number is converted to a normalized floating-point binary word. The exponent part, if present, specifies a power of ten, by which the principal part is multiplied during conversion. For example, all of the following floating-point numbers are equivalent and are converted to the same floating-point binary number.

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

The octal representation of this number is

20262077174
47018606335

Similarly, the number .314159E1 is converted to a double-precision floating-point number. Its octal representation is

20262077174
147018606335

Fixed-Point Numbers. A fixed-point number has three components.

1. The principal part is a signed or unsigned decimal number, which may be written with or without a decimal point. The decimal point may appear at the beginning, at the end, or within the decimal number. If the decimal point is omitted, it is assumed to be at the right end of the decimal number. The principal part cannot exceed twenty digits. If it does, the number will be truncated and only the first twenty digits will be used.

2. The exponent part consists of the letters E or EE followed by a signed or unsigned decimal integer. (The letters EE indicate a double-precision fixed-point number.) The exponent part may be omitted. If the exponent part is used, it must follow the principal part. The exponent part cannot exceed two digits. If it does, it will be truncated and only the first two digits will be used.

3. The binary-place part consists of the letters B or BB followed by a signed or unsigned decimal integer. (The letters BB indicate a double-precision fixed-point number.) The binary-place part must be present in a fixed-point number and must come after the principal part. If the number has an exponent part, the binary-place part may either precede or follow the exponent part. The binary-place part may not exceed two digits. If it does, the number will be truncated and only the first two digits will be used.

A fixed-point number is converted to a fixed-point binary number that contains an understood binary point. The binary-place part specifies the location of this understood binary point within the word. The number that follows the letters B or BB specifies the number of binary places in the word at the left of the binary point. The sign bit is not counted. Thus, a binary-place part of zero specifies a 35-bit binary fraction. B2 specifies two integral places and 33 fractional places. B3 specifies a binary integer. B-2 specifies 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.

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

22.5B5
11.25B4
1125E-2B4
9B7E1

All of the above fixed-point numbers are converted to the binary configuration having the octal representation

26400000000

The following double-precision fixed-point numbers

10BB35
1B35EE1
1BB35EE1

are converted to the binary configuration having the octal representation

000000000012
000000000000

Double-precision literals are stored in consecutive locations. The first or high-order part is automatically stored in an even location relative to the beginning of the Literal Pool. If these literals are to be used as operands in double-precision operations (7094), an even pseudo-operation must be inserted immediately before the long pseudo-operation if there is one; otherwise it must be inserted before the end pseudo-operation.

OCTAL LITERALS

An octal literal consists of the two characters =O followed by an octal integer.

An octal integer is a string of not more than twelve of the digits 0 through 7, and it may be preceded by a plus or minus sign.

Examples of octal literals are:

=O123
=O+123
=O-123

ALPHAMERIC LITERALS

An alphanemic literal consists of the two characters =H followed by exactly six alphanemic characters.
The six characters following the H are treated as data even if one or more of them is a comma or a blank.

Examples of alphanumeric literals are:

=H12ABCD
=HTADbbb, where b represents a blank

Writing Expressions
The programmer writes expressions to represent the subfields of the variable field of symbolic instructions. Expressions are also used in the variable fields of many of the pseudo-operations in accordance with the rules set forth for each specific case.

Expressions are comprised of elements, terms, and operators.

ELEMENTS
An element is the smallest component of an expression and is either a single symbol or a single integer less than 2^8. The asterisk (*) may be used as an element representing the location of the instruction in which it appears.

Examples of valid elements are:

A
427
ALPHA

TERMS
A term is a group of one or more elements and the operators * (indicating multiplication) and / (indicating division).

A term consists of one or more elements, with each element separated by an operator. A term must begin and end with an element. Two operators or two elements in succession are never permissible.

Examples of valid terms are:

A
427
ALPHA
A*B

ALPHA/C/1409

There is no ambiguity between using the asterisk as an element and its use to denote multiplication, since position always makes clear its intended function. For example, a field coded

**B

would be interpreted as "the location of this instruction multiplied by B." Since a term must begin with an element, the first asterisk must be an element. The second asterisk must be an operator, which is required between two elements.

EXPRESSIONS
An expression is a group composed of one or more terms and the operators + (signifying addition) and − (signifying subtraction).

An expression consists of one or more terms, with each term separated by a plus or minus sign. Two operators or two terms in succession are never permissible. However, an expression may begin with a plus or minus sign. Examples of valid expressions are:

A
ALPHA
ALPHA*BETA
−A/B
A*B−C/D+E*2303
−A+B*C

The asterisk in the last example is used first as an element and then as an operator.

Evaluating Expressions
In evaluating expressions, elements are evaluated first, then individual terms, and finally the complete expression. The following procedure is used in evaluating expressions:

1. Each element is replaced with its numeric value.
2. Each term is evaluated by performing the indicated multiplications and divisions from left to right. In division, the integral part of the quotient is retained and any remainder is discarded immediately. For example, the value of the term 5/2*2 is 4.

3. In evaluating a term, division by zero is the same as division by one and results in the original dividend. Division by zero is not regarded as an error.
4. Terms are combined from left to right in the order in which they occur, with all intermediate results retained as 35-bit signed numbers.
5. Finally, if the result is negative, it is complemented; in either case, the rightmost 20 bits are retained if the expression is a subfield of a word pseudo-operation. Otherwise, only the rightmost 15 bits are retained.

Grouping of terms by parentheses or any other means is not permitted. However, a product such as A/(B/C) can be written simply A*B/A*C.

The expression ** may be used to designate a field the value of which is to be computed and inserted by the program. It is an absolute expression having a value of zero.

Rules for Forming Expressions
The use of expressions is sometimes affected by whether elements within the expression are relocatable, absolute, or a combination of both.

In a relocatable assembly, an expression that contains more than one symbol is generally complex. An expression that includes a control-section name is also complex. In addition, any expression that contains a complex element is itself complex.

In an absolute assembly, all expressions are absolute. Relocatable and complex expressions are usually evaluated at load time, when absolute values have been assigned to symbols as part of the loading process. However, in pseudo-operations that affect location counters (such as BP, BS, and BEGIN) or that define
symbols (such as \texttt{MAX} and \texttt{MIN}), the variable field must be evaluated before load time. For further information, see the publication \textit{IBM 7090/7094 IBSYS Operating System: IBJOB Processor}, Form C28-6339.

The rules that must be followed in using expressions are provided in the discussions of the pseudo-operations.

**Boolean Expressions**

A Boolean expression is evaluated as an 18-bit Boolean quantity, unlike the 15-bit integer that normally results from the evaluation of an expression. Elements within a Boolean expression must be constant. All integers are specified as octal integers. An expression is Boolean if:

1. It appears in the variable field of a Boolean pseudo-operation (\texttt{BOOL, BBOOL, LBOOL}; see the section “Boolean Pseudo-Operations”);
2. It appears as an octal subfield of a \texttt{VFD} pseudo-operation (see the section “The \texttt{VFD} Pseudo-Operation”); or
3. It forms the variable field of a Type D or extended Type D machine instruction (see Appendix A).

Relocatable symbols in a Boolean expression are evaluated as 15-bit integers and virtual symbols are evaluated as zero. In either case, an error message is issued.

In a Boolean expression, the four operators (+, −, *, /) have Boolean rather than arithmetic meanings, as shown in the table.

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>“inclusive or” (also, “union”)</td>
<td>0+0=0, 0+1=1, 1+0=1, 1+1=1</td>
</tr>
<tr>
<td>−</td>
<td>“exclusive or” (also, “symmetric difference”)</td>
<td>0−0=0, 0−1=1, 1−0=1, 1−1=0</td>
</tr>
<tr>
<td>*</td>
<td>“and” (also, “intersection”)</td>
<td>0<em>0=0, 0</em>1=0, 1<em>0=0, 1</em>1=1</td>
</tr>
<tr>
<td>/</td>
<td>“complement” (also, “not” or “ones’ complement”)</td>
<td>/0=1, /1=0</td>
</tr>
</tbody>
</table>

The four Boolean operations are defined in the table for 1-bit quantities. The operation is extended to 18-bit quantities by handling each bit position independently.

The following conventions also apply in using Boolean expressions:

The / is a unary or one-term operator. The expression \texttt{A/B} means \texttt{A} / \texttt{B}, and the \texttt{A} is ignored. However, the presence of the \texttt{A} is not regarded as an error by the assembly program. The definitions of / in this case are:

\[
\begin{align*}
0/0 &= 1 \\
0/1 &= 0 \\
1/0 &= 1 \\
1/1 &= 0
\end{align*}
\]

If the other operators (+, −, *) are used as unary operators, the definitions are as follows:

\[
\begin{align*}
+\texttt{A} &= \texttt{A} + \texttt{A} \\
-\texttt{A} &= \texttt{A} - \texttt{A} \\
*\texttt{A} &= \texttt{A} * \texttt{A} = \texttt{A} \texttt{A} = 0
\end{align*}
\]

For the special case of the slash, the definition is:

\[
/\texttt{A} = \texttt{A} / 0 = 1
\]

In expressions where both terms are missing, definitions are as follows:

\[
\begin{align*}
+0 + 0 &= 000000 \\
-0 - 0 &= 000000 \\
* &= \text{Location counter} \\
/0 &= 777777
\end{align*}
\]

In evaluating a Boolean expression, all integers are treated as 18-bit quantities. The operation / is performed first, followed by *, then by + and −.

The operators +, −, and * may immediately precede the slash in a Boolean expression. For example,

\[
\texttt{A} - / \texttt{B}
\]

is a valid Boolean expression. However, in no other case are two operators or two elements in succession permitted.

**Using Symbols in Expressions**

In \texttt{MAP}, ordinary symbols are not assigned values until a pass over the entire program has been completed. Therefore, there are no restrictions in the order of symbol definition. For example,

\[
\begin{align*}
\texttt{A} & \quad \text{ORG} \quad \texttt{A} \\
& \quad \text{EQU} \quad 10000
\end{align*}
\]

is a valid sequence in \texttt{IBMAP}.

Symbols in the variable field of pseudo-operations that affect location counters must be definable at assembly time. In the sequence

\[
\begin{align*}
\texttt{BEGIN} & \quad \texttt{A} \\
& \quad \texttt{BSS} \quad \texttt{B} \\
& \quad \texttt{BES} \quad \texttt{C}
\end{align*}
\]

\texttt{A}, \texttt{B}, and \texttt{C} may not be virtual symbols. (The \texttt{ORG} pseudo-operation is not subject to this restriction.) The values of symbols \texttt{B} and \texttt{C} are always treated as constant.

In general, any valid expression may appear in the variable field of machine instructions. In a relocatable assembly, final evaluation of complex arithmetic expressions containing virtual symbols actually takes place at load time, when all symbols have been defined.

**Relative Addressing**

After an instruction has been named by the presence of a symbol in the name field, references to that instruction can be made in other instructions by using the symbol. Instructions preceding or following the named instruction can also be referenced by indicating their position relative to the named instruction. This procedure is called relative addressing. A relative
address is, effectively, a type of expression. For example, in the sequence

\[
\begin{array}{ll}
\text{ALPHA} & \text{TRA} \\
\text{CLA} & \text{GAMMA} \\
\text{STGAM} & \text{STO} \\
\text{TPL} & \text{LOC1}
\end{array}
\]

control may be transferred to the instruction \text{CLA GAMMA} by either of the following instructions:

\[
\begin{array}{ll}
\text{TRA} & \text{ALPHA} + 1 \\
\text{TRA} & \text{STGAM} - 2
\end{array}
\]

It is also possible to use the asterisk (*) as an element in a relative address. For example, in the sequence

\[
\begin{array}{ll}
\text{AXT} & 10,1 \\
\text{CLA} & A,1 \\
\text{SUB} & B,1 \\
\text{SUB} & C,1 \\
\text{STO} & \text{SUM},1 \\
\text{TIX} & \text{LOOP},1,1
\end{array}
\]

the last instruction indicates a conditional transfer to location \text{LOOP}. This could also be written

\[
\text{TIX} \quad *-4,1,1
\]

The address \(-4\) is interpreted as “the location of this instruction minus 4.”

Relative addressing must be used carefully in combination with pseudo-operations, since some pseudo-operations may generate more than one word or no words in the object program. For example, the instruction

\[
\begin{array}{ll}
\text{ALPHA} & \text{OCT} \\
\text{OCT} & 2732,427,12716
\end{array}
\]

generates three words of octal data, with \text{ALPHA} assigned to the address of the first word generated. Thus the address \text{ALPHA} + 2 refers to the third word generated (12716).

Reference can also be made to a word in a block of storage reserved by a \text{BSS} or \text{BES} pseudo-operation by using relative addressing. For example, the instruction

\[
\begin{array}{ll}
\text{BETA} & \text{BSS} \\
\text{BSS} & 50
\end{array}
\]

reserves a block of 50 words, where \text{BETA} is assigned to the first word of the block. The address \text{BETA} + 1 refers to the second word, and \text{BETA} + n refers to the \((n+1)\)st word.
MAP provides the programmer with more than sixty pseudo-operations that can perform a variety of programming functions with greatly reduced programming effort. They have been grouped according to function, and the structure and purpose of most of the MAP pseudo-operations are described in this part of the publication. The macro-operation facility is covered separately in Part III.

Location-Counter Pseudo-Operations

Location counters enable instructions that are written in one sequence to be loaded in a different sequence. MAP enables a programmer to establish an indefinite number of location counters, which can be represented by symbols of his choice. The symbol used to represent a location counter may duplicate any other symbol in the program except another location-counter symbol.

The blank location counter, so called because it has no associated symbol, is the basic location counter. If the use pseudo-operation is not used, instructions are assembled under the blank location counter. In addition, a location counter represented by two slashes (//) is reserved for use with blank COMMON.

The USE Pseudo-Operation

The use pseudo-operation places succeeding instructions under control of the location counter represented by the symbol in the variable field. The format of the use pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>USE</td>
<td>Either:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. A single symbol, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Blanks, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. //, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. The word PREVIOUS</td>
</tr>
</tbody>
</table>

The location counter in control at the time of the use pseudo-operation is suspended at its current value. It is temporarily preserved as the "previous" counter. It continues from this value if it is reactivated by another USE. If use with the word PREVIOUS in the variable field is coded, the previous location counter is reactivated. For example, the effect of the sequence

USE A
USE B
USE PREVIOUS

is identical to that of

USE A
USE B
USE A

This option provides a means of returning to a location counter even if the counter symbol is not known.

A use pseudo-operation with a blank variable field must precede the first instruction of the deck if the blank counter is set to a value other than zero by the operation

BEGIN expression

The sequence of location counters is: the blank counter first, the other symbolic counters in the order of their first appearance in a USE or BEGIN pseudo-operation, and finally the // counter.

The BEGIN Pseudo-Operation

The BEGIN pseudo-operation specifies a location counter and establishes its initial value. The format of the BEGIN pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>BEGIN</td>
<td>Two subfields, separated by a comma:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. A location counter symbol,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Any expression</td>
</tr>
</tbody>
</table>

The expression in the variable field may contain any symbol or constant. Relocatable symbols are given their assembly value, and this value becomes absolute. Control-section symbols are given a value of zero.

The value of the second subfield of the variable field is used as the initial value for the location counter represented by the symbol in the first subfield. For example, the instruction

BEGIN ALPHA,BETA

would cause the instructions following

USE ALPHA

to be assembled beginning at location BETA.

If no BEGIN is given for the blank location counter, its initial value is defined as 0 (absolute 0 in an absolute assembly and relative 0 in a relocatable assembly). If no BEGIN is given for the nth location counter (taken in location counter order), its initial value is given as the last value by the \((n-1)st\) location counter. If more than one BEGIN is given for a specific location counter, only the first one is used. Each of the others causes an error message to be issued and is printed as a comment, but is otherwise ignored.
A BEGIN may appear anywhere in the program regardless of the location counter in control.

Note that if the blank location counter is set to a value other than zero by the operation

\[ \text{BEGIN } , \text{expression} \]

the use pseudo-operation with a blank variable field must precede the first instruction of the deck.

The order in which location counters are used is illustrated in the example:

\begin{verbatim}
instruction 1
BEGIN A,*
USE A
instruction 2
instruction 3
BEGIN C,*
instruction 4
instruction 5
USE //
instruction 6
instruction 7
USE B
instruction 8
instruction 9
USE C
instruction 10
END
\end{verbatim}

In this sequence, instruction counters are used in the order: blank, A, C, B, and // . At load time, the sequence of instructions will be:

\begin{verbatim}
instruction 1
instruction 2
instruction 3
instruction 10 (instruction 4 will be overlaid)
instruction 8 (instruction 5 will be overlaid)
instruction 9
instruction 6
instruction 7
\end{verbatim}

For example, the location counter is set at the sixth location of the program by

\[ \text{ORG } \text{START}+5 \]

where \text{START} is the first location of the program.

An ORG terminates the effect of a LOC pseudo-operation by redefining the value of the current location counter and ensuring that this value is used as the next binary card origin.

\textbf{The LOC Pseudo-Operation}

The LOC pseudo-operation permits symbolic addressing in segments of an object program that are to be loaded at one area of storage and then moved to, and executed from, another. It redefines the value of the current location counter but does not affect the load address.

The format of the LOC pseudo-operation is:

\begin{tabular}{|c|c|c|}
\hline
\textbf{NAME FIELD} & \textbf{OPERATION FIELD} & \textbf{VARIABLE FIELD} \\
\hline
1. A symbol, or Blanks & LOC & Any expression \\
\hline
\end{tabular}

The current location counter is reset to the value specified in the variable field. If there is a symbol in the name field, it is given this value.

However, binary card origins are not affected by the LOC pseudo-operation. They remain relative to the value of the location counter prior to the LOC, indicating the address at which the program segment is to be loaded.

If another LOC is encountered when one is already in effect, the current location counter is set to the value specified in the new LOC.

The LOC pseudo-operation may be used only if an absolute assembly (ABSMOD) is specified on the \textsc{ibmap} control card. (See the publication IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C28-6389, for a discussion of the \textsc{ibmap} control card.)

The effect of a LOC pseudo-operation must be terminated (see "The ORG Pseudo-Operation.") before a USE pseudo-operation appears. Otherwise, the LOC will be terminated automatically and an error message issued because undesirable location counter values may result.

\textbf{Storage-Allocation Pseudo-Operations}

The storage-allocation pseudo-operations reserve core storage areas within the sequence of the program.

\textbf{The BSS Pseudo-Operation}

The BSS (Block Started by Symbol) pseudo-operation reserves a block of consecutive storage locations and
may describe the data that will occupy the block. The format of the BSS pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or 2. Blanks</td>
<td>BSS</td>
<td>One or two subfields, separated by a comma: 1. Any expression, 2. Modal and dimensional information</td>
</tr>
</tbody>
</table>

The BSS pseudo-operation increases the value of the current location counter by the defined value of the expression in the first subfield of the variable field. The expression may contain any symbol or constant. Relocatable symbols are given their assembly value, and this value becomes absolute. Control-section symbols are given a value of zero. If there is a symbol in the name field, its defined value is that of the location counter just before the increase.

When the second subfield is used, a symbol must appear in the name field. This subfield supplies modal and dimensional information to the debugging dictionary about the data that will occupy the block. The second subfield must be used if the symbol in the name field also appears in the variable field of a KEEP pseudo-operation or if the full debugging dictionary is specified in the variable field of a SBMAP card. (See the publications IBM 7090/7094 ISYS Operating System: IBJOB Processor Debugging Package, Form C28-6393 and IBM 7090/7094 ISYS Operating System: IBJOB Processor, Form C28-6389.) The format for this subfield is:

mode \((d_1, d_2, d_3)\)

where \(d_1, d_2, \text{and} \ d_3\) are the dimensions, if any, of the array denoted by the symbol in the name field, and mode is one of the following:

- O Octal
- F Floating-point number
- X Fixed-point number
- D Double-precision floating-point number
- J Complex number
- L Logical
- S Symbolic instruction
- C Symbolic command
- H Alphanumeric

If an entry for mode is not one of the nine recognized letters above, the mode is assumed to be octal. If an entry for dimension is not a decimal integer, an error message is issued.

When the second subfield is not used, it is assumed that the mode is octal and the dimension is 1.

For example, in the sequence

<table>
<thead>
<tr>
<th>ALPHA</th>
<th>IORT</th>
<th>BETA,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETA</td>
<td>BSS</td>
<td>4</td>
</tr>
<tr>
<td>GAMMA</td>
<td>IORT</td>
<td>DELTA,6</td>
</tr>
</tbody>
</table>

if \(\text{ALPHA}\) has been assigned to location 1001, both \(\text{BETA}\) and \(\text{GAMMA}\) will be assigned to location 1002 and \(\text{GAMMA}\) to location 1006. Thus, four locations are reserved for an array whose first element is at location \(\text{BETA}\); and an entry, specifying that this array is one-dimensional and that its elements are octal numbers, is made into the debugging dictionary.

In addition to reserving four locations, the instruction

\(\text{BETA} \quad \text{BSS} \quad 4,F(2,2)\)

indicates to the debugging dictionary that the data to occupy this area is a 2 by 2 array of floating-point numbers.

The area reserved by the BSS pseudo-operation is not zeroed.

The BES Pseudo-Operation

The BES (Block Ended by Symbol) pseudo-operation also reserves a block of consecutive storage locations and may describe the data that will occupy the block. The format of the BES pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or 2. Blanks</td>
<td>BES</td>
<td>One or two subfields, separated by a comma: 1. Any expression, 2. Modal information</td>
</tr>
</tbody>
</table>

The expression in the first subfield of the variable field may contain any symbol or constant. Relocatable symbols are given their assembly value, and this value becomes absolute. Control-section symbols are given a value of zero.

The BES pseudo-operation functions almost identically to a BSS pseudo-operation except that the symbol in the name field is defined after the location counter increases and thus refers to the first word following the reserved block.

A symbol must appear in the name field of a BES pseudo-operation if the second subfield of the variable field is used. This subfield has the same use as it does in the BSS except that it specifies modal information only. (If dimensional information is given, it is ignored.) Thus an entry in the second subfield of the variable field of a BES instruction consists of one of the letters O, F, X, D, J, S, C, or H, as described in the discussion of the BSS pseudo-operation. If the entry for mode is omitted or is a symbol other than the nine listed above, the mode is assumed to be octal.

For example, in the sequence

<table>
<thead>
<tr>
<th>ALPHA</th>
<th>IORT</th>
<th>BETA,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETA</td>
<td>BES</td>
<td>4</td>
</tr>
<tr>
<td>GAMMA</td>
<td>IORT</td>
<td>DELTA,6</td>
</tr>
</tbody>
</table>

if \(\text{ALPHA}\) has been assigned to location 1001, both \(\text{BETA}\) and \(\text{GAMMA}\) will be assigned to location 1006 and four locations will be reserved for octal data.

In addition to reserving four locations, the instruction
supplies the debugging dictionary with the information that the data to occupy this area is in the fixed-point mode.

The difference between BES and BSS can be seen in the sequence of instructions

\[
\begin{align*}
\text{ALPHA} & \quad \text{BES} & 25 \\
\text{CLA} & \quad \text{BETA}
\end{align*}
\]

which is effectively the same as

\[
\begin{align*}
\text{ALPHA} & \quad \text{BSS} & 25 \\
\text{CLA} & \quad \text{BETA}
\end{align*}
\]

The area reserved by the BES pseudo-operation is not zeroed.

### The EVEN Pseudo-Operation

The EVEN pseudo-operation forces the current location counter to an even value to ensure an even address for the next instruction or data—usually a double-precision floating-point number. It is used only in the 7094 and has no effect in the 7090 or 7094 II. The format of the EVEN pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>EVEN</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

In a 7094 relocatable assembly, the EVEN pseudo-operation causes the instruction

\[
\text{AXT} 0,0
\]

to be inserted at load time if the load address of the AXT instruction is not even. The AXT instruction has no other effect on the program.

In a 7094 absolute assembly, the EVEN pseudo-operation causes the insertion of the instruction

\[
\text{AXT} 0,0
\]

at assembly time so that it is available at load time in the event that the AXT instruction load address is not even. The AXT instruction has no other effect on the program.

### The LDIR Pseudo-Operation

The LDIR (Linkage Director) pseudo-operation places the Linkage Director in the program at the point of the LDIR. The format of the LDIR pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or Blanks</td>
<td>LDIR</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The Linkage Director is a unique location for each assembly, which is one beyond the final value of the location counter if the LDIR pseudo-operation is not used. If the LDIR pseudo-operation is used with a symbol in the name field, the programmer may refer to the Linkage Director.

The Linkage Director serves as a cross-reference for the call and save pseudo-operations. If neither the LDIR nor LONG pseudo-operations is used, the Linkage Director precedes the Literal Pool. If the LDIR pseudo-operation appears more than once, only its first appearance is effective.

For example,

\[
\begin{align*}
\text{ALPHA} & \quad \text{LDIR} \\
\text{ALPHA} & \quad \text{PZE} \\
\text{BCI} & \quad \text{1,decksname}
\end{align*}
\]

to be generated. The second subfield in the BCI operation is the deksname specified on the SUBMAP card (see the publication IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C28-6389).

### The COMMON Pseudo-Operation

The COMMON pseudo-operation has been preserved solely for compatibility with existing programs. When used together with the pseudo-operation

\[
\text{CONTRL} //
\]

it reserves a storage area called blank COMMON for use with such programs. The format of the COMMON pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
</table>
| 1. A symbol, or Blanks | COMMON | One or two subfields separated by a comma: \\
| 2. Blanks | | 1. Any expression \\
| | | 2. Modal and dimensional information |

The first subfield of the variable field may contain any symbol or constant. Relocatable symbols are given their assembly value, and this value becomes absolute. Control-section symbols are given a value of zero.

The COMMON operation causes:

1. Location counter // to be activated
2. A symbol in the name field, if any, to be defined as having the current value of location counter //
3. Location counter // to be increased by the defined value of the variable field expression
4. The location counter in use prior to the COMMON operation to be reactivated

The effect of the sequence

\[
\begin{align*}
\text{A} & \quad \text{COMMON E} \\
\end{align*}
\]

is equivalent to

\[
\begin{align*}
\text{A} & \quad \text{USE //} \\
\text{BSS} & \quad \text{E} \\
\text{USE PREVIOUS}
\end{align*}
\]

The second subfield of the variable field is used in the same way as the second subfield of the BSS pseudo-operation. (See "BSS Pseudo-Operation.")
Literal-Positioning Pseudo-Operations

Two pseudo-operations, LORG and LITORG, enable the programmer to determine the origin of the Literal Pool (see "Literals").

If neither of these pseudo-operations is used, the Literal Pool begins one location beyond the final value of the location counter in use at the end of a program. If more than one location counter is used by a program, the Literal Pool might overlay part of the program. For example, in the sequence

```
A
  USE  Y
  BSS  1
  USE  X

B
  CLA = 1
  USE  Y
  END
```

the Literal Pool would begin at symbolic location B (one beyond the final value of location counter Y). Overlay also occurs if the // location counter is used but is not in effect at the end of the program. Specifying the location of the Literal Pool with a LORG or a LITORG can eliminate this overlapping.

If double-precision literals are used in a 7094 program, an EVEN pseudo-operation should precede a LORG or LITORG to ensure their entry at an even numbered address.

The LORG Pseudo-Operation

The LORG (Literal Pool Origin) pseudo-operation controls the positioning of all literals used in a program that are not under control of a LITORG pseudo-operation. The format of the LORG pseudo-operation is:

```
<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or 2. Blanks</td>
<td>LORG</td>
<td>Ignored</td>
</tr>
</tbody>
</table>
```

As a result of this pseudo-operation, the Literal Pool starts at the location of the LORG in the symbolic deck. This location is assigned the value of the symbol in the name field. For example, the sequence

```
A
  USE  Y
  BSS  1
  USE  X

B
  CLA = 1
  USE  Y
  END
```

would place the beginning of the literal pool one beyond symbolic location B and give it the symbolic name C.

If more than one LORG is given, only the first is effective. Duplicate literals under the control of a LORG are eliminated.

The LITORG Pseudo-Operation

The LITORG (Literal Origin) pseudo-operation enables literals used within a block of coding to be associated with that block. The primary use of this pseudo-operation is to associate the literals used within a control section (see "Control Dictionary Pseudo-Operations") with that control section.

The format of the LITORG pseudo-operation is:

```
<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or 2. Blanks</td>
<td>LITORG</td>
<td>Ignored</td>
</tr>
</tbody>
</table>
```

All literals encountered prior to a LITORG pseudo-operation (either from the beginning of the program or from a previous LITORG) are placed in a pool starting at the location of the LITORG. The beginning of the pool is assigned the symbol in the name field. For example, the sequence

```
A
  USE  X
  CLA = 1
  ADD = 2

B
  USE  Y
  CLA = 3
  ADD = 4
```

would begin the first Literal Pool (containing 1 and 2) at symbolic location A and the second Literal Pool (containing 3 and 4) at symbolic location B.

Any literals encountered following a LITORG are positioned at the next LITORG or, if there is none, under the control of a LORG pseudo-operation. If there is neither a LORG nor another LITORG, subsequent literals follow an *LORG generated immediately preceding the END card.

Under LITORG control, duplicate literals are eliminated within each LITORG section.

Data-Generating Pseudo-Operations

Five pseudo-operations (OCT, DEC, BCI, LIT, and VFD) provide the programmer with a convenient means of introducing data expressed in a variety of forms into a program during assembly. Numbers introduced by these operations are often referred to as constants. A sixth pseudo-operation, PUR, permits a sequence of symbolic cards to be duplicated a specified number of times.

The OCT Pseudo-Operation

The OCT (Octal Data) pseudo-operation introduces binary data expressed in octal form into a program. The format of the OCT pseudo-operation is:
<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>OCT</td>
<td>1. One or more octal integers, separated by commas, or 2. Blanks</td>
</tr>
</tbody>
</table>

Each subfield in the variable field contains a signed or an unsigned octal integer of \( n \) digits, where \( n \leq 12 \). The only limit on the number of subfields is that they must all be contained in the variable field of one card.

A blank variable field results in a word of all zeros. The OCT operation converts each subfield to a binary word. These words are assigned to successively higher storage locations as the variable field is processed from left to right. If a symbol is used in the name field, it is assigned to the first word of data generated.

For example, each of the instructions

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>OCT</td>
<td>77777777777777</td>
</tr>
<tr>
<td>ALPHA</td>
<td>OCT</td>
<td>-37777777777777</td>
</tr>
</tbody>
</table>

would result in a binary word of 36 consecutive ones at location ALPHA.

In the instruction

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>OCT</td>
<td>43,25,64</td>
</tr>
</tbody>
</table>

the binary equivalent of octal number 43 would appear at location ALPHA, the binary equivalent of 25 at location ALPHA+1, and the binary equivalent of 64 at ALPHA+2.

### The DEC Pseudo-Operation

The DEC (Decimal Data) pseudo-operation introduces data expressed as decimal numbers into a program. The format of the DEC pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or 2. Blanks</td>
<td>DEC</td>
<td>1. One or more decimal data items, separated by commas, or 2. Blanks</td>
</tr>
</tbody>
</table>

The only limit on the number of subfields is that they must all be contained in the variable field of one card.

A blank variable field results in a word of all zeros. The DEC operation converts each subfield to one or two binary words, depending on whether the decimal data item is single or double precision. These words are stored in successively higher storage locations as the variable field is processed from left to right. A symbol used in the name field is assigned to the first word of data generated.

For example, the instruction

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>DEC</td>
<td>43,25</td>
</tr>
</tbody>
</table>

would result in the binary equivalent of decimal number 43 appearing at location ALPHA and of decimal number 25 appearing at location ALPHA+1.

### The BCI Pseudo-Operation

The BCI (Binary Coded Information) pseudo-operation introduces binary-coded decimal data into a program. Each data word generated consists of six 6-bit characters in standard BCD code. The format of the BCI pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
</table>

If a digit is used in the count subfield, it must be a single digit from 1 through 9.

A null subfield indicates a count of ten. To accommodate the full ten words of data on the card, the null subfield must be indicated by a comma in column 12.

The data subfield contains any desired alphanumeric information (see Appendix E for the MAP BCD character code).

The length of the data subfield is determined by the number of six-character words specified in the count subfield. The immediate value of the symbol used in this subfield may also be used to determine the length of the data subfield.

The comments field begins immediately after the end of the data subfield, and no blank character is needed to separate the data subfield from the comments field. Any part of the data extending beyond the limit of the data field is treated as comments. Blanks are inserted as required to fill the data subfield to the length specified by the count subfield.

Thus, the BCI pseudo-operation introduces data words into consecutive locations, the number of words generated being equal to the number in the count subfield. A symbol used in the name field is assigned to the first word of data generated.

For example,

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>BCI</td>
<td>2, PROFITRISE IN PERCENT</td>
</tr>
</tbody>
</table>

would generate the data words PROFIT RISE, whereas IN PERCENT would be comments.

### The VFD Pseudo-Operation

Each VFD pseudo-operation generates no, one, or more than one binary data words and assigns them to successively higher storage locations. The format of the VFD (Variable Field Definition) pseudo-operation is:

MAP Pseudo-Operations
<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or Blanks</td>
<td>VFD</td>
<td>Any number of subfields, separated by commas</td>
</tr>
</tbody>
</table>

Each subfield of the variable field generates zero, one, or more than one bits of data. Thus, the unit of information for this pseudo-operation is the single bit.

Each subfield may be any one of three types: octal (Boolean), alphabetic, or symbolic (including decimal integers).

The subfield of the VFD pseudo-operation consists of:

1. The type letter
   a. The letter O signifies an octal (Boolean) field.
   b. The letter I signifies an alphabetic field.
   c. The absence of either O or I signifies a symbolic or decimal field.

2. The bit count — either a decimal integer or an immediate symbol specifies the number of bits to be generated by the subfield. If an immediate symbol is used, care should be taken to avoid confusion caused by the type letter. The maximum allowable bit count for a single subfield is 864.

3. The separation character slash (/)

4. The data item
   a. In an octal subfield, the data item may be a Boolean expression or an octal number.
   b. In an alphabetic subfield, the data item is a string of characters none of which is a comma or a blank.
   c. In a symbolic subfield, the data item is one expression. The item is taken modulo 2^m.

Any number of subfields may be used. 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, the data item in that subfield is converted to an n-bit binary number that is placed in the leftmost n positions of the first data word to be generated. 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−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 subfield has been processed. If the total number of bit positions used is not a multiple of 36, the unused bit positions at the right of the last generated data word are filled with zeros.

If the data item is a single signed octal integer of any length, the sign is recorded as the high-order bit of the specified bit group.

If after conversion a symbolic or octal item occupies more than n bits, only the rightmost n bits of the converted data item are used. If the converted data item occupies fewer than n bits, enough 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 assembly program.

The data item in a symbolic subfield is converted as a symbolic expression. Decimal integers must not exceed 32767.

The data item in an octal subfield may be any valid Boolean expression. In addition, a signed or unsigned octal integer, which may exceed 18 bits, will be recognized and accepted by the assembly program.

The data item in an alphabetic subfield may consist of any combination of characters other than a comma or a blank. Each character is converted to its 6-bit binary code equivalent. If the converted data item occupies more than n bits, only the rightmost n bits are used. If the converted data item occupies fewer than n bits, sufficient 6-bit groups of the form 110000 (the hex 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 6, the leftmost character or blank is truncated. None of these conditions is regarded as an assembly error.

For example, the VFD pseudo-operation could be used to break up a 36-bit word as follows: Positions S and 1 through 9 must contain the binary equivalent of the decimal integer 895, positions 10 through 14 must contain the binary equivalent of the octal integer 37, positions 15 through 20 must contain the binary equivalent of the character C, and positions 21 through 35 must contain the value of the symbol ALPHA. The instruction to generate this word is

VFD 10/895,05/37,H6/C, 15/ALPHA

The LIT Pseudo-Operation

The LIT (Literal) pseudo-operation places data items from the subfields of the variable field into the Literal Pool in successively higher storage locations.

The format of the LIT pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>LIT</td>
<td>Data subfields, separated by commas</td>
</tr>
</tbody>
</table>

Rules for the contents of the data subfields are the same as those governing literals except that the equal sign (=) is omitted.

A Literal Pool entry made using a LIT pseudo-operation is assumed to be double precision if the variable field generates only two consecutive words of data. If a double-precision entry is made in the Literal Pool
by either a \texttt{LIT} pseudo-operation or a double-precision literal, the number is placed in an even location relative to the beginning of the Literal Pool. (In this respect, the assembly program does not distinguish double-precision floating-point numbers from double-precision fixed-point numbers.)

For example,

\begin{verbatim}
LIT 1,2
\end{verbatim}

causes the number 1 to be placed in an even location relative to the beginning of the Literal Pool (which can result in duplicate entries in the Literal Pool).

The instruction

\begin{verbatim}
LIT 1EE1
\end{verbatim}

results in a double-precision entry beginning in an even location in the Literal Pool, but the instruction

\begin{verbatim}
LIT 1EE1,2
\end{verbatim}

results in a three-word entry with the first word not necessarily entered into an even location.

Thus, double-precision floating-point numbers may be used as constants if an \texttt{EVEN} pseudo-operation is used immediately preceding the \texttt{LORD} operation or, if no \texttt{LOC} is present, immediately before the \texttt{END} pseudo-operation.

The \texttt{DUP} Pseudo-Operation

The \texttt{DUP} (Duplicate) pseudo-operation causes an instruction or sequence of instructions to be duplicated. An important application is in generating tables. The format of the \texttt{DUP} pseudo-operation is:

\begin{verbatim}
DUP m,n
\end{verbatim}

duplicates the effect of the next \textit{m} cards \textit{n} times.

In the nested \texttt{DUP} pseudo-operations in the sequence

\begin{verbatim}
DUP 1,2
DUP 1,2
PZE X
PZE Z
\end{verbatim}

the single card to be duplicated by the outer \texttt{DUP} is the inner \texttt{DUP}, and the effect of the inner \texttt{DUP} is actually the two operations

\begin{verbatim}
PZE X
PZE X
\end{verbatim}

The sequence generated when the outer \texttt{DUP} is expanded is

\begin{verbatim}
PZE X
PZE X
PZE X
PZE Z
\end{verbatim}

where the last card in the sequence is the first card beyond both the explicit and implicit ranges of the outer \texttt{DUP}.

In the sequence

\begin{verbatim}
DUP 3,2
DUP 1,2
PZE X
PZE Y
PZE Z
\end{verbatim}
The effect of the inner **DUP** is the sequence

\[ \text{PZE} \quad X \]
\[ \text{PZE} \quad X \]

The range of the outer **DUP** includes the two instructions resulting from the expansion of the inner **DUP** plus the two instructions following, i.e., **PZE X** and **PZE Y**.

Thus, the expansion of the outer **DUP** gives the sequence

\[ \text{PZE} \quad X \]
\[ \text{PZE} \quad X \]
\[ \text{PZE} \quad X \]
\[ \text{PZE} \quad X \]
\[ \text{PZE} \quad Y \]
\[ \text{PZE} \quad Y \]
\[ \text{PZE} \quad Y \]
\[ \text{PZE} \quad Z \]

The range of a **DUP** that occurs within the range of another **DUP** must be fixed before the outer **DUP** is encountered. This can be done by using the **SET** pseudo-operation. (See the section “The SET Pseudo-Operation.”) For example, the sequence

\[ \begin{align*}
K & \quad \text{SET} \quad 1 \\
K & \quad \text{DUP} \quad 2,2 \\
K & \quad \text{SET} \quad K+1 \\
K & \quad \text{DUP} \quad K,0 \\
\end{align*} \]

will result in an error message and assembly will be terminated. However, the iteration count may be variable. For example, the sequence

\[ \begin{align*}
K & \quad \text{SET} \quad 1 \\
K & \quad \text{DUP} \quad 2,2 \\
K & \quad \text{SET} \quad K+1 \\
K & \quad \text{DUP} \quad m,K \\
\end{align*} \]

is valid and will be assembled correctly. For example, the sequence

\[ \begin{align*}
K & \quad \text{SET} \quad 1 \\
K & \quad \text{DUP} \quad 2,2 \\
K & \quad \text{SET} \quad K+1 \\
PZE & \quad X \\
PZE & \quad Y \\
PZE & \quad Z \\
\end{align*} \]

would result in

\[ \begin{align*}
PZE & \quad X \\
PZE & \quad Y \\
PZE & \quad Z \\
PZE & \quad X \\
PZE & \quad Y \\
PZE & \quad Z \\
PZE & \quad X \\
PZE & \quad Y \\
PZE & \quad Z \\
\end{align*} \]

Symbol-Defining Pseudo-Operations

**MAP** provides a group of pseudo-operations specifically designed to define the symbols that appear in their name fields. They are useful in a variety of programming applications, such as equating symbols to combine separately written program segments or changing parameters referred to symbolically throughout a program by redefining the symbol.

The EQU and SYN Pseudo-Operations

The **EQU** and **SYN** pseudo-operations are identical. The format of the **EQU** and **SYN** pseudo-operations is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>EQU or SYN</td>
<td>One or two subfields, separated by a comma:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. Any expression,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Modal and dimensional information</td>
</tr>
</tbody>
</table>

The **EQU** and **SYN** pseudo-operations give the name field symbol the same definition — and the same structure — as the expression in the first subfield of the variable field. Thus, if **A** is defined as **X + Y - Z** and the instruction

\[ B \quad \text{EQU} \quad A \]

is given, **B** is defined as **X + Y - Z**. The instruction

\[ \text{LCS} \quad \text{EQU} \quad * \]

defines **LCS** as having the current value of the location counter.

When the second subfield is used, it supplies to the debugging dictionary modal and dimensional information about the symbol in the name field. This subfield must be used if the symbol in the name field also appears in the variable field of a **KEEP** pseudo-operation. (See the publication IBM 7090/7094 IBSYS Operating System: IBJOB Debugging Package, Form C28-6393.) The format for this entry is

\[ \text{mode} \ (d_1, d_2, d_3) \]

where **d_1**, **d_2**, and **d_3** are the dimensions, if any, of the array denoted by the symbol in the name field, and mode is one of the letters **O**, **F**, **X**, **D**, **J**, **L**, **S**, **C**, or **H**, as explained in the discussion of the **BS** pseudo-operation.

If the mode is omitted or is a symbol other than the nine listed above, it is assumed to be octal. Dimensions must be decimal integers or an error message is issued.

The NULL Pseudo-Operation

The format of the **NULL** pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or</td>
<td>NULL</td>
<td>Ignored</td>
</tr>
<tr>
<td>2. Blanks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The **null** pseudo-operation defines the symbol in the name field, if any, as having the current value of the location counter. The operation

```
LCS    NULL
```

is equivalent to

```
LCS    EQU *
```

except that **null** is preferred.

**The MAX Pseudo-Operation**

The **max** pseudo-operation gives the symbol in the name field an absolute value equal to the expression in the variable field that has the maximum defined value. The format of the **max** pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>MAX</td>
<td>Expressions, separated by commas</td>
</tr>
</tbody>
</table>

The maximum value is computed as if all symbols were absolute. The comparison is made after negative values have been complemented. For example, the sequence

```
BSS    A
A      MAX   100,ALPHA,ALPHA - 100
ALPHA  EQU   150
```

is equivalent to

```
BSS    150
```

**The MIN Pseudo-Operation**

The effect of the **min** pseudo-operation is opposite to that of **max**. The symbol in the name field is given an absolute value equal to the expression in the variable field having the minimum defined value. The format of the **min** pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>MIN</td>
<td>Expressions, separated by commas</td>
</tr>
</tbody>
</table>

The minimum value is computed as if all symbols were absolute. The comparison is made after negative values have been complemented. For example, the sequence

```
A      MIN   100,ALPHA,ALPHA - 100
ALPHA  EQU   150
```

is equivalent to

```
BSS    50
```

**The SET Pseudo-Operation**

The SET pseudo-operation causes the symbol in the name field to be defined immediately. The SET pseudo-operation, which can be used to define symbols in both machine instructions and pseudo-operations, is often used to define the symbols in the variable fields of the DUP, VFD, IFT, and IFF pseudo-operations. The format of the SET pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>SET</td>
<td>Any expression</td>
</tr>
</tbody>
</table>

Qualified symbols may not be used in the variable field.

The symbol in the name field is immediately assigned the value (called the S-value) of the variable field expression during the first pass of the assembly program. Thus, the SET pseudo-operation enables the programmer to use sequences of instructions in which decisions depend on the value assigned to a symbol during the first pass of the assembly program. It also permits symbols to be redefined repetitively. The value assigned to the symbol is always a 15-bit integer.

Use of the SET pseudo-operation is subject to the following conditions:

1. Immediate symbols may not be qualified.
2. An immediate symbol used in the variable field of a pseudo-operation affecting location counters (such as **bss**) assumes the final value assigned to it in the program. For example, the sequence

```
A      SET    100
BSS    A
```

is equivalent to

```
BSS    1000
```

3. An immediate symbol should not normally be given a name identical to an ordinary symbol, since doing so can result in multiple definition.

An example of the use of the SET pseudo-operation to assign a value to a symbol during the first pass of the assembly program is shown in the following sequence:

```
ALPHA  SET    50
VFD    ALPHA/BETA
```

By using the SET pseudo-operation, the value of **ALPHA** can be changed without altering the VFD instruction. A similar use of immediate symbols is in making conditional assembly decisions with the IFF/IFT pseudo-operations.

The SET pseudo-operation permits a symbol to be redefined repeatedly for such programming functions as constructing tables and writing macro-operations. For example, in the sequence

```
ALPHA  SET    1
DUP    2,9
PZE    ALPHA
```

**ALPHA** is first assigned a value of 1 and then redefined nine times with its value incremented by 1 at each iteration.
**Boolean Pseudo-Operations**

The Boolean pseudo-operations define symbols as Boolean quantities.

**The BOOL Pseudo-Operation**

The **BOOL** (Undesignated Boolean) pseudo-operation functions like the `eqv` pseudo-operation except that the variable field expression is Boolean and the name field symbol becomes a Boolean symbol. The format of the **BOOL** pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>BOOL</td>
<td>A Boolean expression</td>
</tr>
</tbody>
</table>

**BOOL** defines the symbol in the name field as an 18-bit constant. Relocatable symbols or virtual symbols used in the variable field result in an error. Octal integers in the variable field may not exceed six characters.

**The RBOOL and LBOOL Pseudo-Operations**

The format of the **RBOOL** (Right Boolean) and **LBOOL** (Left Boolean) pseudo-operations is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>RBOOL or LBOOL</td>
<td>A Boolean expression</td>
</tr>
</tbody>
</table>

Relocatable symbols or virtual symbols used in the variable field result in an error. Octal integers in the variable field may not exceed six characters.

These pseudo-operations are similar to **BOOL** except that the symbol in the name field is defined as right (left) Boolean. They are normally used to determine the correct machine operation for the special type D instructions (SIB, BNT, BFT, RIB, and SIB). The following mechanism is used.

1. If the expression in the variable field of an **SIB** instruction is entirely left (right) Boolean, the instruction is assembled as **SIL** (SIB). Constants are considered to be both left and right Boolean. If the expression is a mixture of both left and right Boolean, **SIL** is assembled but a warning message is issued.

2. If the variable field of an **SIL** (SIB) instruction is not purely left (right) Boolean, left (right) and undesignated Boolean, or purely undesignated Boolean, a warning message is issued.

For example, following the instructions

| X | LBOOL 123 |
| Y | RBOOL 456 |
| Z | RBOOL 321 |

**Conditional-Assembly Pseudo-Operations**

Two pseudo-operations provided by **MAP** enable the programmer to specify that the next instruction is to be assembled only if certain criteria are met. Another pseudo-operation, **COTO**, may be used with the conditional-assembly pseudo-operations to extend their effect over more than one instruction.

**The IFT and IFF Pseudo-Operations**

The **IFT** (If True) and **IFF** (If False) pseudo-operations specify conditions that determine whether the next sequential instruction will be assembled. The format of the **IFT** and **IFF** pseudo-operations is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>IFT or IFF</td>
<td>1. element–relational operator–element 2. If present, either of the words OR or AND, preceded by a comma</td>
</tr>
</tbody>
</table>

The **IFT** (IFF) pseudo-operation assembles the next instruction if the condition expressed by the operation and the first subfield is met.

A relational operator consists of one or two adjacent symbols signifying:

- `=` Equals
- `>=` Greater than
- `<=` Less than

The elements at the left and right of these relations must not be qualified. The element is used in one of two ways:

1. To represent a numerical value equal to its S-value
2. To represent literal BCD information, in which case it is surrounded by slash (/) marks

Interpretation of the relational operator depends on the context. If the elements represent BCD, the relation is a scientific collating sequence comparison. For example,

- **IFT** /A/=+/B/
  - is false and would therefore not permit assembly of the next instruction. However,
  - **IFT** /A/=−/B/
  - is true and would permit assembly of the next instruction.
Also,

\[
\text{IFF } // = / \text{A/}
\]

compares blank to A and would permit assembly of the next instruction.

If the elements represent a numeric quantity, the relation is a numeric comparison. The programmer must avoid noncomparable elements.

Presence of the second subfield signifies that another IFT or IFF is to follow, in which case the combined effect of the two is either a logical on or a logical AND.

The S-value and not the definition is used in numeric evaluation of symbols. The set pseudo-operation may be used to control IFF or IFT pseudo-operations, as in the sequence

\[
\begin{align*}
\text{K} & \quad \text{SET} \quad 4 \\
\text{IFT} & \quad \text{k=4}
\end{align*}
\]

This statement is true, and the next instruction will be assembled.

The fact that the conditional assembly extends over only one instruction is not a serious restriction, since the following instruction may be another IFT or IFF pseudo-operation, a \texttt{COTO} pseudo-operation, or a macro-operation that expands to any length (see the section "Conditional Assembly in Macro-Operations").

The variable field of an IFF or IFT pseudo-operation may not be extended by using the ERC pseudo-operation.

The \texttt{GOTO} Pseudo-Operation

The \texttt{COTO} pseudo-operation, when used with an \texttt{IFT} or \texttt{IFF} pseudo-operation, provides the MAP programmer with a method for either assembling or skipping blocks of instructions. The format of the \texttt{COTO} pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>GOTO</td>
<td>One or two subfields, separated by a comma: 1. Symbol, which may be followed by a comma when used alone, 2. The word BLANK, which may be followed by a comma</td>
</tr>
</tbody>
</table>

The \texttt{COTO} pseudo-operation causes all instructions following it, up to but not including the instruction whose name field contains the symbol specified in the variable field of this pseudo-operation, to be skipped during assembly.

If the \texttt{blank} option is present, the symbol in the variable field will not be entered into the dictionary but will appear on the listing. Therefore, the symbol, which may be completely numeric, may be reused.

If the variable field does not end with a comma, the skipped instructions are listed as comments. If the terminating comma is present, listing of skipped instructions is suppressed.

For example, the sequence

\[
\begin{align*}
\text{CLA} & \quad X \\
\text{IFT} & \quad \text{A=1} \\
\text{GOTO} & \quad \text{LCS} \\
\text{STO} & \quad Z \\
\vdots & \quad \vdots \\
\text{CLA} & \quad \text{B} \\
\text{LCS} & \quad \text{TSX} \quad \text{SUB,4}
\end{align*}
\]

would produce

\[
\begin{align*}
\text{CLA} & \quad X \\
\text{LCS} & \quad \text{TSX} \quad \text{SUB,4}
\end{align*}
\]

if \texttt{A} had previously been defined as 1, and

\[
\begin{align*}
\text{CLA} & \quad X \\
\text{STO} & \quad Z \\
\vdots & \quad \vdots \\
\text{CLA} & \quad \text{B} \\
\text{LCS} & \quad \text{TSX} \quad \text{SUB,4}
\end{align*}
\]

if \texttt{A} had been defined as some value other than 1.

In this example, the instructions omitted would be listed and LCS would be entered into the dictionary. Alternate forms would be

\[
\begin{align*}
\text{GOTO} & \quad \text{LCS,BLANK} \quad \text{(LCS not entered, cards listed)} \\
\text{GOTO} & \quad \text{LCS} \quad \text{(LCS entered, no list)} \\
\text{GOTO} & \quad \text{LCS,BLANK} \quad \text{(LCS not entered, no list)}
\end{align*}
\]

If a \texttt{COTO} occurs within a macro-definition and the referenced symbol is outside the definition, only the instructions between the \texttt{COTO} and the \texttt{ENDDM} pseudo-operation are skipped. If a \texttt{COTO} references past an \texttt{IRP} pseudo-operation, only the instructions between the \texttt{COTO} and the \texttt{IRP} are skipped.

Symbol-Qualifying Pseudo-Operations

MAP provides two pseudo-operations, \texttt{QUAL} and \texttt{ENDQ}, that enable a programmer to qualify symbols within sections of a program.

The \texttt{QUAL} Pseudo-Operation

All symbols between the \texttt{QUAL} pseudo-operation and its associated \texttt{ENDQ} pseudo-operation are qualified. The format of the \texttt{QUAL} pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>QUAL</td>
<td>Symbol</td>
</tr>
</tbody>
</table>

The symbol in the variable field qualifies all symbols defined within the section controlled by the \texttt{QUAL} pseudo-operation. References to a symbol defined in a qualified section from within the same section need not be qualified. References from outside the section are qualified by placing the section symbol (variable field symbol
of the qual pseudo-operation) in front of a connecting dollar sign followed by the desired symbol. For example, the symbol Q@ALPHA refers to symbol ALPHA defined in qualified section Q@. The notation BETA refers to symbol BETA, which is not qualified. The unqualified section effectively has a blank qualifier.

Qualified sections may be nested to provide multiple qualification. The range (from a qual to its corresponding endq) of a lower-level qual must fall completely within the range of the next higher qual. A symbol is automatically qualified by any qualifiers of a higher level than the highest one specified in using the symbol. A multiply qualified symbol can be referenced without using all the qualifiers if enough qualifiers are given to determine the symbol uniquely. In any case, the qualifiers must be specified in the same order that nesting occurs within that section.

A sequence illustrating qualification is

\[
\begin{align*}
\text{A} & \\
\text{QUAL} & \\
\text{BSS} & 1 \\
\text{CLA} & X \\
\text{ENDQ} & H \\
\text{QUAL} & J \\
\text{BSS} & 1 \\
\text{ENDQ} & J
\end{align*}
\]

In this case, if X is written as A OR BSS, or CLA, it refers to the first definition of A; X written as JSS refers to the second definition of A.

In the sequence of nested qualification

\[
\begin{align*}
\text{A} & \\
\text{QUAL} & M \\
\text{BSS} & 1 \\
\text{QUAL} & N \\
\text{CLA} & X \\
\text{ENDQ} & N \\
\text{CLA} & Y \\
\text{ENDQ} & M
\end{align*}
\]

X written as A refers to the first definition of A; X written as A refers to the second nonqualified A. Y written as A refers to the second A; Y written as A refers to the first A.

In the more complicated sequence

\[
\begin{align*}
\text{A} & \\
\text{QUAL} & \text{ONE} \\
\text{BSS} & 1 \\
\text{QUAL} & \text{TWO} \\
\text{CLA} & X \\
\text{ENDQ} & \text{TWO} \\
\text{CLA} & Y \\
\text{ENDQ} & \text{ONE} \\
\text{A} & \\
\text{QUAL} & \text{THREE} \\
\text{BSS} & 1 \\
\text{QUAL} & \text{TWO} \\
\text{ENDQ} & \text{TWO} \\
\text{ENDQ} & \text{THREE}
\end{align*}
\]

X refers to the first A by A; to the second A by A, by TOWS, or by ONE$TWO; to the third A by TOWS; and to the fourth A by THREESTWO.

Y refers to the first A by A or by ONEA; to the second A by TOWS or by ONETWO; to the third A by THREESA; and to the fourth A by THREESTWO. In this sequence, the two sections two are distinct and not separate parts of the same section. The first is section ONE$TWO, and the second is section THREESTWO.

### The ENQ Pseudo-Operation

The format of the ENQ pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>ENQ</td>
<td>Either:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. A symbol, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Blanks</td>
</tr>
</tbody>
</table>

ENQ delimits the range of the qualified section whose symbol is in the variable field of this instruction. If the variable field is blank, the innermost qualified section is terminated. However, a low-severity warning message is issued, since a variable field inadvertently left blank can result in errors when using nested qualification.

In nested qualified sections, a separate ENQ is required to terminate each qualified section. Also, qualified sections must be terminated in order beginning with the lowest level section as shown in the following sequence, or an error message will be issued.

\[
\begin{align*}
\text{NAME} & \\
\text{QUAL} & \text{ALPHA} \\
\text{QUAL} & \text{BETA} \\
\text{QUAL} & \text{GAMMA} \\
\text{ENDQ} & \text{GAMMA} \\
\text{ENDQ} & \text{BETA} \\
\text{ENDQ} & \text{ALPHA}
\end{align*}
\]

### Control-Section Pseudo-Operations

Relocatable programs can be divided into segments. By dividing large programs into relocatable segments, individual segments can be coded and checked in parallel, with consequent savings in time. Also, a segment can be modified without requiring reassembly of the entire program.

The control-section pseudo-operations provide the means for making references to and from such segments. IBLDR makes the cross-references among program segments that are assembled separately but
loaded together. (For further information, see the publication IBM 7090/7094 IBSYS Operating System: IBJOB Processor, Form C25-6389.)

Each program segment is a control section. In addition, sections within segments may be designated as control sections by the programmer.

IBLEN treats control sections as being variable. A control section may be replaced by another control section or even deleted entirely. If more than one control section is given the same designation, generally only the first control section is retained.

An EVEN appearing within a control section must have the same relative location as the beginning of the control section. This will result in the first location of the control section being assigned an even location at load time. Thus, an instruction or data within a control section will be assigned to an even location if it appears an even distance from the control section origin.

The CONTRL Pseudo-Operation

The CONTRL pseudo-operation designates a program or a part of a program as a control section. The format of the CONTRL pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>CONTRL</td>
<td>One of the following:  1. A location counter symbol, or  2. A qualification symbol, or  3. Two subfields, separated by a comma, each containing an ordinary symbol</td>
</tr>
</tbody>
</table>

The CONTRL pseudo-operation delimits the control section named in the name field in accordance with the contents of the variable field. If a location-counter symbol is used in the variable field, all instructions under control of the specified location counter are delimited. The blank location-counter cannot be used as a control section. If a qualification symbol is used, all instructions between the specified QUAL pseudo-operation and its associated ENDP pseudo-operation are delimited. If two subfields are used, all instructions are delimited beginning at the location specified by the first symbol and ending at, but not including, the location specified by the second symbol. CONTRL pseudo-operations that refer to absolute symbols, or to location symbols whose definitions depend on an absolute origin within a relocatable assembly, are discarded and an error message is issued. An error message will also be given if a qualification symbol or location counter symbol defining a control section falls under an absolute origin. If the absolute origin occurs within the control section, no diagnostic will be given and relocation errors may occur during loading.

If there is no symbol in the name field, the first symbol in the variable field is taken as the external name of the control section and a low-severity message is issued. The length of a control section is always the difference between the value of the location counter in control at the end of the section and its value at the beginning of the section. Hence, the use of ORG or USE pseudo-operations within control sections may result in incorrect length calculations, in effect, "losing" instructions from the section. Text which has been placed within a control section by use of an ORG pseudo-operation cannot be deleted. Therefore, if several of such control sections with the same external name occur, the text from the last one encountered during loading will be used. If all of such control sections with the same external name are deleted, execution will be suppressed because of the resulting virtual reference which cannot be defined.

The CONTRL pseudo-operation may appear anywhere in the program. In an absolute assembly, a low-severity error message is printed, but CONTRL is otherwise ignored. For example,

```
X CONTRL A,B
```

defines the portion of the program from A to, but not including, B as control section X.

Control sections may be nested.

To obtain the blank COMMON area, as used by 7090/7094 FORTRAN IV, an instruction of the form

```
CONTRL //
```

must be used. For example, in the sequence

```
CONTRL //
USE A

USE B

USE //
BSS 20
USE C

USE A

END
```

the blank COMMON counter // will have its initial location defined as the last value reached by location counter C. The area under control of the // counter is a control section.

The ENTRY Pseudo-Operation

The ENTRY pseudo-operation provides a reference from outside a program segment to a point within the program segment. The format of the ENTRY pseudo-operation is:
The name field symbol becomes the external name of the entry point. The variable field symbol is the internal name of the entry point and must be an ordinary symbol, although it may be qualified. If the name field is blank, the variable field symbol serves as the external name. If the variable field symbol is qualified, the (leftmost) qualifier is used. For example,

\[
\text{ALPHA ENTRY BETA}
\]

specifies that \text{ALPHA} is the external name of an entry point into this program from another program and that \text{BETA} is the internal name of this entry point.

If an \text{ENTRY} pseudo-operation refers to either an absolute symbol, a symbol that was defined by an \text{EQU} or \text{SYN} pseudo-operation, or a location symbol whose definition depends on an absolute origin within a relocatable assembly, it is discarded and an error message is issued.

**File-Description Pseudo-Operations**

Two pseudo-operations are provided by \text{MAP} for specifying input/output file requirements in relocatable assemblies. These pseudo-operations describe files that are used in conjunction with Library \text{IOCS}. (See the publication \text{IBM 7090/7094 IBSYS Operating System: Input/Output Control System, Form C28-6345}.)

\text{FIELD} generates a file control block and assigns the file to a buffer pool. File control blocks described in one program segment may be referenced in other segments. If the same file is described more than once, only the first description is effective.

**The FILE Pseudo-Operation**

The \text{FILE} pseudo-operation enables the programmer to specify input/output file requirements. The \text{FILE} pseudo-operation causes generation of a \text{FILE} card, as well as any \text{ETC} cards needed. The format of the \text{FILE} pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>FILE</td>
<td>External file name, options, . . .</td>
</tr>
</tbody>
</table>

The symbol in the name field of the \text{FILE} pseudo-operation is the internal name of the file used by the programmer within his program. Whenever this name appears in the variable field of an instruction, the relocatable reference is to the generated file control block for this file.

The first subfield of the variable field is the external file name. The order of the subsequent subfields is arbitrary.

In describing the subfields, options that may be included or omitted are shown in brackets. When an option is not specified, the standard option, which is shown underlined, is assumed. Braces indicate that a choice of the enclosed options is to be made by the user. Options are shown in all upper-case letters in the form in which they must be specified.

**External File Name** is an alphabetic literal of up to 18 characters used to determine equivalence between files. This subfield must be specified as the first subfield in the variable field. It may be null (the variable field may start with a comma), in which case the six-character name field (left-justified with trailing blanks) is inserted as the external name.

**Unit-Assignment Option**

\[[, \text{primary unit}] [, \text{secondary unit}]\]

Two symbolic units may be specified for each file: the primary unit, and a secondary unit to be used as a reel-switching alternate. The format used for these specifications is indicated below, where the following notation is used:

- \(X\) a real channel (specified by one of the letters \(A\) through \(H\))
- \(P\) a symbolic channel (specified by one of the letters \(S\) through \(Z\))
- \(I\) an intersystem channel (specified by one of the letters \(J\) through \(Q\))
- \(K\) a unit number (specified by one of the numbers \(0\) through \(9\))
- \(A\) an access mechanism number (specified by the number \(0\))
- \(M\) a module number (specified by one of the numbers \(0\) through \(9\))
- \(S\) a data channel switch or interface (specified by either of the numbers \(0\) or \(1\))
- \(M\) a model number of 729 Magnetic Tape Unit (specified by \(II, IV, V,\) or \(VI\))
- \(D\) 1301 Disk Storage (specified by the letter \(D\))
- \(N\) 7320 Drum Storage (specified by the letter \(N\))
- \(H\) 7340 Hypertape Drive (specified by the letter \(H\))

The following format is used for assigning units:

**SPECIFICATION**

- **blank**
- **M**
- **X**
- **P**
- **X(k)**
- **PM**
- **P(k)M**
- **I**

Use any available unit.

Use any available 729 Magnetic Tape Unit of this model.

Use any available unit on this channel.

Use any available unit on this channel.

Use \(k\)th available unit on this channel. Parentheses are required.

Use any available 729 Magnetic Tape Unit of specified model on designated symbolic channel. Parentheses are required.

Use \(k\)th available 729 Magnetic Tape Unit of specified model on designated symbolic channel. Parentheses are required.

Use any available unit on channel \(I\). This specification can be used for input and output units.
IM
Use any available 729 Magnetic Tape Unit of specified model on channel I. This specification can be used only for output units.

I(k)
Use kth available unit on channel I. Parentheses are required. This specification can be used for input and output units.

I(k)M
Use kth available 729 Magnetic Tape Unit of specified model on channel I. Parentheses are required. This specification can be used only for output units.

I(k)R
Use kth available unit on channel I, and release unit from reserve status after use. Parentheses are required.

XDAm/s
Use 1301 Disk Storage on channel X, access mechanism number a (which must be 0), and module number m.

XNam/s
Use 7320 Drum Storage on channel X, access mechanism number a (0), module number m (0, 2, 4, 6, 8) and data channel switch setting s (0, 1).

XHk/s
Use 7340 Hypertape Drive on channel X, unit number k, and data channel switch s.

IN, IN1, IN2
Use the system input unit (SYSIN1) and the alternate system input unit (SYSIN2) as the primary and secondary units respectively.

OU, OU1, OU2
Use the system output unit (SYSOUT1) and the alternate system output unit (SYSOUT2) as the primary and secondary units respectively.

PP, PP1, PP2
Use the system peripheral punch (SYSPP1) and the alternate system peripheral punch (SYSPP2) as the primary and secondary units respectively.

UTk
Use system utility unit number k.

CKk
Use system checkpoint unit number k.

RDX
Use card reader on channel X.

PRX
Use printer on channel X.

PUX
Use card punch on channel X.

INT
File is internal.

* An asterisk in the secondary unit field indicates that the secondary unit of a file is to be any unit on the same channel and of the same model as the primary unit.

NONE
No units are assigned. A file control block is generated but does not refer to a unit control block.

The effects of these operations are:

MOUNT
A message is printed before execution, and a stop occurs for the required operator action. MOUNT is the standard option for nonsystem units.

READY
A message is printed before execution, but no stop occurs. READY is the standard option for all input/output units assigned to system unit functions.

DEFER
A message and operator stop are deferred until the file is opened by the IOCS calling sequence.

TSX .OPEN, 4
PZE internal file name

The i form of this option overrides for unit i any general option specified.

As an example of the file-mounting option,

MOUNTI, DEFER2 causes the MOUNT action for the primary unit and the DEFER action for the secondary unit.

Operator File-List Option

[ (LIST (NOLIST) ) ]

LIST This file will appear in the operator's mounting instructions.

NOLIST No message will be printed unless the DEFER option has been specified.

File-Usage Option

[ (INPUT (OUTPUT (INOUT (CHECKPOINT (or CKPT)))) ) ]

INPUT This is an input file.

OUTPUT This is an output file.

INOUT This file may be either an input or an output file. The object program sets the appropriate bits in the file block. The file is initially set at input.

CHECKPOINT (or CKPT) This is a checkpoint file.

Block-Size Option

[ , BLOCK=xxxx (or, BLK=xxxx) ]

xxxx is an integer (0-9999) that specifies block size for this file. If the block-sequence and/or check-sum options (see below) are specified, a word must be added in determining block size. If the BLOCK option is omitted, the assembly program assumes a block size of 14 for BCD or MBCD files and 256 for BIN and MBIN files.

Activity Option

[ , ACT=xx ]

xx is an integer (0-99) that specifies activity of this file in relation to other files. If the activity subfield is omitted, activity is assumed to be 1. The activity value is used in determining the number of input/output buffers assigned to each buffer pool in the object program.
Reel-Handling Option

- **ONEREEL**  
  for unlabeled files only
- **MULTIREEL**  
  (or REELS)
- **NOSEARCH**  
  (or SEARCH)

| No reel switching should occur. 
| Reel switching will occur. Every output file will switch reels if an end-of-tape condition occurs. |
| A labeled file is not specified. |

File-Density Option

- **HIGH**  
  Tape-density switch is assumed to be set so that execution of an SDH will result in using correct density.
- **LOW**  
  Tape-density switch is assumed to be set so that execution of an SDH will result in using correct density.
- **200**  
  Tape density switch is assumed to be set so that execution of an SDH will result in a file-recording density of 200 cpi.
- **556**  
  Tape density switch is assumed to be set so that execution of an SDH will result in a file-recording density of 556 cpi.
- **800**  
  Tape density switch is assumed to be set so that execution of an SDH will result in a file-recording density of 800 cpi.

If a system unit is assigned to this file, system set density supersedes the density specified by these options.

Mode Option

- **BCD**  
  File is in BCD mode.
- **BIN**  
  File is in binary mode.
- **Mxbcd**  
  File is in mixed mode, and first record is BCD.
- **mxbin**  
  File is in mixed mode, and first record is binary.

Label-Density Option

- **SLABEL**  
  All header label operations performed at installation standard density, which is currently high density.
- **HILABEL**  
  All header label operations performed at high density.
- **LOLABEL**  
  All header label operations performed at low density.
- **FLABEL**  
  All header label operations performed at same density as file.

Regardless of these options, the **LABEL** pseudo-operation must be used to specify a labeled file. If label density is not specified, all label options are performed at the density that is high density at the particular installation.

Block-Sequence Option

- **NOSEQ**  
  Block-sequence word neither checked if reading, nor formed and written if writing.
- **SEQUENCE**  
  Block-sequence word checked if reading, or formed and written if writing.

Check-Sum Option

- **NOCKSUM**  
  Check sum neither checked if reading, nor formed and written if writing.
- **CKSUM**  
  Check sum checked if reading, or formed and written if writing.

Check-sum options may not be specified unless a block-sequence option has been specified.

Checkpoint Option

- **NOCKPTS**  
  No checkpoints initiated by this file.
- **CKPTS**  
  Checkpoints initiated by this file.

Checkpoint-Location Option

- **[AFTER LABEL]**  
  Checkpoints are written following the label on this file when reel switching occurs. If **ckpts** is specified and this field is omitted, checkpoints are written on the checkpoint file when reel switching occurs.

File-Close Option

- **SCRATCH**  
  File is rewound at end of application.
- **PRINT**  
  File is to be printed and is rewound and unloaded at end of application. **PRINT** will appear in on-line removal message at end of execution.
- **PUNCH**  
  File is to be punched and is rewound and unloaded at end of application. **PUNCH** will appear in on-line removal instructions.
- **HOLD**  
  File is to be saved and is rewound and unloaded at end of application. **HOLD** will appear in on-line removal instructions.

If the unit assigned is system input unit 1, system output unit 1, or system peripheral punch unit 1, the unit will not be rewound and the removal message will not be printed.

Starting Cylinder-Number Option

- **[CYLINDER=***]**  
  *** is the number (000-249 for disk, 000-009 for drum) of the starting cylinder for this file. The equals sign is required. When disk or drum storage is specified for a file, the starting cylinder number must be specified by the user.
Cylinder-Count Option

[CYLCOUNT=xx (or, CYLCT=xxx)]

xxx is the number (000-250 for disk, 000-010 for drum) of consecutive cylinders to be used by this file. The equals sign is required. When disk or drum storage is specified for a file, cylinder count must be specified by the user.

Disk Write-Checking Option

[WRITECK]

Write-checking is performed after each disk-write or drum-write sequence for this file.

Hypertape Reel-Switching Options

HYPERTAPE

These options may be used in conjunction with the Hypertape option, HYPER, where reel switching is likely to occur. If any of these options are used but HYPER is not specified, a warning message is issued.

The effects of these options are:

HRFP  Hypertape, rewind, file protect.
HRNFP Hypertape, rewind, no file protect.
HNRFP  Hypertape, no rewind, file protect.
HNRFP  Hypertape, no rewind, no file protect.

Five subfields provide information for cross-checking by INDR. These subfields, the conversion, block-size check, nonstandard label routine, Hypertape, and pool-attachment options, are not placed on the $FILE card.

Conversion Option

CONV

NOHCVN Alphanumeric-to-BCD conversion routine not necessary. File may not be assigned to card equipment.
REQHCV Alphanumeric-to-BCD conversion routine required. File must be assigned to card equipment.
OPTHCV Alphanumeric-to-BCD conversion optional.

Regardless of the conversion options specified, it is the responsibility of the programmer to provide the required conversion routines. File may be assigned to any input/output device.

Block-Size Check Option

[MIN=xxx]

MULTI=xxx Block size is a multiple of xxx.
MIN=xxx Minimum block size is xxx.

Only one of the block-size check options may appear. The quantity specified is used by IXDR to check the block size indicated by the BLOCK option. If neither option appears, block size is assumed to be exactly that specified by the BLOCK option.

Nonstandard-Label-Routine Option

[NSLBL=symbol]

The symbol is the name of a nonstandard-label routine. If the label routine is part of the program segment being assembled, the label routine must be made a control section with the symbol used as its external name. If the label routine is not part of this program segment, the symbol must be a virtual symbol.

Hypertape Option

[HYPER]

HYPER must be specified if a program requires Hypertape for a particular file. If reel switching may occur, the Hypertape reel-switching options may be used in conjunction with HYPER. However, use of Hypertape reel-switching options without specifying HYPER results in a warning message.

If a file may be attached to a Hypertape or a 729 Magnetic Tape Unit, the HYPER specification is not necessary.

Pool-Attachment Option

[POOL]

POOL This file is to be attached to a pool.
NOPOOL A file control block is to be generated, but this file is not to be attached to a pool.

If a CHECKPOINT (or CKPT) is specified for the file, usage option, NOPOOL is automatically assumed.

If the NOPOOL option is specified, it is assumed that the IOCS initialization sequences of .DEFIN and .ATTACH for this file will be executed by the object program prior to opening the file. (See the publication IBM 7090/7094 IBSYS Operating System: Input/Output Control System, Form C28-6345.)

Example of FILE pseudo-operation

For example, the options for an input file might be specified in the instruction

INPUT FILE .A(1), READY.BLK=20, 556,HOLD

Since the first subfield is null, the symbol INPUT in the name field is regarded as the external name. The remaining subfields specify the first available unit on channel 1, the file-mounting option, a block size of 20 words, a file density of 556 characters per inch, and the file is to be saved and must be rewound and unloaded at the end of the application.

The LABEL Pseudo-Operation

The LABEL pseudo-operation enables the programmer to label a file and causes generation of the $LABEL control card. Whereas the FILE pseudo-operation describes the file characteristics, LABEL simply labels the file. The format of the LABEL pseudo-operation is:

MAP Pseudo-Operations 31
The file name is an alphameric name of eighteen or fewer BCD characters. If this subfield is null (the variable field begins with a comma), the symbol in the name field is inserted as the file name. If the name field is also blank, 000000 is inserted as the file name.

The file serial number is an alphameric subfield of five or fewer characters, and it may be null. If the label is for a file on disk or drum, this subfield must contain two BCD characters to specify the Home Address-2. (For further information, see the publication IBM 1301, Models 1 and 2, Disk Storage and IBM 1302, Models 1 and 2, Disk Storage with IBM 7090, 7094, and 7094 II Data Processing Systems, Form A22-6785.)

The reel-sequence number is a numeric subfield of four or fewer digits and it may be null. For retention period in days, four or fewer numeric characters are used. For date, two or fewer numeric characters represent the year, and three or fewer numeric characters represent the day of the year. The year and the day of the year are separated by the character / (slash).

The file identification name is an alphameric subfield of eighteen or fewer BCD characters. This subfield may contain blanks but not commas. A comma will terminate this subfield, and excessive subfields will be flagged as errors. This subfield may also be null.

For example,
LABEL INVOICE,.241,63/248,PRIMARY FILE specifies that the INVOICE file be labeled, provides its reel-sequence number of 241, dates it as the 248th day of 1963, and specifies PRIMARY FILE as the file identification name.

The variable field of the LABEL pseudo-operation must be contained on one card. No etc cards may be used following LABEL.

The variable field is checked for errors. If there are more than five subfields, the variable field is truncated, only the first five subfields are used, and a warning message is printed. If there are fewer than five subfields, an appropriate number of commas is supplied so that the LABEL card always has the required subfields and a warning message is printed.

Each subfield is then checked for length except for the last one. Subfields that are longer than the specified maximums are truncated to the maximum number of characters allowed for each, and a format error message is printed. Numeric subfields are also checked for validity, and the presence of any nonnumeric character causes a format error message to be printed.

### Operation-Defining Pseudo-Operations

Three pseudo-operations that define symbols as operation codes are provided by MAP.

#### The OPD Pseudo-Operation

The OPD (Operation Definition) pseudo-operation defines the symbol appearing in the name field as an operation code. The format of the OPD pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>OPD</td>
<td>12-digit octal machine operation code definition</td>
</tr>
</tbody>
</table>

The 12-digit machine operation code definition in the variable field must be specified according to the general format given in Appendix C.

The OPD pseudo-operation defines the symbol in the name field as an operation code. The symbol must be defined by the OPD pseudo-operation before its use in an operation field.

For example,

ALPHA OPD 430106000500 defines ALPHA as an operation code having the same effect as the machine instruction CLA.

#### The OPVFD Pseudo-Operation

The OPVFD (Operation Variable Field Definition) pseudo-operation defines the symbol in the name field as the operation code represented by the expression in the variable field. The format of the OPVFD pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>OPVFD</td>
<td>From 1 to 36 subfields, separated by commas</td>
</tr>
</tbody>
</table>

The format of the variable field is the same as that given for the VFD pseudo-operation. The variable field expression must result in a 36-bit word having the format given in Appendix C.

The symbol in the name field becomes the mnemonic operation code of the instruction. The symbol must be defined by OPVFD before being used in an operation field.

For example,

ALPHA OPVFD 06/43,05/00,07/106,06/00, 012/0500 defines ALPHA as an operation code having the same effect as the machine instruction CLA.
The OPSYN Pseudo-Operations

The OPSYN (Operation Synonym) pseudo-operation equates the symbol in the name field to the mnemonic operation code in the variable field. The format of the OPSYN pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>OPSYN</td>
<td>Mnemonic operation code</td>
</tr>
</tbody>
</table>

The mnemonic operation code in the variable field must be a valid operation code (i.e., a machine operation code, a pseudo-operation code, a macro-operation code, or a code that has been defined previously by OPD, OPVFD, or another OPSYN).

If a previously defined operation code is redefined with OPD, OPVFD, or OPSYN, a warning message is issued. For example,

CLA OPSYN CAL

redefines CLA as CAL. The message warns the programmer of possible inadvertent redefinition of an existing operation code.

Miscellaneous Pseudo-Operations

The END Pseudo-Operation

The END pseudo-operation signals the end of the symbolic deck and terminates assembly. The END operation must be present and must be the last card in the symbolic deck. The format of the END pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or Blanks</td>
<td>END</td>
<td>1. An element, or Blanks</td>
</tr>
</tbody>
</table>

In a relocatable assembly, the value of the element in the variable field is the nominal starting point of the program segment.

The END pseudo-operation performs the following functions in an absolute assembly:

1. Any binary output waiting in the punch buffer is written out.
2. A binary transfer card to which control is transferred is produced. It has a transfer address that is the value of the expression in the variable field.

If UNPNCH is in effect, no cards are punched.

The ETC Pseudo-Operation

The variable field of most instructions may be extended over additional cards by using the ETC pseudo-operation. The format of the ETC pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignored</td>
<td>ETC</td>
<td>Subfields, separated by commas, or partial subfields</td>
</tr>
</tbody>
</table>

The ETC pseudo-operation appends its variable field as a continuation of the variable field of the previous instruction. The blank that separates the variable field from the comments field of a card is an end-of-card indicator and not an end-of-variable-field indicator. The number of ETC cards in one group is generally limited by the size of the resultant expression and/or the number of subfields. A variable-field expression is limited to about 100 elements, operators, and/or subfields. No element of an expression may be split between two cards. For example, the instruction

```
TIX NAME +1,4,1
```

could be written

```
TIX NAME +1
ETC ,4,1
```
or

```
TIX ETC NAME +1,4
ETC ,1
```
or

```
TIX ETC NAME +1,4,
```

but could not be written

```
TIX NA
ETC ME+1,4,1
```

The following operations may not be followed by an ETC card:

- ABS
- IFT
- ORGCRS
- BCI
- INDEX
- PCC
- DEC
- LABEL
- PGC
- DETAIL
- LBL
- PMC
- DUP
- LDIR
- PUNCH
- EJECT
- LIST
- QUAL
- END
- LORG
- REM
- ENDM
- NOCRS
- TITL
- ENQ
- NULL
- TITLE
- EVEN
- OCT
- UNLIST
- FUL
- OPD
- UNPNCH
- IFF
- OPSYN
- USE

If an ETC follows any of these operations except END, the ETC will be ignored and a low-severity warning message issued.

The REM Pseudo-Operation

The REM (Remarks) pseudo-operation permits remarks to be entered into the assembly listing. The format of the REM pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
</table>
| Any informa-
| tion | REM | Any information |

The contents of columns 8-10 (the operation field) are replaced by blanks, and the remaining contents of the card are copied onto the assembly listing. The REM pseudo-operation supplements the remarks card that
has * in column 1. In a macro-definition, the variable field of the REM card is scanned for substitutable parameters, whereas the * card causes an error message but is otherwise completely ignored.

The KEEP Pseudo-Operation

The KEEP pseudo-operation permits the programmer to specify a debugging dictionary that contains only those symbols he wishes to use in debug requests. The format of the KEEP pseudo-operation is:

<table>
<thead>
<tr>
<th>Name Field</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>KEEP</td>
<td></td>
</tr>
</tbody>
</table>

The symbols in the variable field are entered into the debugging dictionary along with any modal and dimensional information that was supplied in BSS, BES, EQU, and SYN pseudo-operations. Any number of KEEP pseudo-operations may appear in a program. If the NODE option was specified on the SIBMAP card, the KEEP pseudo-operation is ignored. (For further information concerning debugging and the debugging dictionary, see the publications IBM 7090/7094 ISYS Operating System: IBJOB Processor, Form C28-6389, and IBM 7090/7094 ISYS Operating System: IBJOB Processor Debugging Package, Form C28-6393.)

The FUL Pseudo-Operation

The FUL pseudo-operation specifies card output in the 24-word-per-card "full" mode. The format of the FUL pseudo-operation is:

<table>
<thead>
<tr>
<th>Name Field</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>FUL</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The FUL pseudo-operation always causes the next output word to start a new card. Any words remaining in the punch buffer are written out in the previously specified format. Cards produced in the column binary full mode contain the first word of output in columns 1-3; the second, in 4-6; and so on to a maximum of 24 words per card. No control words or check sums are generated by the assembler in full mode.

The PUNCH and UNPNCH Pseudo-Operations

The PUNCH and UNPNCH pseudo-operations cause resumption and suspension, respectively, of binary card punching. The format of the PUNCH and UNPNCH pseudo-operation is:

<table>
<thead>
<tr>
<th>Name Field</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>PUNCH or UNPNCH</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The TCD Pseudo-Operation

A binary transfer card directs an absolute loader program to stop loading cards and to transfer control to a designated location. In most cases, a transfer card is required at the end of the binary deck. In absolute assemblies, the END pseudo-operation causes a binary transfer card to be punched. However, the TCD pseudo-operation can cause a transfer card to be produced before the end of the binary deck.

The format of the TCD pseudo-operation is:

<table>
<thead>
<tr>
<th>Name Field</th>
<th>Operation Field</th>
<th>Variable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>TCD</td>
<td>A symbolic expression</td>
</tr>
</tbody>
</table>

The TCD pseudo-operation performs the following two functions:
1. Any binary output waiting in the punch buffer is written out.
2. A binary transfer card is produced having a transfer address that is the value of the expression in the variable field. See the publication IBM 7090/7094 ISYS Operating System: IBJOB Processor, Form C28-6389, for a description of the format of a transfer card.

If UNPNCH is in effect, no cards are punched.
List-Control Pseudo-Operations

The PCC Pseudo-Operation
The PCC (Print Control Cards) pseudo-operation has the following format:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>PCC</td>
<td>Either:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Blanks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Any Information</td>
</tr>
</tbody>
</table>

Unless the UNLIST pseudo-operation is in effect, PCC on causes the listing of the following cards: TTL, TITLE, LBL, LIST, INDEX, SPACE, EJECT, DETAIL, PCC, PMC, IFT, IFF, and COTO, and any cards under the scope of the IFT, IFF, and COTO pseudo-operations that are not assembled. PCC off suppresses listing of these cards and is the normal mode. The PCC card is always listed unless UNLIST is in effect. If the variable field is blank or contains anything other than ON or OFF, the current setting of the PCC switch is inverted.

The UNLIST Pseudo-Operation
The UNLIST pseudo-operation causes all listing to be suspended. The format of the UNLIST pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>UNLIST</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The UNLIST pseudo-operation is itself listed unless a previous UNLIST is still in effect. After an UNLIST, no lines are listed by the assembly program until a LIST or END pseudo-operation is encountered.

The LIST Pseudo-Operation
The LIST pseudo-operation causes listing to be resumed following an UNLIST. The format of the LIST pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>LIST</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The LIST pseudo-operation does not appear in the assembly listing unless the mode of PCC is ON.

The TITLE Pseudo-Operation
The TITLE pseudo-operation abbreviates the assembly listing by eliminating certain kinds of information. The format of the TITLE pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>TITLE</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

TITLE causes the assembly program to exclude the following information from the listing:
1. Any line that contains octal information except the instruction that causes it, i.e., all but the first word generated by OCT, DEC, BCI, and VFD
2. All but the entire first iteration of each instruction in the range of a DUP
3. All complex fields in a relocatable assembly
4. The expansion of SAVE and all but the first three instructions in the expansion of CALL.
5. All literals in the Literal Pool except the first
A TITLE pseudo-operation is effective until the assembly program encounters a DETAIL operation. TITLE is not listed except when the mode of PCC is ON.

The DETAIL Pseudo-Operation
The DETAIL pseudo-operation causes the listing of generated data to be resumed after it has been suspended by a TITLE pseudo-operation. The format of the DETAIL pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>DETAIL</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The sole effect of the DETAIL operation is to cancel the effect of a previous TITLE pseudo-operation. If TITLE is not in effect, the DETAIL operation is ignored by the assembly program. The DETAIL operation does not appear in the assembly listing unless the mode of PCC is ON.

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 format of the EJECT pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>EJECT</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The EJECT pseudo-operation appears in the assembly listing only if the mode of PCC is ON.

The SPACE Pseudo-Operation
The SPACE pseudo-operation permits one or more blank lines to be inserted in the assembly listing. The format of the SPACE pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>SPACE</td>
<td>1. A symbolic expression, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Blanks</td>
</tr>
</tbody>
</table>

The definition of the expression in the variable field determines the number of blank lines in the assembly
listing. If the value of the expression is zero or the variable field is blank, one blank line appears. SPACE itself is listed only if the mode of PCC is ON.

The LBL Pseudo-Operation

Serialization of a deck normally begins with the first four characters of the deck name, which are left-justified and filled with trailing zeros. However, serialization can be altered by using the LBL pseudo-operation. The format of the LBL (Label) pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>LBL</td>
<td>One or two subfields, separated by a comma:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. Up to 8 BCD characters, which may or may not be followed by a comma if the second subfield is not present</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. The word BEGIN followed by a blank or a comma</td>
</tr>
</tbody>
</table>

LBL causes binary cards to be identified and serialized in columns 73-80, as follows:

1. Serialization begins with the characters appearing in the variable field, which is left-justified and filled with terminating zeros.

2. Serialization is incremented by one for each card until the rightmost nonnumeric character or the seventh character is reached, after which the numeric portion recycles to zero. The two leftmost characters are regarded as fixed, even though they may be numeric.

For example, if the variable field is coded as ID, the first card is identified and serialized as ID000000.

If the variable field is coded as INSTR00, serialization is as follows:

INSTR030
INSTR031
.
.
INSTR999
INSTR000
.
.

If the BEGIN option is not included, reserialization begins with the binary card following the one that is currently being punched. Serialization can be altered at any point in the program by using additional LBL pseudo-operations.

If the BEGIN option is included, the LBL will be effective from the beginning of the program, serializing all binary cards, including the FILE and LABEL cards created by IBMAP as a result of the corresponding pseudo-operations, no matter where the LBL appears in the program. If more than one LBL pseudo-operation with the BEGIN option is used, only the last one will be effective. Any LBL pseudo-operations that do not contain the BEGIN option are processed normally.

If the variable field of the LBL currently in effect does not end with a comma, the assembly program prints the phrase

BINARY CARD ID.Number

at the beginning of each card. If a comma is used to terminate the variable field, printing of this phrase is suppressed. Printing of this phrase can be reinitiated by using an LBL pseudo-operation ending in a blank. LBL is listed only if the mode of PCC is ON.

The INDEX Pseudo-Operation

The INDEX pseudo-operation provides a table of contents of important locations within an assembly. The format of the INDEX pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>INDEX</td>
<td>Symbols, separated by commas</td>
</tr>
</tbody>
</table>

The first appearance of an INDEX card causes the message

**TABLE OF CONTENTS**

to be listed. Each subfield of an INDEX pseudo-operation causes the symbol and its definition to be listed. If a virtual symbol is used, its definition will be the control section number assigned to the symbol.

INDEX pseudo-operations may appear anywhere in the source program and need not be grouped. The listing generated by INDEX pseudo-operations is inserted where the pseudo-operations appear.

For meaningful commentary, INDEX pseudo-operations can be grouped and interspersed with explanatory remarks cards.

An INDEX pseudo-operation is not processed if any of the following three conditions exist:

1. It is in a macro-expansion and PMC is off.
2. It is in the range of a DOF pseudo-operation and a TITLE pseudo-operation is in effect.
3. An UNLIST pseudo-operation is in effect.

Listing of the INDEX card itself is governed by the mode of the PCC switch.

The variable field of an INDEX pseudo-operation cannot be extended by use of the ETC pseudo-operation.

The PMC Pseudo-Operation

The PMC pseudo-operation causes (or suppresses) listing of the card images generated by macro-instructions and by the RETURN pseudo-operation. The format of the PMC (Print Macro Cards) pseudo-operation is:
### Special Systems Pseudo-Operations

Users of the MAP language are provided with a wide range of subroutines which are included in the systems library. A group of system pseudo-operations permits the transfer of control and data between the main program and the subroutine. Details about specific calling sequences are provided in the publications *IBM 7090/7094 IIBSYS Operating System: IBMJ0B Processor*, Form C28-6389 and *IBM 7090/7094 Operating System: Input/Output Control System*, Form C28-6345.

### The CALL Pseudo-Operation

The CALL pseudo-operation produces the standard IBMJ0B subroutine calling sequence. The format of the CALL pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>CALL</td>
<td>One or more subfields; 1. Symbol or ** 2. Calling sequence parameters 3. Error returns 4. Identification number</td>
</tr>
</tbody>
</table>

The first subfield in the variable field must contain an unqualified symbol (the name of a subroutine) or **. The next subfield contains the parameters of the calling sequence (if any), enclosed in parentheses and separated by commas. These may be any symbolic expression.

Error returns (if any), separated by commas, occupy the next subfield. The last subfield is an identification number (if desired), less than 32,768 and delimited by apostrophes. If specified, this number appears in the calling sequence in place of the assembly line number. When an identification number is not specified, the assembly line number appears.

For example, a typical CALL operation might be coded

```
LCS CALL name(arg1, arg2, ..., argn) ETC ret1,ret2, ...,retn'id'
```

where name is the name of a subroutine; arg1, arg2, ..., argn are the parameters of the calling sequence; ret1, ret2, ..., retn are the error returns; and 'id' is the identification number.

A comma should not precede the left parenthesis, follow the right parenthesis, nor precede the 'id'.

If the subroutine is part of the program being assembled, the reference is to the routine in the program. However, if the subroutine is not part of the program being assembled, the symbol in the first subfield of the variable field becomes the external name of the subroutine called. If ** is used, a constant zero becomes the called address.
The remaining subfields generate the calling sequence.

### Expansions of the CALL Pseudo-Operation

The linkage produced by

| LCS | CALL | ETC | NAME(P1, ..., Pn)R1, ...Rm'11Y |

is

| LCS | TSX | NAME,4 |
| TXX | *+2+2+n,m,n |
| FZE | ID,,Linkage Director |
| FZE | P1 |
| Rm |
| TRA | Pn |
| R1 |

where P is a subroutine parameter, R is an error return, n is the number of parameters and m is the number of error returns. The Linkage Director is a location unique for each assembly, and has no associated symbol. It may be given a symbolic designation using the LDIN pseudo-operation.

The operation

| LCS | CALL | NAME(P1,P2) |

produces

| LCS | TSX | NAME,4 |
| TXX | *+2+2+0,2 |
| FZE | Line number,,Linkage Director |
| FZE | P1 |
| FZE | P2 |

The statement

| LCS | CALL | NAME,R1 |

produces

| LCS | TSX | NAME,4 |
| TXX | *+2+0+1,0 |
| FZE | Line number,,Linkage Director |
| TRA | R1 |

The statement

| LCS | CALL | NAME |

generates

| LCS | TSX | NAME,4 |
| TXX | *+2+0+0,0 |
| FZE | Line number,,Linkage Director |

### The SAVE Pseudo-Operation

The SAVE pseudo-operation produces the instructions necessary to save and restore the index registers and indicators, to disable and restore all operative traps, to provide error returns used by a subprogram, and to store the contents of index register 4 in SYSLOC and in the Linkage Director. The format of the SAVE pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>SAVE</td>
<td>Up to 7 subfields containing integers or immediate symbols and any or all of the letters I, D, E</td>
</tr>
</tbody>
</table>

The order in which the subfields in the variable field of the SAVE pseudo-operation are used is not important. All subfields are optional.

As many as 7 numeric subfields may be used to specify the index registers that are to be saved and restored. Immediate symbols may also be used to specify index registers. Any or all index registers may be specified in any order.

Index registers are saved in the order 4, 1, 2, 3, 5, 6, 7 and are restored in the opposite order. Index register 4 is automatically saved and restored, although it may still be specified.

One of the three remaining subfields is literally the character I; another is literally the character D; and the last is literally the character E.

The presence of I signifies that the sense indicators are to be saved and restored.

The presence of D causes all operative traps to be disabled and restored.

The presence of E generates the instructions necessary to facilitate use of error returns in the CALL pseudo-operation.

The contents of index register 4 are stored in the Linkage Director each time the SAVE pseudo-operation is executed. SYSLOC is a standard communication location used by all programs loaded under TDLIB. If the assembly is absolute (the ASSEMB option is specified), the symbol SYSLOC must be defined by the programmer.

The general form of the SAVE pseudo-operation is:

```
locsym SAVE (X1, ..., Xk)I,D,E
```

or

```
locsym SAVE X1, ..., Xk,I,D,E
```

### Expansions of the SAVE Pseudo-Operation

The instruction

```LCS SAVE 2,1,I```

or its equivalent

```LCS SAVE (2,1)I```

specifies that index registers 2 and 1 and the sense indicators are to be saved.
The expansion is:

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXI</td>
<td>.0003,0</td>
</tr>
<tr>
<td>AXT</td>
<td>**2</td>
</tr>
<tr>
<td>AXT</td>
<td>**1</td>
</tr>
<tr>
<td>.0001</td>
<td>AXT</td>
</tr>
<tr>
<td>LDI</td>
<td>.0002+1</td>
</tr>
<tr>
<td>.0002</td>
<td>TRA</td>
</tr>
<tr>
<td>PZE</td>
<td></td>
</tr>
<tr>
<td>.0003</td>
<td>STI</td>
</tr>
<tr>
<td>SXA</td>
<td>SYSLOC4</td>
</tr>
<tr>
<td>SXA</td>
<td>Linkage Director,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001-1,1</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001-2,2</td>
</tr>
</tbody>
</table>

The instruction

<table>
<thead>
<tr>
<th>LCS</th>
<th>SAVE 2</th>
</tr>
</thead>
</table>

generates

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXI</td>
<td>.0003,0</td>
</tr>
<tr>
<td>AXT</td>
<td>**2</td>
</tr>
<tr>
<td>.0001</td>
<td>AXT</td>
</tr>
<tr>
<td>.0002</td>
<td>TRA</td>
</tr>
<tr>
<td>SXA</td>
<td>SYSLOC4</td>
</tr>
<tr>
<td>SXA</td>
<td>Linkage Director,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001-1,2</td>
</tr>
</tbody>
</table>

In the next two examples, the instructions generated because of using the letters I, D, or E in the subfield of a SAVE pseudo-operation are identified by the appearance of the particular letter in the comments field of the generated instruction.

The instruction

<table>
<thead>
<tr>
<th>LCS</th>
<th>SAVE 2,I,D</th>
</tr>
</thead>
</table>

generates

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXI</td>
<td>.0003,0</td>
</tr>
<tr>
<td>AXT</td>
<td>**2</td>
</tr>
<tr>
<td>.0001</td>
<td>AXT</td>
</tr>
<tr>
<td>.0002</td>
<td>TRA</td>
</tr>
<tr>
<td>PZE</td>
<td></td>
</tr>
<tr>
<td>.0003</td>
<td>XEC</td>
</tr>
<tr>
<td>STI</td>
<td>.0002+1</td>
</tr>
<tr>
<td>SXA</td>
<td>SYSLOC4</td>
</tr>
<tr>
<td>SXA</td>
<td>Linkage Director,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001-1,2</td>
</tr>
</tbody>
</table>

Locations SYSDSB, .TRPSW, and .TRAPX are in the System Monitor (see the publication IBSYS Operating System: System Monitor (IBSYS), Form C28-6248). A switch at .TRPSW indicates whether enabling is permissible at this time; .TRAPX gives the address of the location that contains the bits for proper enabling. (Note that the enabling instruction below refers to .TRAPX indirectly.)

The following sequence illustrates the expansion that is generated when the E option is specified:

<table>
<thead>
<tr>
<th>LCS</th>
<th>SAVE (2)I,D,E</th>
</tr>
</thead>
</table>

generates

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXI</td>
<td>.0003,**</td>
</tr>
<tr>
<td>LDI</td>
<td>.0002+1</td>
</tr>
<tr>
<td>.0002</td>
<td>TRA</td>
</tr>
<tr>
<td>PZE</td>
<td></td>
</tr>
<tr>
<td>AXE</td>
<td></td>
</tr>
<tr>
<td>.0003</td>
<td>XEC</td>
</tr>
<tr>
<td>STI</td>
<td>.0002+1</td>
</tr>
<tr>
<td>SXA</td>
<td>SYSLOC4</td>
</tr>
<tr>
<td>SXA</td>
<td>Linkage Director,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001,4</td>
</tr>
<tr>
<td>SXA</td>
<td>.0001-1,2</td>
</tr>
</tbody>
</table>

If the SAVE pseudo-operation has no symbol in the name field, a symbol will be generated and an error message will be printed.

**The SAVEN Pseudo-Operation**

The SAVEN pseudo-operation produces the instructions necessary to save and restore the index registers used by a subroutine. The format of the SAVEN pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>SAVEN</td>
<td>Up to 10 subfields</td>
</tr>
</tbody>
</table>

The SAVEN pseudo-operation is similar to SAVE except that the instructions

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>LCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXA</td>
<td>Linkage Director,4</td>
</tr>
</tbody>
</table>

are not generated. SAVEN is generally used when entering a subroutine from another subroutine without destroying the linkage information. If the SAVEN pseudo-operation has no symbol in the name field, a symbol will be generated and an error message will be printed. If the variable field is blank, index register 4 is saved and restored.

**The RETURN Pseudo-Operation**

The RETURN pseudo-operation is designed for use with CALL and SAVE, making use of the error (or alternate) returns used in these operations.
The format of the RETURN pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
</table>
| 1. A symbol, or 2. Blanks | RETURN | 1 or 2 subfields separated by a comma:  
1. A symbol,  
2. An integer, a symbol, or an immediate symbol |

The first subfield in the variable field of the RETURN pseudo-operation is required. It contains the name of the associated SAVE pseudo-operation. If the second subfield is present, it specifies the particular error return.

The RETURN pseudo-operation often takes the general form

```
name RETURN locsym,i
```

where locsym is the symbolic address of the associated SAVE pseudo-operation, and i is the desired error return (i = 0 is the normal return).

The form of the RETURN instruction may vary. For example, to specify a particular error return (e.g., 2), the instruction

```
LOC   RETURN LCS,2
```

is written, where LCS is the location of the SAVE pseudo-operation to be used. The following instructions are generated:

```
LOC AXT 2,4
SXD LCS,4
TRA LCS+1
```

If the E option of the SAVE or SAVEN pseudo-operation is not used, the following form should be used:

```
RETURN LCS
```

which generates

```
TRA LCS+1
```

This form should also be used even where the E option is specified if the error return is inserted into the decrement of the SAVE or SAVEN pseudo-operation at execution time.

The variable field of the RETURN pseudo-operation may not be left blank, since it results in a TRA instruction with a blank variable field. The PMC pseudo-operation governs the listing of the instructions generated by the RETURN pseudo-operation.
Macro-operations are special types of pseudo-operations that provide the MAP user with a powerful programming tool. After a programmer has defined a macro-operation, he can cause a whole sequence of instructions to be called into a program by coding a single instruction. The sequence can be repeated as often as desired. Moreover, any field or subfield of any instruction in the sequence can be changed each time the sequence is repeated.

Any machine instruction, pseudo-operation, or macro-operation can be included in a macro-operation. The sequence of instructions generated (usually called a macro-expansion) is an open subroutine. The instructions are executed in-line with the rest of the program.

Two general requirements must be met to take advantage of the macro-operation facility. First, the macro-operation must be defined by a macro-definition. Then, wherever the sequence of instructions is desired in the program, it must be called by a macro-instruction.

**Defining Macro-Operations**

A macro-definition provides a name for the macro-operation, determines the instructions that will be included in the macro-expansion, and establishes the parts of the instructions that are to be variable.

Three kinds of instructions must be coded to define a macro-operation. The first is the MACRO pseudo-operation. (The card containing this instruction is sometimes called the macro-definition heading card.) Prototype instructions (sometimes called prototype card images) immediately follow the MACRO pseudo-operation to establish the instructions that will be generated in the macro-expansion. Finally, the ENDM pseudo-operation ends the macro-definition.

**The MACRO Pseudo-Operation**

The MACRO pseudo-operation establishes the name of a macro-operation. The format of the MACRO pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD name up to 6 characters long</td>
<td>MACRO</td>
<td>Up to 63 substitutable arguments (groups of not more than 6 characters) separated by punctuation characters</td>
</tr>
</tbody>
</table>

The name in the name field becomes the name of the macro-operation that is being defined. This name is later used to call the macro-operation and thus, in effect, becomes an operation code. Any valid symbol may be used in the name field of the MACRO pseudo-operation, or all numeric characters may be used. However, six zeros may not be used.

The name in the name field of a MACRO pseudo-operation may be the same as a symbol used anywhere in the program, even in this or any other macro-operation. However, if this name is the same as any other machine operation code, pseudo-operation code, or macro-operation code, the operation code is re-defined.

The subfields in the variable field of the MACRO pseudo-operation contain substitutable arguments.

**SUBSTITUTABLE ARGUMENTS IN THE MACRO PSEUDO-OPERATION**

Much of the flexibility of the macro-operation facility results from the principle of substitutable arguments. These subfields in the variable field of the MACRO pseudo-operation are dummy names that will be replaced in the macro-expansion.

Substitutable arguments permit any field or subfield of any instruction to be changed each time the macro-operation is called. The programmer can also change parts of subfields and even add entire instructions.

A substitutable argument is from one to six characters long. Any valid symbol may be used, and the name of a substitutable argument may consist of all numeric characters. For example, in the MACRO pseudo-operation

```
ALPHA MACRO ABC,123
```

each of the two groups of three characters in the variable field is a substitutable argument.

No punctuation characters except the period may be used as part of a substitutable argument.

A substitutable argument may be the same as a symbol or an operation code, including the operation code for this or any other macro-operation. However, substitutable arguments should not be identical to symbols or operation codes used in the prototype that immediately follows unless the symbols or operation codes are actually intended to be substitutable arguments.
DELIMITING SUBSTITUTABLE ARGUMENTS IN THE MACRO PSEUDO-OPERATION

Substitutable arguments in the variable field of the MACRO pseudo-operation may be separated by any of the following punctuation (special) characters: 

\[ +=-*/(),' \]

If parentheses are used, they must be used in pairs.

The use of these characters permits meaningful notation in a macro-definition. For example,

```
ALPHA MACRO 23,RATE,TIME,DIST, QUSYM
```

could also be written

```
ALPHA MACRO 23(RATE*TIME=DIST) QUSYM
```

The variable field of the MACRO pseudo-operation may be extended over more than one card by using the ETC pseudo-operation. When the substitutable arguments appear on more than one card, the blank character acts as a separator. Hence, no punctuation character is needed between consecutive substitutable arguments that appear on separate cards. For example,

```
BETA MACRO A,B,C
```

could also be written

```
BETA MACRO A,B ETC C
```

This usage of the ETC pseudo-operation differs from the usual case, in which all punctuation characters must be written.

Consecutive punctuation characters or an explicit zero are ignored and do not result in a substitutable argument of zero.

Prototypes in Macro-Definitions

The prototype of a macro-definition determines the instructions that will be included in the macro-expansion, their sequence in the expansion, and the positions of the substitutable portions of the instructions. The prototype, which consists of one or more prototype instructions, immediately follows the MACRO pseudo-operation.

A prototype instruction is similar to any other instruction. It has a name field, an operation field, and a variable field. It may also have a comments field, although this field does not appear in the card image generated in the macro-expansion. The distinguishing feature of a prototype instruction is that parts of it can be made variable.

The fields or subfields of a prototype instruction may contain text or substitutable arguments.

TEXT IN PROTOTYPES

Text represents the fixed parts of the instructions that will be generated in the macro-expansion. Any part of a prototype instruction that has not been made a substitutable argument by its appearance in the variable field of the MACRO pseudo-operation is treated as text. For example, in the prototype

```
ALPHA MACRO A,B
CLA A
B BUFFER
```

the operation code CLA and the location BUFFER are text. (BUFFER has been defined elsewhere in the program.)

Text is reproduced in the macro-expansion exactly as it appears in the prototype instruction. Thus, if only text is used in a field of an instruction, it must conform to the rules governing that field of the instruction in which it is used. For example, if the operation field of a prototype instruction is text, it must be a valid operation code.

Since parentheses can be used to delimit substitutable arguments within the prototype, parentheses must be used carefully as part of text to avoid confusing the enclosed characters with a substitutable argument.

SUBSTITUTABLE ARGUMENTS IN PROTOTYPES

Substitutable arguments represent the variable parts of the instructions that will be generated in the macro-expansion. The same substitutable arguments are used in the prototype that appeared in the variable field of the MACRO pseudo-operation. However, in the prototype, substitutable arguments appear in the fields or subfields of the prototype instructions that are to be variable. A substitutable argument may appear in any field or subfield of a prototype instruction. For example, in the sequence

```
BETA MACRO ONE,TWO,THREE
ONE CLA PART1
TWO PART2
STO THREE
```

ONE, TWO, and THREE are substitutable arguments in the name, operation, and variable fields, respectively, of prototype instructions.

DELIMITING SUBSTITUTABLE ARGUMENTS IN PROTOTYPES

The same punctuation (special) characters may be used in prototype instructions that were used to separate the substitutable arguments in the variable field of the MACRO pseudo-operation. Except for the apostrophe, these characters are reproduced in the macro-expansion. Only the apostrophe may be used to delimit substitutable arguments in the variable fields of REM
and TTL pseudo-operations and the data subfields of BCI pseudo-operations. (Another use of the apostrophe in macro-operations is explained in the section "Combining Substitutable Arguments and Text.")

A comma or a left parenthesis immediately following the operation code (as near the beginning of the card as column 11) signifies the end of the operation field and the beginning of the variable field.

A blank delimits a substitutable argument in a prototype. For example, in the macro-definition (where b represents a blank)

\text{XYZ MACRO A,B,C AbbbB}

three blanks separate A (in the operation field) from B (in the variable field). It is not always necessary that three blanks separate these two fields, but at least three characters are used for an operation field code. In this example, A and two blanks are used, whereas the third blank is required to terminate the operation field. If fewer blanks separated A from B, both would be taken as part of the operation code, causing errors.

If a blank is encountered before card column 72 in the variable field of a prototype instruction other than a BCI, REM, or TTL pseudo-operation, the card is terminated. Substitutable arguments may appear anywhere from column 1 through column 72 on BCI, REM, and TTL cards. Any information to the right of the blank will not be included in the macro-definition.

If a BCI, TTL, or REM pseudo-operation has been redefined in a macro-definition or by an operation-defining pseudo-operation, it should not be used within a macro-definition. Also, if an operation code has been defined to have the effect of a BCI, TTL, or REM pseudo-operation and is used within a macro-definition, the variable field will be terminated by a blank.

Every field or subfield of six or fewer characters in any field of a prototype instruction is compared with the substitutable arguments in the variable field of the MACRO pseudo-operation. Therefore, care must be taken to avoid confusing fields intended as text with fields intended as substitutable arguments.

Cards with an asterisk (*) in column 1 (remarks cards) may be interspersed with macro-definition cards but will not be included in the macro-expansion.

The ENDM Pseudo-Operation

The ENDM pseudo-operation terminates a macro-definition. The format of the ENDM pseudo-operation is:

\text{TABLE}

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>ENDM</td>
<td>Either: 1. One or two subfields, separated by a comma, or 2. Blanks</td>
</tr>
</tbody>
</table>

the ENDM pseudo-operation is coded so that symbol creation is suppressed whenever this macro-operation is called.

If the variable field or the first subfield of the variable field is blank, this and all unterminated macro-definitions are terminated (see the section "Nested Macro-Operations").

Calling Macro-Operations

After a macro-operation has been defined, it may be called so that the generated sequence of instructions is brought into a program at a desired point. In the macro-expansion, each prototype instruction in the macro-definition is reproduced. Text and all punctuation characters except the apostrophe are reproduced exactly as they appeared in the prototype. However, the substitutable arguments that appeared in the variable field of the MACRO pseudo-operation and in the prototype are replaced by the actual parameters that the programmer wishes to appear in the expansion. These parameters are provided in the macro-instruction, which is used to call the macro-operation.

The Macro-Instruction

The macro-instruction calls a previously defined macro-operation into a program. The format of the macro-instruction is:
<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A symbol, or</td>
<td>2. Blanks</td>
<td>Parameters, separated by commas or enclosed in parentheses.</td>
</tr>
<tr>
<td>Macro-op</td>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>

If there is a symbol in the name field, it is assigned to the first instruction of the macro-expansion.

The name that was assigned in the name field of the MACRO pseudo-operation is an operation code and is used in the operation field of the macro-instruction.

The variable field of the macro-instruction contains the actual parameters that the programmer wishes to appear in the macro-expansion.

**PARAMETERS IN MACRO-INSTRUCTIONS**

The parameters in the variable field of the macro-instruction replace the substitutable arguments that appeared in the macro-definition. These parameters must appear in the same order as the substitutable arguments they are to replace originally appeared in the variable field of the MACRO pseudo-operation.

The length of a parameter in the variable field of a macro-instruction is not limited to six characters as is a substitutable argument. The variable field of a macro-instruction may be extended over more than one card by using the ETC pseudo-operation. However, a single parameter must appear completely on one card unless it is enclosed in parentheses.

Macro-instruction parameters consist of any appropriate character or group of characters that would normally appear in the particular instruction. For example, in the macro-definition,

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

mnemonic symbols represent the substitutable arguments. LOOP appears in a name field, OP in an operation field, and COEFF, DEC, and T appear in subfields of the variable field. GAMMA is text and not a substitutable argument, since it does not appear in the variable field of the MACRO pseudo-operation. In this macro-instruction, a symbol should be substituted for LOOP and a valid operation code for OP. Such a macro-instruction might be written

```
X015     QPOLY     C1-4,FIRST,54,FAD
```

The macro-expansion would cause the following six card images to be generated.

```
X015     AXT       5,4
LDQ       C1-4
```

**FIRST**     **FMP**     **GAMMA**
FAD         C1-4+5+1,4
XCA         FIRST,4,1
TIX

The symbol X015 is assigned to the first instruction, and each substitutable argument is replaced by the corresponding parameter that appeared in the variable field of the macro-instruction.

**DELIMITING MACRO-INSTRUCTION PARAMETERS**

The parameters in the variable field of a macro-instruction are separated either by commas or parentheses. A single comma following a right parenthesis or preceding a left parenthesis is redundant and may be omitted. Neither of these combinations results in a null parameter. A null parameter is indicated by two consecutive commas or by a single comma at the beginning of the variable field. If the blank terminating the variable field is preceded by a comma, the last subfield is not null (see the example in the section "Created Symbols"). An explicit zero must be used to obtain a zero parameter.

Parentheses around data used as a macro-instruction parameter signify that everything within the parentheses, including blanks, is to replace the corresponding substitutable argument in the prototype. (In fact, if blanks are to be included in a macro-instruction parameter, the parameter must be enclosed in parentheses.) For example, the macro-definition

```
XYZ      MACRO     A,B
         A
         B
         ENDM     XYZ
```

deprecated by the macro-instruction

```
XYZ      (AXTbbb10,1)
         (ALPHAAbbbBET1)
```

results in the expansion

```
ALPHA     AXT       10,1
         TRA       BETA,1
```

If parentheses are to appear in a macro-expansion, they must be enclosed in an outer pair of parentheses. The outer parentheses are removed in the expansion.

Pairs of parentheses must be balanced. For example, given the macro-definition

```
CALLIO    MACRO     ICOM,T1,OP,LABEL,T2
         ETC       TAPNO,PFX,ERRET
         TIX       (TAPE),4
         PZE       ICOM,T1,OP
         PZE       LABEL,T2,TPNO
         IFT       ERRET+1
         PFX       ERRET
         ENDM      CALLIO
```

and using the following parameters in place of the substitutable arguments
IOCOM = CITIO  T2 = null
T1 = 2  TAPNO = CITTAP
OP = (RBEP)  PFX = null
LABEL = CITLB  ERRRET = null

the corresponding macro-instruction would be

CALLIO  CITIO2,(RBEP),CITLB
ETC  "CITTAP"

This macro-instruction could also be written

CALLIO  CITIO2,(RBEP),CITLB
ETC  "CITTAP"

since the commas around (RBEP) are redundant.

Note that TAPE must not be a substitutable argument and that (RBEP) must be enclosed in outer parentheses. Also, an explicitly null parameter appears in the macro-instruction at a position corresponding to the substitutable argument ERRRET in the macro-definition. This null parameter causes the fifth word of the expansion to be omitted.

**Inserting Instructions into Macro-Expansions**

A single parameter in a macro-instruction may include more than one field or even an entire instruction to replace a single substitutable argument that appeared in a field of a prototype instruction. The parameter is inserted into the field in which the original substitutable argument appeared, and it may extend to other fields to the right of the field in which it is inserted. For example, if a substitutable argument appeared in the operation field of a prototype instruction, a parameter could be inserted that would have an operation field and a variable field.

When a parameter consists of more than one field or is an entire instruction, the programmer must provide enough blanks in the parameter so that the fields of the instruction appear in their proper positions in the macro-expansion.

In the following example, a substitutable argument in the operation field of a prototype instruction is replaced by an instruction having an operation field and a variable field. The macro-definition.

```
XYZ  MACRO  A,B,C,
CLA  A
B
STO  C
ENDM  XYZ
```

followed by the macro-instruction

```
SUM  XYZ  ALPHA(ADDbbbbbbBETA)
ETC  GAMMA
```

results in the macro-expansion

```
CLA  ALPHA
ADD  BETA
STO  GAMMA
```

**Conditional Assembly in Macro-Operations**

The IFT and IFF pseudo-operations may be used to determine whether instructions within a macro-expansion are assembled. For example, the sequence

```
ADDM  MACRO  B,C,D
CLA  B
ADD  C
IFF /D/=/AC/
STO  D
```

allows the sum to be stored if the name substituted for D is not literally the characters AC and prevents it from being stored if the name is the characters AC.

If the IFF pseudo-operation in the above example were replaced by

```
IFT  D=1
```

the sum would be stored only if the parameter substituted had already appeared in the name field of some instruction (i.e., if the S-value of D is 1).

The two conditions can be combined to obtain

```
IFF /D/=/AC,/AND
IFT  D=1
```

which assembles the store operation only if D is not literally AC and has appeared before in a name field.

**Combining Substitutable Arguments and Text**

The apostrophe can be used to combine substitutable arguments and text to form a single prototype subfield. The apostrophe delimits a substitutable argument in a macro-definition prototype but is not itself included in the macro-expansion. However, the apostrophe may not be used to combine partial subfields in lower-level nested macro-definitions (see the section "Nested Macro-Operations").

For example, given the macro-definition

```
ALPHA  MACRO  A,B,C
BCI  A,bbbbBERROR,b
CONDITION'C'b
IGNORED
ENDM  ALPHA,NOCRS
```

the macro-instruction

```
ALPHA  6,(FIELD),
```

causes the following instruction to be generated:

```
BCI  6,bbFIELDbERROR,b
CONDITIONb
IGNORED
```

By using the apostrophe, instructions within macro-operations can be altered and even name field symbols can be changed. For example, the macro-definition

```
FXCY  MACRO  B,W,Z,Y,T
N'B  PXX  Y,T
PAC  0,4
'WXXZ  Y,4
ENDM  FXCY
```

after the macro-instruction

```
FXCY  AME,S,A,DATA,1
```

The Macro-Operation Facility  45
results in the sequence

<table>
<thead>
<tr>
<th>NAME</th>
<th>MACRO</th>
<th>DATA,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PXA</td>
<td>0,4</td>
<td>DATA,4</td>
</tr>
</tbody>
</table>

The name field may not exceed six characters in the prototype, including substitutable arguments, text, and punctuation characters. The operation field may not exceed six characters or six characters and an asterisk.

**Nested Macro-Operations**

A macro-definition may be included completely within the prototype of another higher-level macro-definition. When the higher-level macro-operation is expanded, the MACRO pseudo-operation and the prototype of the lower-level macro-operation are generated. Thus, a macro-instruction for a lower-level macro-operation cannot be used until all higher-level macro-operations have been expanded.

A new macro-operation may be defined or an existing macro-operation redefined, depending on whether the name of the lower-level macro-operation appears as text or as a substitutable argument in the higher-level macro-operation. For example, in the macro-definition

```
MAC1  MACRO  MAC2,ALPHA,BETA
ETC    GAMMA,DELTA
MAC2  MACRO  ALPHA
BETA   A
GAMMA  B
DELTA  C
ENDM   MAC2
ENDM   MAC1
```

the lower-level macro-operation, MAC2, appears as a substitutable argument. The macro-instruction

```
MAC1  ABC, (A,B,C), CLA, ADD, STO
```

generates

```
ABC  MACRO  A,B,C
CLA  A
ADD  B
STO  C
ENDM ABC
```

which defines a macro-operation, ABC, where A, B, and C are substitutable arguments; and CLA, ADD, and STO are text.

However, had MAC2 appeared as text rather than as a substitutable argument in the macro-definition of MAC1, MAC2 would be redefined each time MAC1 was expanded.

There is no significant limit to the depth of nesting permitted.

**Macro-Instructions in Macro-Definitions**

The prototype of a macro-definition may include macro-instructions for which macro-operations have not yet been defined. However, these macro-instructions must be defined before using a macro-instruction that expands the macro-operation. Care must be exercised to avoid a macro-definition loop. For example, if macro-operation A contains an unconditional call on macro-operation A (a macro-instruction for A), a macro-definition loop will result. However, if the call on macro-operation A is conditional (i.e., there is provision for eventual exit from the loop), a macro-operation A may contain a macro-instruction for A.

Data enclosed within parentheses may be used as a parameter in a macro-instruction. When a macro-instruction is used within another macro-definition, special handling of data containing blanks is required. Such data must be replaced by a substitutable argument in the outer macro-definition. The actual data must appear as a parameter in the macro-instruction. An additional pair of parentheses must surround the data (already within a single pair of parentheses) to ensure proper substitution.

For example, in the macro-definition

```
MAC1  MACRO  A,C
CLA   C
MAC   A,(AXTb**4b)
STO   A
ENDM  MAC1
```

MAC is a previously defined macro-operation. Proper substitution would not occur, because a blank terminates the variable fields of prototype instructions except for the BCI, REM, and TTL pseudo-operations. Instead, the following sequence should be used:

```
MAC1  MACRO  A,C,D
CLA   C
MAC   A,D
STO   A
ENDM  MAC1
```

When using the macro-instruction MAC1, the substitution would be

```
MAC1  X,Y,((AXTb**4b))
```

The resulting generated sequence is

```
CLA   Y
MAC   X,(AXTb**4b)
STO   X
```

The macro-instruction for MAC can now cause the data to be substituted properly.

**Qualification Within Macro-Operations**

The qualification in effect when the macro-instruction is used will be used for symbols in the macro-expansion. If a qualification symbol is required within a macro-definition, it may be a substitutable argument, as may the symbols it qualifies.

If a macro-expansion having a qualified section falls within the range of a qualified section in the program, the rules for nested qualification apply when referring to symbols within the macro-expansion.
**Macro-Related Pseudo-Operations**

The **IRP** pseudo-operation is used to supplement the definition of macro-operations, whereas **ONC**s and **NOC**s are used in macro-expansions.

**The IRP Pseudo-Operation**

The **IRP** (Indefinite Repeat) pseudo-operation causes a sequence of instructions within a macro-operation to be repeated with one parameter varied at each repetition. The format of the **IRP** pseudo-operation is:

<table>
<thead>
<tr>
<th>NAME FIELD</th>
<th>OPERATION FIELD</th>
<th>VARIABLE FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanks</td>
<td>IRP</td>
<td>1. A single substitutable argument, or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Blanks</td>
</tr>
</tbody>
</table>

To repeat a sequence of instructions, two **IRP** pseudo-operations are required within a macro-definition — one to initiate the sequence and the other to end it. A single substitutable argument that originally appeared in the variable field of the preceding **MACRO** pseudo-operation must appear in the variable field of the initial **IRP** pseudo-operation. The variable field of the second **IRP** pseudo-operation is left blank.

For example, the operation

```
IRP A
```

initiates a sequence, whereas

```
IRP
```

ends the sequence.

Substitutable argument **A** governs the iteration of the instructions. If a symbol other than a single substitutable argument or more than one subfield appears in the variable field of the first **IRP** pseudo-operation, the pseudo-operation is ignored and a warning message is issued.

In the macro-instruction, substitutable argument **A** is replaced by one or more subarguments enclosed in parentheses and separated by commas. Each time the macro-instruction is used, the assembly program generates the sequence of instructions with each of the subarguments used successively in place of suitable argument **A**. If only one subargument is used, the sequence of instructions will be generated only once. If no subarguments are given, the whole sequence will be skipped. If a blank appears within the parentheses, only the arguments to the left of the blank will be effective. For example, given the macro-definition

```
XYZ \ MACRO ARG,B
IRP \ ARG
CLA \ ARG
ADD \ B
STO \ ARG
IRP
ENDM \ XYZ
```

the macro-instruction

```
XYZ (J,K,L),CONST
```

generates

```
CLA J
ADD CONST J \ (First iteration, with subargument J)
STO J
CLA K \ (Second iteration, with subargument K)
ADD CONST K
STO K
CLA L \ (Third iteration, with subargument L)
ADD CONST L
STO L
```

If the substitutable argument does not appear between the two **IRP** pseudo-operations, the generated sequences will be identical, their number depending on the number of subarguments given.

For example, given the macro-definition

```
BBB \ MACRO C,D,E
IRP \ C
CLA \ D
STO \ E
IRP
ENDM \ BBB
```

the macro-instruction

```
BBB (1,2,3),DATA1,DATA2
```

generates

```
CLA \ DATA1
STO \ DATA2
CLA \ DATA1
STO \ DATA2
CLA \ DATA1
STO \ DATA2
```

An **IRP** pseudo-operation may not occur explicitly within the range of another **IRP**. Such a nested pair causes termination of the first range and opening of a second range. However, a macro-instruction within the range of an **IRP** pseudo-operation may itself cause pairs of **IRP** pseudo-operations to be generated at a lower level.

**Created Symbols**

If parameters are missing from the end of the variable field of the macro-instruction, symbols are created to fill the vacancies. These symbols take the form

```
..0001
..0002
...
..nnnn
```

No created symbols are supplied for an explicitly null argument. Created symbols are supplied only at the end of the parameters.

For example, if the **MACRO** pseudo-operation

```
ALPHA \ MACRO A,B,C
```

is followed by the macro-instruction

```
ALPHA \ X,,
```

The Macro-Operation Facility  47
substitutable argument A is replaced by X, substitutable argument B is omitted since the parameter is explicitly void, and substitutable argument C is replaced by a symbol of the form ... mmm. This symbol is created to replace the omitted parameter at the end of the variable field of the macro-instruction.

Given the macro-definition

```
XFAD MACRO N,B,C,D,E,A,X
SXA   A,4
AXT   N,4
X     CLA B,4
IFF /C/=//
F'C' D,4
STO   E,4
TIX   X,4,1
A     AXT **,4
ENDM   XFAD,CBS
```

the macro-instruction

```
XFAD 4,DATA,AD,DATA1,DATA2
```

generates

```
SXA  .0001,4
AXT  4,4
CLA  DATA,4
FAD  DATA1,4
STO  DATA2,4
TIX  .0002,4,1
AXT  **,4
```

However, the macro-instruction

```
XFAD 4,DATA,,DATA2
```

generates

```
SXA  .0001,4
AXT  4,4
CLA  DATA,4
STO  DATA2,4
TIX  .0002,4,1
AXT  **,4
```

In this example, the number of instructions can vary between references, making a relative-address reference difficult. However, by permitting the assembly program to generate the names, the references are correct without requiring programmer-specified names.

The NOCRS Pseudo-Operation

The nocas pseudo-operation suppresses symbol creation, which causes missing parameters at the end of the variable field of the macro-instruction to be treated as if they were explicitly null. The format of the NOCRS pseudo-operation is:

```
NAME FIELD   OPERATION FIELD   VARIABLE FIELD
Blanks       NOCRS            Ignored
```

The ORGCRS Pseudo-Operation

The oncas pseudo-operation may be used to alter the form of created symbols. This pseudo-operation also causes symbol creation to be resumed if it has been suppressed by a nocas. The format of the ORGCRS pseudo-operation is:

```
NAME FIELD   OPERATION FIELD   VARIABLE FIELD
Blanks       ORGCRS            1. Blanks, or
2. Up to 4 numeric digits, or
3. One BCD character and
   up to 4 numeric digits.
```

If a BCD character appears in the variable field, it replaces the second period of the created symbols. If digits appear, they will be the origin of the new set of created symbols. This origin will be one lower than the first symbol actually created. If fewer than four digits are used, they will be right-justified with leading zeros. If no digits are supplied or if the variable field is blank, the number will continue from the last created symbol.

With a blank variable field, the oncas pseudo-operation causes resumption or continuation of symbol creation.

The PURGE Pseudo-Operation

The purge pseudo-operation removes from core storage prototypes of macro-operations that are no longer needed, thus providing space for other prototypes. However, the purge pseudo-operation does not remove the name of the macro-operation from the name table. The format of the PURGE pseudo-operation is:

```
NAME FIELD   OPERATION FIELD   VARIABLE FIELD
Blanks       PURGE             A list of macro-operation names,
                          separated by commas.
```

If an entry in the variable field of a PURGE pseudo-operation is not the name of a macro-operation, it is ignored and no message is issued.

If a macro-operation is redefined, purging of the previous form is automatic (a PURGE pseudo-operation is not needed).

If a call is made to a purged macro-operation, the call will be ignored and an error message will be issued.
Appendix A: Machine Operations

All machine operations recognized by MAP are tabulated in this appendix, including supplementary information about their format and use. Listings are provided of the 7090 machine operations, extended machine operations, special operations, 7094 machine operations, 7909 data channel commands, 1301 disk file orders, and 7340 Hypertape orders.

The code letters used under identical column headings have the same significance in all tables. The BCD name of the machine operation is given in the column headed Mnemonic.

Type indicates machine-instruction format characteristics by code letters having the following meanings:

<table>
<thead>
<tr>
<th>CODE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>15-bit decrement field</td>
</tr>
<tr>
<td>B</td>
<td>No decrement field</td>
</tr>
<tr>
<td>C</td>
<td>8-bit decrement field</td>
</tr>
<tr>
<td>D</td>
<td>18-bit address field</td>
</tr>
<tr>
<td>E</td>
<td>13-bit address field</td>
</tr>
<tr>
<td>K</td>
<td>4-bit prefix field</td>
</tr>
<tr>
<td>L</td>
<td>Disk orders</td>
</tr>
</tbody>
</table>

The codes in the address (Addr), tag, and decrement (Decr) columns signify:

<table>
<thead>
<tr>
<th>CODE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Subfield required. If missing, error message FIELD REQUIRED is issued.</td>
</tr>
<tr>
<td>P</td>
<td>Subfield permitted. Neither its presence nor absence results in a message.</td>
</tr>
<tr>
<td>U(n)</td>
<td>Subfield unexpected. If present, message FIELD NOT EXPECTED is issued. Assembly program truncates definition value of subfield to number of bits (n) shown in parentheses and, treating this value as a constant, adds it to value normally appearing in subfield.</td>
</tr>
<tr>
<td>N</td>
<td>Subfield not allowed. If present and its value is not zero, message FIELD NOT ALLOWED is issued. Its value is always treated as a constant zero.</td>
</tr>
</tbody>
</table>

If the size of a subfield differs from normal, field size is indicated in parentheses following the subfield letter code.

In the column headed Ind, the letter (P) indicates that indirect addressing is permitted and (N) indicates that is not permitted.

7090 Machine Operations

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>TYPE</th>
<th>ADDR</th>
<th>TAG</th>
<th>DECR</th>
<th>IND</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ADD</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ADM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ALS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>ANA</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ANS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ARS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>AXC</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>AXT</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>BSF</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>BSR</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>BTT</td>
<td>E</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>CAL</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>CAQ</td>
<td>C</td>
<td>R</td>
<td>P</td>
<td>U(8)</td>
<td>N</td>
</tr>
<tr>
<td>CAS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>CHS</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>CLA</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>CLM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>CLS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>COM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>CRQ</td>
<td>C</td>
<td>R</td>
<td>P</td>
<td>U(8)</td>
<td>N</td>
</tr>
<tr>
<td>CVR</td>
<td>C</td>
<td>R</td>
<td>P</td>
<td>U(8)</td>
<td>N</td>
</tr>
<tr>
<td>DCT</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>DVH</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>DVP</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ECTM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>EFTM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>ENB</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ENK</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>ERA</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ESN</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>ESTM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>ETM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>ETT</td>
<td>E</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>FAD</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>FAM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>FDH</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>FDP</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>FMP</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>FRN</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>FSB</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>FSM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>HPR</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>HTR</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>IIA</td>
<td>B</td>
<td>P</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>III</td>
<td>D</td>
<td>R(18)</td>
<td>N</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>IIR</td>
<td>D</td>
<td>R(18)</td>
<td>N</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>IIU</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>IOT</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>IOCD</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>IOCP</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>IORP</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>IOCT</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>IORT</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>IOSP</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>IOST</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>LAC</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LAS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>LBT</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LCH</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>LDC</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LDI</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>LDQ</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4)</td>
<td>P</td>
</tr>
<tr>
<td>LFT</td>
<td>D</td>
<td>R(18)</td>
<td>N</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LFTM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LGL</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LGR</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LLS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LNT</td>
<td>D</td>
<td>R(18)</td>
<td>N</td>
<td>U(6)</td>
<td>N</td>
</tr>
<tr>
<td>LRS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(6)</td>
<td>N</td>
</tr>
</tbody>
</table>
Extended Operations

MAP provides a group of sense and select-type extended operation codes for programmer convenience. These codes permit the address portion of certain instructions to be specified symbolically as part of the operation code, rather than octally in the address portion of the instruction. These codes also provide more meaningful mnemonics for some machine instructions.

MAP also recognizes a group of prefix codes that can be used in such programming applications as forming constants or in subroutine calling sequences.

SENSE TYPE

The following extended operation codes of the sense type are recognized by the assembly program. The tag subfield is permitted in all these operation codes, but indirect addressing is not permitted in any of them. The letter x is to be replaced by one of the channel letters.

SELECT TYPE

The following extended operation codes of the select type are recognized by the assembly program. The tag subfield is permitted in all these operation codes, but
indirect addressing is not permitted in any of them. If the following instructions are used in a relocatable deck, the address field must contain an integer or an immediate symbol. The letter x is to be replaced by a channel letter.

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>ADDR</th>
<th>DECR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSRx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>BSFx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>RCxDx</td>
<td>N</td>
<td>U(6)</td>
</tr>
<tr>
<td>REWx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>RPRx</td>
<td>N</td>
<td>U(6)</td>
</tr>
<tr>
<td>RTBx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>RTDx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>RUNx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>SDIx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>SDLx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>WEFx</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>WPBx</td>
<td>N</td>
<td>U(6)</td>
</tr>
<tr>
<td>WPdx</td>
<td>N</td>
<td>U(6)</td>
</tr>
<tr>
<td>WPux</td>
<td>N</td>
<td>U(6)</td>
</tr>
<tr>
<td>WTX</td>
<td>R</td>
<td>U(6)</td>
</tr>
<tr>
<td>WTDx</td>
<td>R</td>
<td>U(6)</td>
</tr>
</tbody>
</table>

**PREFIX CODES**

The following prefix codes are recognized by the assembly program. All have 15-bit decrement fields (Type A) with address, tag, and decrement permissible. Indirect addressing is permitted for all prefix codes.

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>MEANING</th>
<th>PREFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIVE</td>
<td>Five</td>
<td>5</td>
</tr>
<tr>
<td>FOR</td>
<td>Four</td>
<td>4</td>
</tr>
<tr>
<td>FOUR</td>
<td>Four</td>
<td>4</td>
</tr>
<tr>
<td>FVE</td>
<td>Five</td>
<td>5</td>
</tr>
<tr>
<td>MON</td>
<td>minus One</td>
<td>5</td>
</tr>
<tr>
<td>MTH</td>
<td>minus Three</td>
<td>7</td>
</tr>
<tr>
<td>MTW</td>
<td>minus Two</td>
<td>6</td>
</tr>
<tr>
<td>MZE</td>
<td>minus Zero</td>
<td>4</td>
</tr>
<tr>
<td>ONE</td>
<td>One</td>
<td>1</td>
</tr>
<tr>
<td>PON</td>
<td>Plus One</td>
<td>1</td>
</tr>
<tr>
<td>PTH</td>
<td>Plus Three</td>
<td>3</td>
</tr>
<tr>
<td>PTV</td>
<td>Plus Two</td>
<td>2</td>
</tr>
<tr>
<td>PZE</td>
<td>Plus Zero</td>
<td>0</td>
</tr>
<tr>
<td>SEVEN</td>
<td>Seven</td>
<td>7</td>
</tr>
<tr>
<td>SIX</td>
<td>Six</td>
<td>6</td>
</tr>
<tr>
<td>SVN</td>
<td>Seven</td>
<td>7</td>
</tr>
<tr>
<td>THREE</td>
<td>Three</td>
<td>3</td>
</tr>
<tr>
<td>TWO</td>
<td>Two</td>
<td>2</td>
</tr>
<tr>
<td>ZERO</td>
<td>Zero</td>
<td>0</td>
</tr>
</tbody>
</table>

**Special Operations**

The following special operations recognized by MAP reduce coding effort for frequently occurring conditions. Indirect addressing is not permitted in any of these operations.

| MNEMONIC | TYPE | ADDR | TAG | ASSEMBLES
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BFT</td>
<td>D</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BNT</td>
<td>D</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BRA</td>
<td>A</td>
<td>N</td>
<td>N</td>
<td>P</td>
</tr>
<tr>
<td>...</td>
<td>A</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>...</td>
<td>A</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>IIB</td>
<td>D</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RIB</td>
<td>D</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SIB</td>
<td>D</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ZAC</td>
<td>B</td>
<td>U</td>
<td>N</td>
<td>U(6) PDX</td>
</tr>
<tr>
<td>ZSA</td>
<td>B</td>
<td>R</td>
<td>N</td>
<td>U(6) SXA</td>
</tr>
<tr>
<td>ZSD</td>
<td>B</td>
<td>R</td>
<td>N</td>
<td>U(6) SXD</td>
</tr>
</tbody>
</table>

(See the section "The RBOOL and LBOOL Pseudo-Operations" for a description of special type D instructions.)

**7094 Machine Operations**

The following 7094 machine operations are recognized by MAP. In a 7090 assembly, these operations are replaced by system macro-instructions (see Appendix B) with the exceptions noted below. The 7090 system macro-instructions closely approximate the effects of the 7094 machine operations but may cause slight differences in the precision of the results obtained.

If a subfield is to be supplied at execution time, the notation ** must be used. The field may not be blank.

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>TYPE</th>
<th>ADDR</th>
<th>TAG</th>
<th>ASSEMBLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFAD</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DFAM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DFDH</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DFDP</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DFMP</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DFSB</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DSFS</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DFM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DLD</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DST</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>*DUAM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>*DUFN</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>*DUFM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>*DUSF</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>DUSM</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>*EMTM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6) N</td>
</tr>
<tr>
<td>*LMTM</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6) N</td>
</tr>
<tr>
<td>PCA</td>
<td>B</td>
<td>P</td>
<td>R</td>
<td>U(6) N</td>
</tr>
<tr>
<td>PCD</td>
<td>B</td>
<td>P</td>
<td>R</td>
<td>U(6) N</td>
</tr>
<tr>
<td>SCA</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>U(6) N</td>
</tr>
<tr>
<td>SCD</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>U(6) N</td>
</tr>
</tbody>
</table>

*These 7094 instructions are not replaced by macro-instructions when assembling in the 7090 mode; they are assembled as NOP instructions, and a low-severity message is issued.

**IBM 7909 Data Channel Commands**

The following 7909 data channel commands are recognized by the assembly program. An x in the operation code is to be replaced by a channel letter.

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>TYPE</th>
<th>ADDR</th>
<th>TAG</th>
<th>ASSEMBLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYD</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
</tr>
<tr>
<td>CPYP</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>R</td>
</tr>
<tr>
<td>CTL</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CTLR</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CTLW</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(1)ICC</td>
<td>K</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LAR</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LCC</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LIP</td>
<td>K</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LIPT</td>
<td>A</td>
<td>R</td>
<td>N</td>
<td>U(4) P</td>
</tr>
<tr>
<td>RICx</td>
<td>E</td>
<td>N</td>
<td>P</td>
<td>U(6) N</td>
</tr>
<tr>
<td>RSCx</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>SAR</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SMS</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SNS</td>
<td>K</td>
<td>P</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>STCx</td>
<td>B</td>
<td>P</td>
<td>P</td>
<td>U(4) P</td>
</tr>
<tr>
<td>(1)TCM</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>R</td>
</tr>
<tr>
<td>TDC</td>
<td>K</td>
<td>R</td>
<td>N</td>
<td>P</td>
</tr>
</tbody>
</table>
IBM 1301 Disk and 7320 Drum File Orders

The following disk and drum file orders are recognized by MAP. The symbolic order should be written:

Location DORD access and module, track, record and assembles as ten BCD digits in two successive locations.

The access/module and track subfields may be any symbolic expression. They are evaluated in the normal manner and converted to BCD. The low-order two characters of the access/module subfield and the low-order four characters of the track subfield are inserted into the instruction. The record subfield is taken as alphanumeric data, and the first two characters are used. A DNOP (1212a) order is inserted in the last two character positions of the second word.

IBM 7340 Hypertape Orders

The following Hypertape orders are recognized by MAP. The symbolic order should be written:

Location HORD Tape unit (if required)

and assembles as two (or three for HSBR and HSEL) BCD characters, left-justified in the word. Locations containing two-character orders are filled with trailing HNOP (1212a) codes. Three-character orders are repeated in the rightmost three characters.

Appendix B: 7090 Macro-Expansions of 7094 Instructions

When assembling in the 7090 mode, certain 7094 instructions are replaced by equivalent 7090 macro-instructions. The expansions of these macro-instructions are provided in this appendix.

The expansions are divided into groups in which only a few instructions vary from the given operation code. The generic macro-expansion of each group is given with the necessary substitutions.

In this appendix, the symbols E.1, E.2, E.3, and E.4 are generated by the expansions as names of temporary storage locations. However, these symbols must be defined by the programmer or they will be virtual. The symbol is generated and defined by the expansion even if the mode of the created symbol switch is NOCRS.

Relative addressing must be used carefully when 7090 macro-expansions of 7094 instructions are used. The reason for this, of course, is that the single macro-instruction appearing in the source program will be replaced by a group of 7090 instructions during assembly.

For example, at first glance the following sequence of instructions appears to be a loop that will form five, double-precision, floating-point products, where

1. the first factor of each product is composed of the contents of location ALPHA+i and BETA+i (i = 0, 1, 2, 3, 4),
2. the second factor is composed of the contents of locations MULT and MULT+1,
3. each product replaces its first factor in storage, and
4. multiplication does not occur if the contents of ALPHA+i are negative.
However, since the macro-instruction DFMP will be replaced by a macro-expansion consisting of 12 instructions (see below), the TMI and TXI instructions in this sequence would both cause transfers to some "unknown" instruction in the macro-expansion.

Similarly, the sequence

\[
\begin{align*}
\text{SCD} & \quad \ast 1,2 \\
\text{TXL} & \quad \text{NAME},4,**
\end{align*}
\]

would result in an error since the \(\ast 1\) indicates an instruction in the macro-expansion of the SCD instead of the TXL instruction as intended.

When assembling in the 7094 or 7094 II mode, the macro-operations listed in this section are automatically purged.

**Group 1. PCA and PCD**

For PCA, take \(w = A\).

For PCD, take \(w = D\).

The expansion is then

\[
\begin{align*}
\text{PCw} & \quad \text{Y,T} \\
\text{PwC} & \quad \text{Y,T} \\
\text{PwC} & \quad \text{Y,T}
\end{align*}
\]

**Group 2. SCA and SCD**

For SCA, take \(w = A\).

For SCD, take \(w = D\).

The expansion is then

\[
\begin{align*}
\text{SCw} & \quad \text{Y,T} \\
\text{SXA} & \quad \ast +1,T \\
\text{AXC} & \quad \ast +2 \\
\text{SXw} & \quad \text{Y,T} \\
\text{LX} & \quad \ast -2,T
\end{align*}
\]

**Group 3. DFAD, DFSB, DFAM, and DESM**

For DFAD, take \(op = AD\).

For DFSB, take \(op = SB\).

For DFAM, take \(op = AM\).

For DESM, take \(op = SM\).

In this and the following groups, the expansion below is also used:

\[
\begin{align*}
\text{(SAVE) & \quad \text{Y,T}}
\end{align*}
\]

There are four forms for these expansions:

\[
\begin{align*}
\text{DFop} & \quad \ast +1,T \\
\text{OP} & \quad \text{Y,T} \\
\text{STO} & \quad \text{E.1} \\
\text{CLA} & \quad \ast +2 \\
\text{STA} & \quad \ast +2 \\
\text{STT} & \quad \ast +1 \\
\text{PXA} & \quad \ast -1 \\
\text{SUB} & \quad \ast -1 \\
\text{SX} & \quad \text{CS,4} \\
\text{PAC} & \quad 0.4 \\
\text{CLA} & \quad \text{E.1}
\end{align*}
\]

**Group 4. DLD and DST**

For DLD, take \(op = \text{LD, opa = CLA, and opb = LDXQ}\).

For DST, take \(op = \text{ST, opa = STO, and opb = STQ}\).

There are four forms for these expansions:

\[
\begin{align*}
\text{DFop} & \quad \ast +1,T \\
\text{OP} & \quad \text{Y,T} \\
\text{STO} & \quad \text{E.1} \\
\text{CLA} & \quad \ast +2 \\
\text{STA} & \quad \ast +2 \\
\text{STT} & \quad \ast +1 \\
\text{PXA} & \quad \ast -1 \\
\text{SUB} & \quad \ast -1 \\
\text{SX} & \quad \text{CS,4} \\
\text{PAC} & \quad 0.4 \\
\text{CLA} & \quad \text{E.1}
\end{align*}
\]

**Group 5. DFMP**

There are four forms for this expansion:

\[
\begin{align*}
\text{DFMP} & \quad \ast +1,T \\
\text{OP} & \quad \text{Y,T} \\
\text{STO} & \quad \text{E.1} \\
\text{CLA} & \quad \ast +2 \\
\text{STA} & \quad \ast +2 \\
\text{STT} & \quad \ast +1 \\
\text{PXA} & \quad \ast -1 \\
\text{SUB} & \quad \ast -1 \\
\text{SX} & \quad \text{CS,4} \\
\text{PAC} & \quad 0.4 \\
\text{CLA} & \quad \text{E.1}
\end{align*}
\]
Appendix C: Operation Code Formats

The operation code formats to be used with the orp and orpfd pseudo-operations are given in this appendix.

Operations

Entries are made in the Combined Operations Table for all orp and orpfd pseudo-operations, since the lookup process is the same as for any other symbol. To specify machine operations using orpfd, the general form of this entry is

\[ 6/A,5/0,1/IND,2/ADD,2/TAG,2/DEC,18/V \]

where each group of bits in the instruction word is specified by an octal number. The same general form is used for orp except that the 36-bit word is specified as a whole by a 12-digit octal number that will result in the same bit structure.

In this format, \( V \) varies with adjective code \( A \) (described below). The fields \( ADD, TAG, \) and \( DEC \) refer to address, tag, and decrement, respectively. The following code is used:

- 0: Field required
- 1: Field permissible
- 2: Field unexpected but allowed
- 3: Field not permissible

A 1 for \( IND \) indicates that indirect addressing is permitted, and a 0 indicates that it is not permitted.

The seven bits specifying fields \( IND, ADD, TAG, \) and \( DEC \) are denoted by \( FR \) (Fields Required).

Pseudo-Operations

The general form of the entry for pseudo-operations in the Combined Operations Table is

\[ 06/A,12/N,18/0 \]

where \( A \) is the adjective code and \( N \) defines a specific pseudo-operation under that adjective code.

The single exception is the macro call (adjective code 62). The format for this entry is

\[ 06/62,12/C,3/SC,15/L \]

where \( c \) is the number of parameters, \( sc \) is symbol creation information, and \( L \) is the location of the internal prototype. Adjective 62 is not allowed with orp and orpfd pseudo-operations.

Adjective Codes

The following list of octal adjective codes gives the type of operation to which each applies.

<table>
<thead>
<tr>
<th>Code (Octal)</th>
<th>Operation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>A</td>
</tr>
<tr>
<td>41</td>
<td>Prefix</td>
</tr>
<tr>
<td>42</td>
<td>Input/Output Command</td>
</tr>
<tr>
<td>43</td>
<td>B</td>
</tr>
<tr>
<td>44</td>
<td>C</td>
</tr>
<tr>
<td>45</td>
<td>D</td>
</tr>
</tbody>
</table>
CODE (OCTAL)  OPERATION TYPE
46  E
47  Select
50  Disk and Drum Channel Commands (4 fields)
51  Disk and Drum Channel Commands
52  Boolean Variable
53  Disk File and Drum File Orders
54  Unexpanded 7094 Instructions
55  Hypertape Orders
56  Special Internal Code
57  Special Internal Code

* This adjective code is not usable with OPD and OPVFD pseudo operations.

TYPE A INSTRUCTIONS (40)
The entry is
O6/40,05/0,07/FR,06/0,012/OPCODE
where opcode is an octal machine code written with the prefix in the first digit. For example, opcode for the instruction TXL would be 1000.

PREFIX OPERATIONS (41)
The entry is
O6/41,05/0,07/FR,06/0,012/OPCODE
where opcode is written with the prefix in the first digit. For example, opcode for the instruction PON would be 1000.

INPUT/OUTPUT COMMANDS (42)
The entry is
O6/42,05/0,07/FR,03/0,N,03/0,012/OPCODE
where opcode is written with the prefix in the first digit. For example, opcode for the instruction TCN would be 1000. If the instruction is a nontransmitting command, N = 2.

TYPE B INSTRUCTIONS (43)
The entry is
O6/43,05/0,07/FR,06/0,012/OPCODE

TYPE C INSTRUCTIONS (44)
The entry is
O6/44,05/0,07/FR,06/0,012/OPCODE

TYPE D INSTRUCTIONS (45)
The entry is
O6/45,05/0,07/FR,06/0,012/OPCODE

TYPE E INSTRUCTIONS (46)
The usual format is
O6/46,05/0,07/FR,03/S,015/E
where S is the sign of the operation:
If S = 0, opcode would be +0760.
If S = 1, opcode would be −0760.
EA is the actual extended address of the instruction.

SELECT INSTRUCTIONS (47)
The entry is
O6/47,05/0,07/FR,03/0,03/E,06/C,06/OPCODE
opcode is the Select type, as follows:
0 = Read
1 = Write
2 = Set Density High
3 = Set Density Low
4 = Rewind
5 = Rewind Unload
6 = Backspace Record
7 = Backspace File
10 = Write End of File
C is the channel number of Select, starting at 1 for Channel A.
E is the equipment code and has the following significance:
0 = Decimal Tape
1 = Binary Tape; or either Binary or Decimal Tape
2 = Card Reader
3 = Punch
4 = Decimal Printer
5 = Binary Printer
See “Select Type” operations for specific address field requirements.

DISK AND DRUM CHANNEL COMMANDS (50)
The entry is
O6/50,05/0,07/FR,06/0,012/OPCODE

DISK AND DRUM CHANNEL COMMANDS (51)
The entry is
O6/51,05/0,07/FR,03/0,N,03/0,012/OPCODE
where N = 2 if a 1 in bit position 19 is part of the operation code.

BOOLEAN VARIABLE (52)
The entry is
O6/52,05/0,07/FR,06/0,012/OPCODE
where the high-order bit of opcode is always ON.

DISK FILE AND DRUM FILE ORDERS (53)
The format is
O6/53,06/0,02/ACC,02/TRK,02/REC.
O6/0,012/OPCODE
ORCODE is the order, written in external BCD notation. For example, DRIL with an order code of 04 would be written 1204.

The fields ACC, TRK, and REC refer to the access module, track, and record, respectively. Encoding is:

1 Field permissible

UNEXPANDED 7094 INSTRUCTIONS (54)

The entry is

06/54,05/07/FB,06/0,012/OPCODE

HYPERTAPE ORDERS (55)

The entry is

06/55,06/0,02/ADD,010/0,012/OPCODE

The following is a list of pseudo-operations corresponding to specific A's and N's.

<table>
<thead>
<tr>
<th>A</th>
<th>N</th>
<th>PSEUDO-OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>0</td>
<td>MACRO</td>
</tr>
<tr>
<td>61</td>
<td>0</td>
<td>IFT</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>IFF</td>
</tr>
<tr>
<td>61</td>
<td>2</td>
<td>IRP</td>
</tr>
<tr>
<td>61</td>
<td>3</td>
<td>SET</td>
</tr>
<tr>
<td>61</td>
<td>4</td>
<td>ORGCRS</td>
</tr>
<tr>
<td>61</td>
<td>5</td>
<td>NOCRS</td>
</tr>
<tr>
<td>61</td>
<td>6</td>
<td>ENDM</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
<td>USE</td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>BEGIN</td>
</tr>
<tr>
<td>62</td>
<td>2</td>
<td>ORG</td>
</tr>
<tr>
<td>62</td>
<td>3</td>
<td>LOC</td>
</tr>
<tr>
<td>62</td>
<td>4</td>
<td>BSS</td>
</tr>
<tr>
<td>62</td>
<td>5</td>
<td>BES</td>
</tr>
<tr>
<td>62</td>
<td>6</td>
<td>COMMON</td>
</tr>
<tr>
<td>62</td>
<td>7</td>
<td>EVEN</td>
</tr>
<tr>
<td>62</td>
<td>8</td>
<td>LORG</td>
</tr>
<tr>
<td>62</td>
<td>9</td>
<td>LDIR</td>
</tr>
<tr>
<td>62</td>
<td>10</td>
<td>LITORG</td>
</tr>
<tr>
<td>62</td>
<td>11</td>
<td>EQU (and SYN)</td>
</tr>
<tr>
<td>62</td>
<td>12</td>
<td>MAX</td>
</tr>
<tr>
<td>63</td>
<td>3</td>
<td>MIN</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>BOOL</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>LBOOL</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>RBOOL</td>
</tr>
<tr>
<td>65</td>
<td>0</td>
<td>OPD</td>
</tr>
<tr>
<td>65</td>
<td>1</td>
<td>OPVFD</td>
</tr>
<tr>
<td>65</td>
<td>2</td>
<td>OPSYN</td>
</tr>
<tr>
<td>67</td>
<td>0</td>
<td>DUP</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>GOTO</td>
</tr>
<tr>
<td>71</td>
<td>0</td>
<td>OCT</td>
</tr>
<tr>
<td>71</td>
<td>1</td>
<td>DEC</td>
</tr>
<tr>
<td>71</td>
<td>2</td>
<td>BCI</td>
</tr>
<tr>
<td>71</td>
<td>3</td>
<td>LIT</td>
</tr>
<tr>
<td>71</td>
<td>4</td>
<td>VFD</td>
</tr>
<tr>
<td>72</td>
<td>0</td>
<td>QUAL</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>ENDOQ</td>
</tr>
<tr>
<td>72</td>
<td>2</td>
<td>CONTRO</td>
</tr>
<tr>
<td>72</td>
<td>3</td>
<td>FILE</td>
</tr>
<tr>
<td>72</td>
<td>4</td>
<td>ENTRY</td>
</tr>
<tr>
<td>72</td>
<td>5</td>
<td>LABEL</td>
</tr>
<tr>
<td>73</td>
<td>0</td>
<td>ABS</td>
</tr>
<tr>
<td>73</td>
<td>1</td>
<td>FUL</td>
</tr>
<tr>
<td>73</td>
<td>2</td>
<td>TCD</td>
</tr>
<tr>
<td>73</td>
<td>3</td>
<td>PUNCH</td>
</tr>
<tr>
<td>73</td>
<td>4</td>
<td>UNPUNCH</td>
</tr>
<tr>
<td>74</td>
<td>0</td>
<td>ETC</td>
</tr>
<tr>
<td>74</td>
<td>1</td>
<td>REM</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
<td>NULL</td>
</tr>
<tr>
<td>74</td>
<td>3</td>
<td>END</td>
</tr>
<tr>
<td>74</td>
<td>4</td>
<td>PURGE</td>
</tr>
<tr>
<td>74</td>
<td>5</td>
<td>CALL</td>
</tr>
<tr>
<td>74</td>
<td>6</td>
<td>SAVE</td>
</tr>
<tr>
<td>74</td>
<td>7</td>
<td>SAVEN</td>
</tr>
<tr>
<td>74</td>
<td>8</td>
<td>UNLIST</td>
</tr>
<tr>
<td>74</td>
<td>9</td>
<td>LIST</td>
</tr>
<tr>
<td>74</td>
<td>10</td>
<td>TITLE</td>
</tr>
<tr>
<td>74</td>
<td>11</td>
<td>DETAIL</td>
</tr>
<tr>
<td>74</td>
<td>12</td>
<td>EJECT</td>
</tr>
<tr>
<td>74</td>
<td>13</td>
<td>SPACE</td>
</tr>
<tr>
<td>74</td>
<td>14</td>
<td>LBL</td>
</tr>
<tr>
<td>74</td>
<td>15</td>
<td>PCC</td>
</tr>
<tr>
<td>74</td>
<td>16</td>
<td>INDEX</td>
</tr>
<tr>
<td>74</td>
<td>17</td>
<td>PMC</td>
</tr>
<tr>
<td>74</td>
<td>18</td>
<td>TTL</td>
</tr>
<tr>
<td>74</td>
<td>19</td>
<td>PCG</td>
</tr>
<tr>
<td>74</td>
<td>20</td>
<td>KEEP</td>
</tr>
</tbody>
</table>

Appendix D: IBMAP-FAP Incompatibilities

This appendix lists the incompatibilities that will occur when assembling a FAP or IBFAP program using IBMAP.

1. All FAP machine operations and extended machine operations will assemble properly in MAP.

2. All FAP pseudo-operations pertaining to the update facility have no counterpart in MAP. They are:

   DELETE   NUMBER   UMC  
   ENDFIL   REWIND   UNLOAD  
   ENDUP    SKIPTO   UPDATE  
   IGNORE   SKPFL

3. The following FAP pseudo-operations have no counterpart in MAP. (Since MAP treats undefined operations as remarks, some of these operations do not affect assembly.)

   IFEOF    TAPENO  
   PRINT    704    
   SST      9LP    

4. The following FAP pseudo-operations must be replaced by the indicated MAP equivalent:

   FAP        MAP
   HEAD,HEX   QUAL,ENDQ
   BCD        BCI
   MOP        MACRO
   MAC        Standard macro-instruction
   END (of macro) ENDM
   RMT        USE and USE PREVIOUS

(Note that QUAL permits nesting of qualifiers, whereas HEAD does not.)

The following is an example of a program segment coded first in FAP and then in MAP.

<table>
<thead>
<tr>
<th>FAP</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMT</td>
<td>BSS</td>
</tr>
<tr>
<td>RMT</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ALPHA</td>
<td>RMT</td>
</tr>
<tr>
<td></td>
<td>EXTERN</td>
</tr>
</tbody>
</table>
SKP
SFC
TSX $SUB,4

MAP

USE RMT
BSS 10
USE PREVIOUS
.
.
BEGIN RMT, ALPHA
Leave symbols
undefined
EJECT
SPACE
TSX $SUB,4 (remove all $ signs that specify external names)

LOC
In MAP, the expression
LOC blank
is meaningless. The effect of a LOC is terminated with
the ORG pseudo-operation.

NULL
In MAP, a symbol in the name field of the card will be
given the value of the current location counter.

CALL
In MAP, the arguments must be enclosed in parentheses.
Also, a different calling sequence is generated.

SET
In MAP, the S-value of symbols in the variable field is
used.

6. The following FAP pseudo-operations have their
equivalent option specified on the IBM MAP card:

- REF
- COUNT
- SST

7. In MAP, the NULL pseudo-operation rather than
equ should be used for symbol definition, because of
the limited size of the Pseudo-Operation Dictionary.

8. The following FAP pseudo-operations will as-
semble properly in MAP:

- ABS FULL ORGCRS
- BCI INDEX PCC
- BES LBL PMC
- BOOL LIST REM
- BSS MAX SPACE
- DEC MIN TCD
- DETAIL NOCRS TITLE
- EJECT OCT UNLIST
- ENTRY OPSYN TTL
- ETC ORG VFD

9. In MAP, virtual entries in the Control Dictionary
correspond to the transfer vector of FAP except that
IBM provides direct rather than indirect references.

10. In MAP, normal arithmetic truncates to 15 bits
in the address field, 3 in the tag, and as specified in
the decrement. VFD symbolic arithmetic truncates to
a maximum of 20 bits. Boolean arithmetic truncates to
18 bits.

11. Literals are placed in the literal pool in the
order in which they appear in a program; they are not
arranged numerically.
### Appendix E: The MAP BCD Character Code

The MAP BCD character code is shown in octal form in the following table with the corresponding IBM punched card code.

<table>
<thead>
<tr>
<th>CHARACTER</th>
<th>BCD CODE (OCTAL)</th>
<th>CARD CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>60</td>
<td>(blank)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>21</td>
<td>12-1</td>
</tr>
<tr>
<td>B</td>
<td>22</td>
<td>12-2</td>
</tr>
<tr>
<td>C</td>
<td>23</td>
<td>12-3</td>
</tr>
<tr>
<td>D</td>
<td>24</td>
<td>12-4</td>
</tr>
<tr>
<td>E</td>
<td>25</td>
<td>12-5</td>
</tr>
<tr>
<td>F</td>
<td>26</td>
<td>12-6</td>
</tr>
<tr>
<td>G</td>
<td>27</td>
<td>12-7</td>
</tr>
<tr>
<td>H</td>
<td>30</td>
<td>12-8</td>
</tr>
<tr>
<td>I</td>
<td>31</td>
<td>12-9</td>
</tr>
<tr>
<td>J</td>
<td>41</td>
<td>11-1</td>
</tr>
<tr>
<td>K</td>
<td>42</td>
<td>11-2</td>
</tr>
<tr>
<td>L</td>
<td>43</td>
<td>11-3</td>
</tr>
<tr>
<td>M</td>
<td>44</td>
<td>11-4</td>
</tr>
<tr>
<td>N</td>
<td>45</td>
<td>11-5</td>
</tr>
<tr>
<td>O</td>
<td>46</td>
<td>11-6</td>
</tr>
<tr>
<td>P</td>
<td>47</td>
<td>11-7</td>
</tr>
<tr>
<td>Q</td>
<td>50</td>
<td>11-8</td>
</tr>
<tr>
<td>R</td>
<td>51</td>
<td>11-9</td>
</tr>
<tr>
<td>S</td>
<td>62</td>
<td>0-2</td>
</tr>
<tr>
<td>T</td>
<td>63</td>
<td>0-3</td>
</tr>
<tr>
<td>U</td>
<td>64</td>
<td>0-4</td>
</tr>
<tr>
<td>V</td>
<td>65</td>
<td>0-5</td>
</tr>
<tr>
<td>W</td>
<td>66</td>
<td>0-6</td>
</tr>
<tr>
<td>X</td>
<td>67</td>
<td>0-7</td>
</tr>
<tr>
<td>Y</td>
<td>70</td>
<td>0-8</td>
</tr>
<tr>
<td>Z</td>
<td>71</td>
<td>0-9</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>(plus)</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>(minus)</td>
</tr>
<tr>
<td>/</td>
<td></td>
<td>(slash)</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td>(equals)</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>(apostrophe)</td>
</tr>
<tr>
<td>.</td>
<td></td>
<td>(period)</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td>(right parenthesis)</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td>(dollar sign)</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>(asterisk)</td>
</tr>
<tr>
<td>,</td>
<td></td>
<td>(comma)</td>
</tr>
<tr>
<td>(</td>
<td></td>
<td>(left parenthesis)</td>
</tr>
</tbody>
</table>
file check-sum 30
file close 30
file conversion 31
file density 30
file label density 30
file mode 30
file mounting 29
file nonstandard-label routine 31
file reel handling 30
file unit assignment 26
file usage 29
Hypertape 31
Hypertape reel switching 31
operator file list 29
opsvro pseudo-operation 32, 54
Orders
disk and drum file 52
Hypertape 52
Ordinary symbols 8
orng pseudo-operation 15
orngcrs pseudo-operation 48
Ord.
absolute 9, 15
relocatable 9
Parameters
delimiting in macro-instructions 44
in macro-instructions 44
pcc pseudo-operation 35
pcc pseudo-operation 37
pmc pseudo-operation 36
Prefix codes 51
Prefix operation code format 54
Primary file unit 28
Principal part
fixed-point number 10
floating-point number 9
Prototype card images 42
Prototype instructions 42, 43
substitutable arguments in

text in 45
Pseudo-operation 5
Pseudo-operation code formats 54, 55
Pseudo-operations
ABS 34
absolute-assembly 6, 34
BCL 10
BEGIN 14
BES 16
BOOL 24
Booleans 5, 24
BSS 15
CALL 17
COMMON 24
conditional-assembly 5, 24
CONTROL 27
control-section 5, 26
data-generating 5, 18
dec 18
DETAIL 30
dup 35
EJECT 21
END 35
ENDM 41, 43
ENDER 26
ENTRY 27
EQU 25
ETC 33
EVEN 17
FILE 28
file-description 5, 29
FUL 30
GOTO 25
IFF 24, 45
IFT 24, 45
INDEX 36
INT 47
KEEP 16, 22, 34
LABEL 31
LBD 36
LBOOL 24
LDR 17
LST 35
list-control 6, 35
LIT 20
LITORG 18
LOC 15
location-counter 5, 14
LONG 18
MACRO 41
macro-defining 6, 41
macro-related 6, 46
MAX 23
MIN 23
miscellaneous 6, 32
NOCRS 48
NULL 22
OCT 18
OPD 32, 54
operation-defining 5, 32
OPSYN 32
OPVFD 32, 54
OPR 15
ORCRS 25
PCC 35
PCG 37
PMC 36
PUNCH 34
QUAL 25
RBOOL 24
REM 33
RETURN 39
SAVE 38
SAVEN 39
SETR 9, 22, 23
SPACE 35
special systems 6, 37
storage-allocation 5, 15
symbol-defining 5, 22
symbol-qualifying 5, 25
SYN 22
TCD 34
TITLE 35
TTL 37
UNLIST 35
UNPUNCH 34
USE 14
PUNCH pseudo-operation 24
PURGE pseudo-operation 48, 53
QUAL pseudo-operation 25
Qualification
in macro-operations 46
nested 26
Qualified section 26
Qualified symbol 26
rel pseudo-operation 24
Relational operator 24
Relative addressing 12
Relocatable assemblies 6
Relocatable expressions 11
Relocatable origin 9
Relocatable segments 26
Relocation properties of symbols 9
REM pseudo-operation 33

Index 61