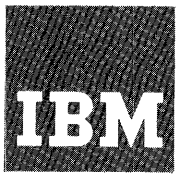
File No. 1410/7010-22
Form C28-0326-2



Systems Reference Library

IBM 1410/7010 Operating System (1410-PR-155)

Autocoder – 1410-AU-968

This publication is a reference text for personnel engaged in writing programs in the Autocoder language for use within the framework of the 1410/7010 Operating System. The Autocoder language is composed primarily of symbolic one-for-one source statements. Its associated processor (Program Number 1410-AU-968) is a symbolic assembly program with macro-generation facilities.

MINOR REVISION (November 1964)
This publication is a minor revision of, and makes obsolete, the publication *IBM 1410/7010 Operating System; Autocoder*, Form C28-0326-1, with its associated Technical Newsletter, N28-1126. Minor changes to the text have been indicated by a vertical line to the left of the change; revised illustrations are denoted by the symbol • to the left of their figure captions.

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, Department 637, Neighborhood Road, Kingston, New York 12401

Contents

Purpose of this Publication

This publication is a reference text for personnel writing programs in the Autocoder language for use with the IBM 1410/7010 Operating System.

Purpose of the Language and Processor

The 1410/7010 Autocoder is one of three programming systems provided by the Operating System. (The other two are FORTRAN and COBOL.) Autocoder consists of a symbolic coding language and an assembly program called a processor.

Utilizing the required elements provided by the Autocoder language, a program can be created by coding the steps necessary for the solution of the problem, and presenting them in the form of statements. The Autocoder processor translates these statements into the computer's internal language, and assembles them in the form of a relocatable subprogram. The resulting relocatable subprogram can then be processed by the Linkage Loader and used whenever necessary.

The Autocoder Language: Elements of Source Statements

The user's source program is written by using mnemonic symbols to represent the principal elements of the source statements. These elements are (1) the name of the statement (a *label*), (2) the operation to be performed by the statement (an *operation code*), and (3) the program elements or parameters on which the operation is to be based and/or performed (the *operand*).

Macro statements in a user's source program cause additional symbolic source statements to be generated and inserted into the program. The generated source statements will be tailored according to the logic of the macro routine the programmer has placed in the Macro Library and the parameters in the source macro-instructions.

The Autocoder Processor: Used with the Operating System

The 1410/7010 Autocoder processor is designed to operate in conjunction with the Operating System. The object programs produced are assembled and run according to the conventions of the Operating System.

THE ASSEMBLY PROCESS

The assembly process produces an assembly listing and an object program in card-image form called, in this publication, an *object deck*. The object deck is in relocatable format, ready for processing by the Linkage Loader.

The programs assembled by the 1410/7010 Autocoder processor must be designed to run exclusively within the framework of the Operating System. The Resident Monitor, including the Resident IOCS, provides a great deal of power by simplifying the task of programming. A minimum of programming need be concerned with operating control and input/output operations, since macro-instructions are available for these functions.

THE OBJECT PROGRAM

The following points should be noted concerning the object program as it is discussed in this publication, and executed within the framework of the Operating System:

1. The *object program* will be referred to as a *subprogram*. (A subprogram is the basic program unit with which the Linkage Loader performs its processing.) This subprogram can be a self-contained program, or it can be a subroutine to be executed in conjunction with other subprograms, forming a multiphase program.

2. Subprograms to be combined during one run of the Linkage Loader can be assembled individually at different times or during a single assembly run.

Prerequisites

It is assumed that the user has completed a basic course in programming for the IBM 1410 or IBM 7010 Data Processing System, and is familiar with information contained in the following publications:

IBM 1410 Principles of Operation, Form A22-0526, or
IBM 7010 Principles of Operation, Form A22-6726, and
IBM 1410/7010 Operating System; Basic Concepts, Form C28-0318
IBM 1410/7010 Operating System; System Monitor, Form C28-0319

The Autocoder user must know certain Operating System conventions and requirements in order to write,

assemble, and execute his programs properly. The pertinent information is contained in the following publications:

IBM 1410/7010 Operating System; System Monitor, Form C28-0319.

IBM 1410/7010 Operating System; Basic Input/Output Control System, Form C28-0322.

Operating instructions for the System are contained in the publication, *IBM 1410/7010 Operating System; Operator's Guide,* Form C28-0351.

This manual will indicate specific cross references to the above-listed publications at the appropriate places. The user will find that the many advantages and conveniences of the Operating System can be implemented through the use of Autocoder.

Minimum Machine Requirements

The minimum machine requirements for assembling source programs written in the Autocoder language are specified in the publication, *IBM 1410/7010 Operating System; System Generation,* Form C28-0352.

Machine requirements for the execution of the object program depend upon the combined requisites of the System Monitor and the user's program.

This section describes Autocoder statements and the principal elements used in their construction; programming concepts under the Operating System; the assembly listing produced by the processor; and the coding sheet upon which source statements are coded. Succeeding sections discuss the various types of permissible operands, Autocoder operation codes, and the Macro System.

Autocoder Statements

The source program, which is translated by the Autocoder processor into an object program, is composed of five types of Autocoder statements:

- Imperative (symbolic machine instructions)
- Declarative
- Linkage Loader
- Control
  - Assembly Control Statements
  - Subprogram Control Statements
- Macro

Imperative Statements are translated into the machine instructions that appear in the object program.

Declarative Statements are translated into data areas, data constants, and address constants used by the object program. They are also used to define symbols (or labels) in the assembly process.

Linkage Loader Statements enable the Linkage Loader to properly convert the relocatable subprograms assembled by the Autocoder processor into absolute format (ready for execution).

Control Statements are directions to the Autocoder processor, which control the performance of specified operations at assembly time.

Macro-instructions enable the programmer to extract, from a library of macro routines, instruction sequences tailored (by means of parameters written in macro-instructions) to meet programmer specifications. These instruction sequences are inserted automatically into the object program. (See the section entitled "The Macro System.")

Principal Elements of Autocoder Statements

The principal elements of an Autocoder statement are: a *label,* an *operation code,* and an *operand.*

Label

A label is a name assigned by the programmer to an element in a program (e.g., a data area, constant, or instruction). This enables operands of Autocoder statements, referencing labeled elements, to have symbolic form. The terms label and symbol will be used synonymously throughout this publication.

A label can contain from one to ten alphameric characters, which are left-justified in the label field (card columns 6-15). The first character must be alphabetic. A special case in which labels are not left-justified is explained in the sections concerning DS, DC and DCW statements.

Special characters must not be used in the label field. (An exception is permitted in the DEFIN statement. See "DEFIN — Definition," under "Linkage Loader Operation Codes.")

Operation Code

The operation code field (card columns 16-20) contains a one-to-five-character mnemonic that specifies the nature of the Autocoder statement and indicates to the processor the function to be performed during the assembly process. A table of operation codes is supplied in the section, "Autocoder Operation Codes."

Operation codes in machine language must never be used. (The Compare, Add, and Subtract imperative operations are exceptions, in that the machine-language equivalent for each is identical to the corresponding mnemonic.)

Blank operation codes are permitted in conjunction with the following mnemonics: DC, DCW, DA, and DAV. These mnemonics are discussed in the subsection entitled "Declarative Operation Codes."

The mnemonic operation codes are listed in Appendix D. In studying this list, the programmer will note that groups of mnemonic operation codes (for example, the group of mnemonics for the scan instructions) are represented in machine language by a one-character operation code and an operation modifier (d-character) which defines the precise function the operation is to perform. This is true even though the mnemonic may be as many as five positions long.

Operand

The operand field begins in card column 21. The form and content of the various permissible operand elements vary according to the operation to be performed.

However, the basic elements in the operand field are the A- and B-addresses, and the d-character (when required). An A-address, a B-address, and a d-character are separated from each other by single commas (Figure 1).

The formats of these and the other permissible operand elements are discussed in the section "Types of Operand Entries." The use of these elements in association with specific operation codes is discussed in the sections "Autocoder Operation Codes," and "The Macro System."

Relocation

All object programs produced by the compilers within the Operating System are in relocatable format. The aspects of relocation with which the Autocoder user must be familiar are noted here. The reader interested in a detailed discussion of this subject and the relocation indicators that can appear in the assembly listing should refer to the publication, *IBM 1410/7010 Operating System; System Monitor,* Form C28-0319.

Relocation is achieved in three steps:

1. The programmer codes subprograms with or without regard to their actual location in core storage. He can, however, specify a starting location for his program.

2. The processor assigns relative addresses to the program statements and constants, starting at the address specified by the programmer, or zero (00000) if not specified. The processor also indicates to the Linkage Loader whether these addresses are to be given *upward, downward,* or *NO* relocation, as explained below.

3. When the program is loaded, the addresses are adjusted by a relocation factor calculated and applied by the Linkage Loader. (The adjusted addresses maintain the same relative locations and relationships to each other as specified in the source program.) This subject is covered in more detail under "ORG — Origin," in the subsection "Subprogram Control Statements."

Each address within the program is assigned a code by the processor, indicating to the Linkage Loader one of three types of relocation. When the Linkage Loader is executed, it calculates and applies to the address the relocation factor called for by the associated indicator.

The three types of relocation factors that can be applied are:

Upward
Downward
NO

Upward Relocation

Upward relocation means that the address in the object program will normally be incremented. For example, if the *compiled* origin of the object program is 00000 and the relocation factor is 12,000, the program will be loaded starting at core-storage location 12000. The relocation factor is added to each load address and to each address within the program to which an upward relocation indicator is assigned by the processor.

Of the three relocation types possible, upward relocation is the one most frequently applied.

Downward Relocation

Downward relocation means that the address in the object program will be decremented by a value calculated by the Linkage Loader. A downward relocation indicator is assigned by the processor to those addresses that refer to the COMMON data area.

NO Relocation

NO relocation means that the address in the object program is to be unchanged by the Linkage Loader. A NO relocation indicator will be assigned by the processor to those addresses whose absolute value is already supplied and must be maintained. For example, the addresses of index registers receive NO relocation indicators.

Definition of COMMON

COMMON is the formal name, predefined in the processor's symbol table, of a relocatable work area that can be referenced by more than one subprogram. Certain language conventions must be observed if separately-assembled subprograms are to achieve compatible addressing of shared data fields in COMMON.

During the assembly process, references to COMMON are translated according to an addressing and relocation convention designed to make these references suitable for resolution by the Linkage Loader. The convention chosen is the assignment of the value 99999

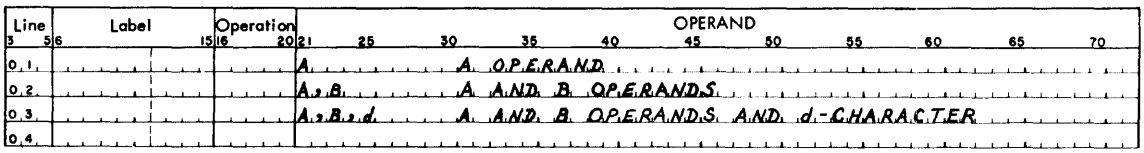


Figure 1. The Basic Operand Entries

as the reference address of COMMON during the assembly process. All relocatable addresses of data in COMMON are relative to 99999. For example, the 15th location downward in COMMON is assigned the value 99985, and appears as the same relative address in all subprograms. Labels referencing COMMON are assigned downward relocation indicators for absolute adjustment by the Linkage Loader.

Absolute adjustment involves changing the relative values of the labels (assigned to them by the processor) to absolute values in the relocated COMMON data area. The adjustment factor applied is the difference between the value of COMMON in the assembly process (99999) and the absolute value of COMMON determined by the Linkage Loader. Normally, the Linkage Loader will place COMMON at the location represented by the value of the system symbol /AMS/ (Absolute Memory Size). However, the programmer can specify a different absolute location for COMMON by means of a BASE2 statement. (The interested reader will find a fuller discussion of this subject in the publication, *System Monitor.*)

The steps necessary to use COMMON in a subprogram are discussed under "DAV – Define Area in COMMON," in the subsection "Declarative Operation Codes."

Processing Options

There are four processing options which can be exercised by the user:

1. He can suppress the printing of the assembly listing (on the Standard Print Unit).
2. He can suppress the punching of the object deck (on the Standard Punch Unit).
3. If there are no macro statements in the source deck, he can speed up the assembly process by indicating this fact.
4. He can suppress the diagnostic generation of an "M" flag for uses of index registers 14 and 15 when there is no true multiple definition. (See NOTE 1, under "Indexing with Address Adjustment.")

These options are indicated by means of additional parameters in the EXEQ card that calls the Autocoder processor.

The four parameters are:

NOPRT – Suppress printing
NOPCH – Suppress punching
NOMAC – No macros present
NOFLG – Suppress "M" flag

Any or all of these parameters may be used in the EXEQ card. They can appear in any order immediately following the EXEQ parameters required by the System Monitor. (See the publication, *System Monitor,* for details concerning the EXEQ card.)

Specification of parameters in the EXEQ card is concluded by the first blank encountered in the operand field. The following examples illustrate the format:

LABEL	OPERATION CODE	OPERAND
MON$$	EXEQ	AUTOCODER, SOF, SIU, NOPRT
MON$$	EXEQ	AUTOCODER, , , NOMAC, NOPCH, NOFLG
MON$$	EXEQ	AUTOCODER, , , NOFLG, NOPRT, NOMAC, NOPCH

Autocoder Multiple Compilation

Autocoder can compile any number of programs with a single MON$$ EXEQ AUTOCODER card. The output is the same as if it were produced by several separate compilations.

Input for a multiple compilation consists of the MON$$ EXEQ AUTOCODER card followed by the source decks to be compiled. No control cards are necessary between the END statement of one program and the first card of the next program if the programmer wants the subsequent compilation to receive standard treatment; that is, printing, punching, and normal macro and flag processing.

A different set of processing options (NOPRT, NOPCH, NOMAC or NOFLG) can be specified for an ensuing program in a multiple compilation by placing an Option card after the preceding END statement. This card has the same requirements and options as the MON$$ EXEQ AUTOCODER card except that the label and operation fields, card columns 6-20, must contain blanks (instead of MON$$ EXEQ). The processing options specified in this Option card will be applied until the next Autocoder END card is read by the processor.

Autocoder multiple compilation has two potential advantages:

1. It enables the programmer to process a series of source decks from the Alternate Input Unit as well as the Standard Input Unit.
2. It bypasses the monitor processing which normally is necessary between compilations.

Terminating the Object Program

The object progam must terminate execution by means of one of the following instructions:

B	/EOP/	Normal End of Program
B	/UEP/	Unusual End of Program

Both forms of termination are shown in Figure 2. Full details can be found in the publication, *System Monitor.*

```
     64015                    SAMPLE SUBPROGRAM USING THE 1410/7010 AUTOCODER                PAGE    1              SAMPL
SEQNO  PGLIN  LABEL       OPCOD  OPERAND                                          REL   CT  ADDRS  INSTRUCTION      CARD  FLAG

   1   AA020              TITLE  SEQUENCE                                                                          001
   2   AA030  * THIS SUBPROGRAM CHECKS THE SEQUENCE OF THE PGLN/ FIELD
   3 S  A040  * IF THE PGLN/ FIELD IS 99999, THE PROGRAM IS TERMINATED NORMALLY
   4   AA050  * A NON-ASCENDING SEQUENCE RESULTS IN AN UNUSUAL END OF PROGRAM.
   5   AA060  SEQROUTINE  SBR    EXITSEQRT&5                                      ¤     7  00000  G 00056       B  002
   6   AA070              C      PGLN/,@99999@   IS THIS THE LAST ENTRY           1    11  00007  C PGLN/ 00153      002
   7   AA080              BE     ENDOFJOB        YES                              ¤     7  00018  J 00058       S  002
   8   AA090              NOPWM                                                   9     1  00025  N                   002
   9   AA100              B      CHECKSEQ                                         ¤     7  00026  J 00101           002
  10   AA110              SW     *-12            SET FIRST TIME NOP SWITCH TO BRANCH @     6  00033  , 00026      002
  11   AA120              MLCWB  PGLN/,PGLNHOLD#5                                 A    12  00039  D PGLN/ 00158 P  003
                                                                                  L
  12   AA130  EXITSEQRT   B      0               EXIT - RETURN TO MAIN PROGRAM    T     7  00051  J 00000           003
  13   AA135  *
  14   AA140  ENDOFJOB    IOCTL  TYPE,MESSAGE    NOTIFY OPERATOR OF END OF JOB
  15 G AA140  ENDOFJOB    EQU    *                                                                 00058
  16 G 01510              BZN    *-11,/CTB/                                       C    12  00058  V 00058 /CTB/ 2  003
                                                                                  L
  17 G 01520              BXPA   /CNC/                                            T     7  00070  Y /CNC/       X  003
  18 G 01530              DCW    MESSAGE                                          N     5  00081    00126           003
  19 G 01580              BZN    *-11,/CTB/                                       C    12  00082  V 00082 /CTB/ 2  004
                                                                                  L
  20   AA145              B      /EOP/           NORMAL END OF PROGRAM            T     7  00094  J /EOP/           004
  21   AA148  *
  22   AA150  CHECKSEQ    C      PGLN/,PGLNHOLD                                   1    11  00101  C PGLN/ 00158     004
  23   AA160              BH     EXITSEQRT-12    BRANCH IF PGLN/ IS IN SEQUENCE   ¤     7  00112  J 00039       U  004
                                                                                  L
  24   AA170              B      /UEP/           UNUSUAL END OF PROGRAM           T     7  00119  J /UEP/           004
  25   AA175  *


     64015                    SAMPLE SUBPROGRAM USING THE 1410/7010 AUTOCODER                PAGE    2              SAMPL
SEQNO  PGLIN  LABEL       OPCOD  OPERAND                                          REL   CT  ADDRS  INSTRUCTION      CARD  FLAG

  26   AA180  SEQR/       DEFIN  SEQROUTINE      SEQR/  LINKAGE SYMBOL FOR SUBPROGRAM                  00000           005
  27   AA185   MESSAGE    DCW    @END OF JOB@,G  CONSOL PRINTER NOTICE                 11  00126                   006
  28   AA190              HALT   12345           EXAMPLE OF AN ERRONEOUS STATEMENT A  12  00137  N 12345 .....    007  0
  29   AA200              END
  30                             @99999@                                            5  00153                   008
  31           PGLNHOLD          #0005                                              5  00158                   008
                                                  NUMBER OF FLAGGED STATEMENTS     1
   28

                                              1410/7010 AUTOCODER...SYSTEM /MID/ 0001
```

Figure 2. A Page from an Assembly Listing

Assembly Listing

Each page of the assembly listing contains a page heading line and a column heading line.

The page heading line contains the following information, from left to right:

1. The date contained at location /DAT/ (the system symbol for the five-position date field in the Resident Monitor)
2. Information supplied via HEADR card
3. Page number in the listing
4. The identification supplied by HEADR or RESEQ cards

The column heading line is illustrated in Figure 2, which shows the assembly listing of a subprogram assembled by the 1410/7010 Autocoder processor. The subprogram contains a deliberate error contrived to exhibit Autocoder's diagnostic flagging system. Figure 2 illustrates the following items, going from left to right in the column heading line:

1. SEQNO — Sequence Number: The sequence number of statements as they appear in the assembly listing.

2. PGLIN — Page and Line Number: The page and line number as it appears in columns 1 through 5 of the cards in the source deck. Page and line numbers must consist of five non-blank characters and must appear in ascending sequence.

Statements generated by the macro generator will have a page and line number in this field supplied by the generator. These numbers have no relationship to the numbers of the hand-coded statements; they represent the order in which the statements appear in the Macro Library.

The space between the SEQNO and PGLIN columns of the listing are used by the processor to contain either an "S" or a "G," under the following conditions.

S — The page and line number of the statement is not in ascending sequence in relation to the preceding source statement. This is only a warning to the programmer that his source statements may be out of sequence.

G — This character differentiates statements produced by the macro generator from the hand-coded source statements.

3. LABEL — Label: The contents of the label field, columns 6 through 15, of the Autocoder statement.

4. OPCOD: The Operation Code, columns 16 through 20, of the Autocoder statement.

5. OPERAND: The contents of the operand field, columns 21 through 72, of the Autocoder statement.

6. REL — Relocation Indicator: This is a code character that indicates to the Linkage Loader the type of relocation to be applied to the element(s) in the statement.

7. CT — Character Count: The length in characters of the assembled imperative statement, or the number of core-storage locations reserved for a constant defined in a declarative statement.

8. ADDRS: The relative address assigned by the processor to the instruction or constant. This address is subject to relocation.

9. INSTRUCTION: The assembled machine-language instruction or constants from which the object deck is constructed.

10. CARD — Card Number: The sequence number of the card in which the associated constants or instructions appear in the object deck. This sequence number is automatically computed and placed in columns 73-75 of each card in the object deck, in ascending order.

11. FLAG: An alphabetic character indicating an actual or possible programming error. As many as five flags can be assigned to one Autocoder statement. The flags provided are as follows:

F — invalid statement Format

M — Multiple definition of a label

N — macro generation Note

O — invalid Operation code

R — Restricted operation code (if not generated by a macro)

U — Unidentified label in the operand

W — Warning, general classification of error

Details concerning the above flags can be found in Appendix A. The total number of flagged statements is indicated at the end of the assembly listing, followed by a line which contains the sequence number of each flagged statement, to a maximum of 20 numbers. The presence of any flag except "R" causes the processor to set the "no-go" switch during assembly. This setting of the "no-go" switch can cause a bypassing of all the source cards up to the next job. See the *System Monitor* publication.

The assembly listing can be supplemented by a cross reference listing at the option of the user, by means of the PST statement. This listing analyzes the subprogram(s) just assembled, and lists each label, followed by the sequence number of the statement in which it was defined, and the sequence number of each statement in which the label is used as a reference address. See "PST — Print Symbol Table," in the subsection "Control Operation Codes," for a more detailed explanation.

NOTE: The system symbol /LIN/ controls the line count on the listing page. However, if this system symbol calls for the printing of less than 30 lines per page the processor will reject this direction and print the assembly listing at the normal 55 lines per page. See the *System Monitor* publication for details concerning this system symbol.

Replacement Codes

The Autocoder processor utilizes a second line (normally blank) in the assembly listing, for the representation of non-printable characters. Each of these characters is represented by two characters, one printed above the other, at the appropriate place in the listing. These two-character substitutions are called replacement codes, and they appear most frequently as relocation indicators or operation modifiers.

The two-character replacement codes with their conventional graphic representations, card codes, and names are listed in Figure 3.

Replacement Code	Graphic	Card Code	Name
$\overset{+}{0}$	?	12-0	Plus Zero
$\overset{-}{0}$	!	11-0	Minus Zero
GM	‡	12-7-8	Group Mark
QT	+++	0-7-8	Segment Mark
WS	⌒	0-5-8	Word Separator
DL	Δ	11-7-8	Delta
CT	¢ or ♿	2-8	Cent Sign or Substitute Blank
LP	[	12-5-8	Left Bracket
RP	]	11-5-8	Right Bracket
TM	√‾‾	7-8	Tape Mark
LT	<	12-6-8	Less Than
GT	>	6-8	Greater Than
;	;	11-6-8	Semicolon
:	:	5-8	Colon
" b	\	0-6-8	Backslash

Figure 3. Replacement Codes

Coding Sheet

The Autocoder Coding Sheet (Figure 4) provides a convenient form for coding source program statements. Column numbers on the coding sheet have a one-for-one correspondence to the columns on the card used to punch the source statements (*Autocoder Input Card,* Form A36199).

Each line of the coding sheet is punched into a separate card. The source deck, therefore, consists of a sequenced set of punched cards containing a line-by-line representation of the coding sheets.

The following paragraphs explain the function of each field. The heading information, *Program, Programmed By,* and *Date,* are only for documentation, and are not punched.

Identification (Card Columns 76-80)

This five-position field can contain a name created by the programmer to identify the program. This identification will be punched into 76-80 of the object deck only if it appears in a HEADR or RESEQ control card. (See "Control Operation Codes.") However, the identification is not checked on the other Autocoder statements, and serves only to identify the program to which the card belongs. Special, as well as alphameric, characters are permitted.

Page Number and Line Number (Card Columns 1-5)

The page number (columns 1 and 2), in conjunction with the line number (columns 3-5), provides a means of sequencing the cards in the source deck. This enables the programmer to identify and correlate the entries on the coding sheet and assembly listing with the entries in the source deck. Alphabetic, as well as numeric, characters can be used. (If the standard collating sequence is not followed, the processor will place a sequence (S) flag next to the PGLIN field in the assembly listing, as previously explained.)

Label (Card Columns 6-15)

This field, if used, contains the label being defined in this statement.

Operation Code (Card Columns 16-20)

This field contains the operation code.

Operand (Card Columns 21-72)

This field, if used, contains the operand element(s) of the statement.

NOTE: Columns 73-75 should be left blank.

COMMENTS

Comments are remarks or notes written by the programmer in the operand field. At least two blank spaces must separate a comment from the last character of the statement. The comment, punched in the source deck, appears in the assembly listing but is not contained in the object deck, and has no effect on the object program.

COMMENTS CARD

It may, at times, be helpful to insert an entire line of descriptive information. This is done by placing an asterisk in column 6 and using the balance of the line (up to column 72) for comments. When this line of information is punched into a card of the source deck, the asterisk will identify it to the processor as a comments card. The comments will be printed on a single line of the assembly listing at the point of encounter, which can be anywhere in the source deck. Comments

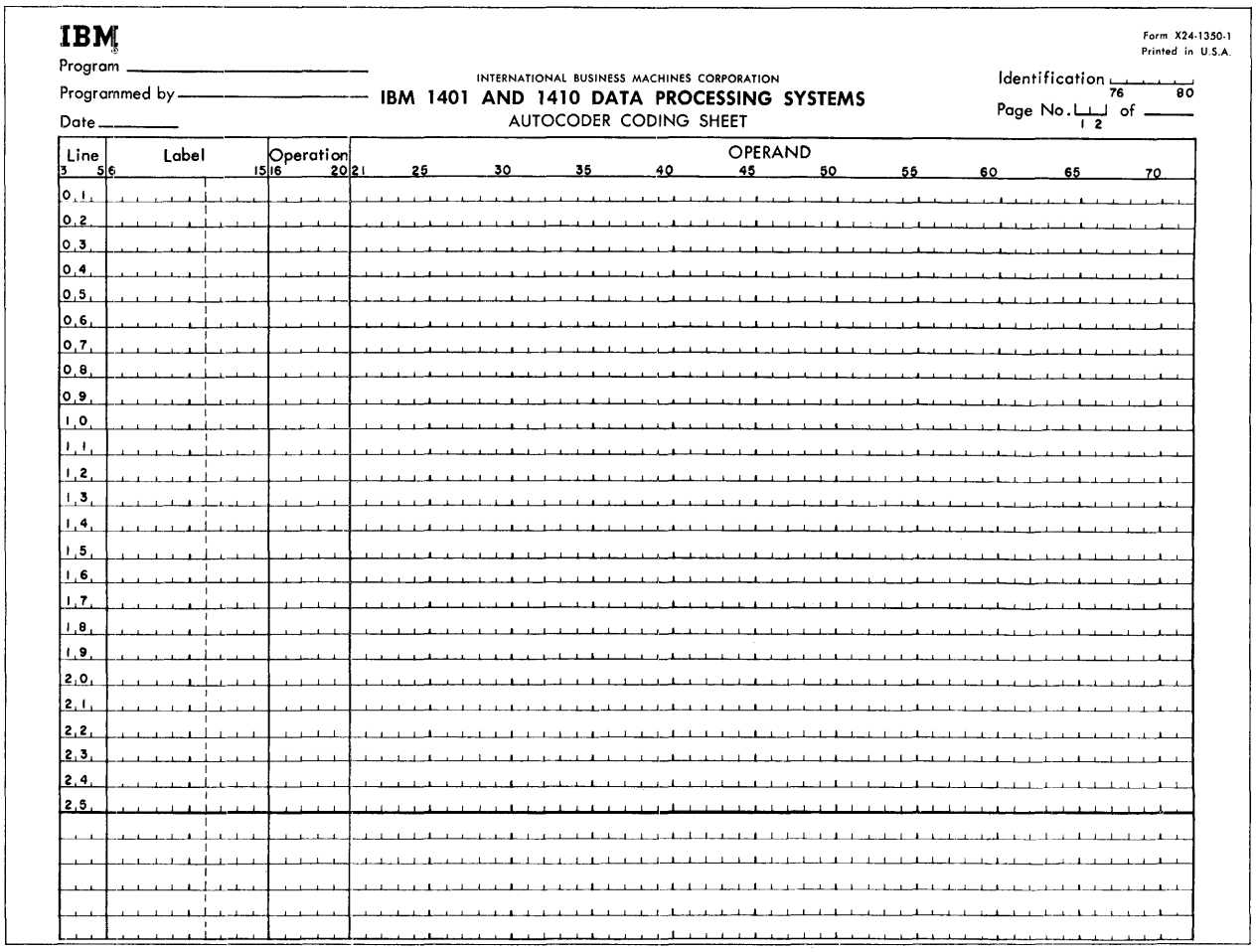


Figure 4. The Coding Sheet Form

cards inserted in a series of chained instructions will break the chain. To avoid this, the operation code should be re-stated and the appropriate operand entered on the first source card following any comments cards. (Comments cards have no effect on the object program and are not included in the object deck.)

Types of Operand Entries

This section explains the form and use of the various entries permitted in the operand field of imperative, declarative, Linkage Loader, and control statements.

The operand field of an Autocoder statement is used to specify a variety of information to the processor. The function of a specific entry is dependent upon the type of Autocoder statement in which it appears. The normal operand usage with each of the five types of Autocoder statements is as follows.

STATEMENT TYPE	OPERAND CONTENTS
IMPERATIVE	Symbolic address(es) to be operated upon by the machine instruction, and a d-modifier, when required
DECLARATIVE	Constants, symbols, and/or control parameters necessary to declare the desired fields
LINKAGE LOADER	Symbolic (or actual) addresses and/or control parameters required to convert the object deck into absolute format
CONTROL	Symbolic (or actual) addresses and constant information indicated by the operation code
MACRO	Parameters of the macro statement (These parameters are discussed in the section entitled, "The Macro System.")

All permissible operand entries are explained and illustrated under the following headings:

Basic Addresses
Address Adjustment
Indexing
Literals
Linkage Symbols
System Symbols
Miscellaneous

Basic Addresses

Basic addresses contained in the operand field of an Autocoder statement are the primary elements of information conveyed to the processor. They can be altered or modified by means of additional elements contained in the operand field.

A basic address is the symbolic or actual representation of a core-storage location of the data field or instruction referred to by the Autocoder statement.

A basic address can be in one of three forms:

Symbolic
Asterisk
Actual

Symbolic

A symbolic address is an operand entry that appears elsewhere in the source program as a label. As a rule, this symbol can be defined as a label either before or after the Autocoder statement in which it appears as an address. The exceptions to this rule are as follows:

1. All symbolic operands appearing in ORG, LTORG, and EQU statements must have been *previously* defined within the same program.

2. The symbolic address appearing in an RSV statement must *precede* any other use of this symbol in a program. (See "RSV — Reserve.")

3. The symbolic representations of index registers (X0, X1-X15) and the common data area (COMMON), must never appear in the label field. They cannot be defined by the user because they are predefined labels in the symbol table maintained by the Autocoder processor.

The instruction in Figure 5 illustrates the use of symbolic addresses. The symbols TOTAL and ACCUMULATE are defined as labels elsewhere in the program. The assembled instruction will cause the contents of the core-storage area labeled TOTAL to be moved to the area labeled ACCUMULATE.

NOTE: A symbolic address will receive upward, downward, or NO relocation, depending on the manner in which the symbol is defined.

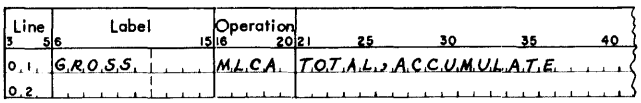


Figure 5. Autocoder Instruction with Symbolic Addresses

Asterisk (*)

An asterisk (11-4-8 punch) can be used as a basic address in an Autocoder statement. When compiling the object program, the processor will replace the asterisk with the relative core-storage address of the last character of the instruction or data field created by the statement in which it appears. However, if an asterisk address is used in a statement that does not cause the generation of an instruction or data area in the object program, the value substituted for the asterisk will be the current location in the object program.

These uses of the asterisk address are illustrated by means of the three Autocoder statements in Figure 6, and are discussed in the order of their appearance.

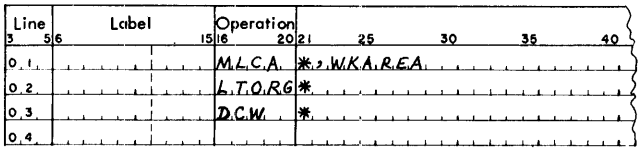


Figure 6. Asterisk Addresses in Autocoder Statements

1. The first statement in Figure 6 illustrates the use of the asterisk in an imperative instruction. Assume that this instruction is assigned to core-storage positions 12340-12351 and that the reference address of WKAREA is 13598 (the low-order position). The assembled instruction is D1235113598T.

2. The second statement in Figure 6 illustrates the use of the asterisk in an Autocoder control statement. In this case, the asterisk represents the current address in the processor's relative location counter. For example, if the last address assigned was 12351, the relative location counter will contain the address 12352 (the representative value of the asterisk).

It should be noted that no data or instructions are generated by the asterisk in this statement. The value the asterisk represents is the location where a specific function is to be performed. In this case, the asterisk means the beginning address for the assignment of previously-encountered literals in the subprogram. (Literals are discussed later in this section; the LTORG statement is discussed under "Control Operating Codes.")

3. The third statement in Figure 6 illustrates the use of the asterisk in a declarative statement. (The DCW statement is discussed under "Declarative Operation Codes.") In this case, a five-position address constant is defined which is the low-order address of the generated field. For example, if the asterisk represented the address 12356, this address would be converted by the processor into a five-position constant assigned to positions 12352-12356, with a word mark over the high-order position.

The asterisk address is used most often in imperative statements with address adjustment, as a means of reducing the number of labels required in a program. (See "Address Adjustment.")

NOTE: Asterisk addresses are assigned upward relocation indicators.

Actual

An actual address is the numeric designation for a core-storage location. Thus, the actual addresses of a 1410 or 7010 with 40,000 core-storage positions range from 00000 to 39999. In coding, the high-order zeros of actual addresses can be omitted (except for location 00000, which must be represented by at least one zero).

Figure 7 illustrates the mixing of actual and symbolic addresses. The statement represents a data move of the contents of the area whose low-order position is at location 22101 to the area labeled MONEY.

CAUTION

All programs written to be run within the framework of the 1410/7010 Operating Systems are relocatable. Since actual addresses are assigned a NO relocation indicator, the programmer must exercise extreme caution when using actual addresses. This is especially true when previously-coded programs written for the IBM 1410 are being converted for use with the Operating System.

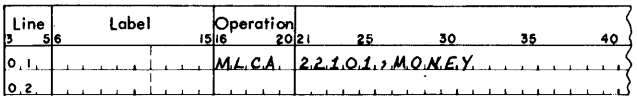


Figure 7. An Actual and a Symbolic Address

Address Adjustment

A basic address, specified in the operand field, can be altered or modified to refer to a different position of core storage. The basic address can be altered during program *assembly* by means of address adjustment; during program *execution* by means of indexing. Address adjustment is discussed below; indexing will be explained in the next subsection.

Address adjustment enables the programmer to reference a location which is a specified number of core-storage positions away from a basic address. Address adjustment is indicated by writing after the basic address a plus or minus sign followed by a one-to-five-digit number that specifies the adjustment factor.

Assume that in Figure 8 the label MANNO is assigned address 15000 and TOTAL is assigned the location 20075. The assembled instruction is A1501220075. The contents of MANNO+12 (15012), *not* MANNO (15000), will be added to TOTAL.

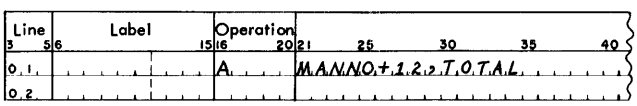


Figure 8. Address Adjustment

The Form * + X00

Asterisk and actual basic addresses, as well as symbolic, can be address adjusted with similar results for the same adjustment factor. However, there is a special adjustment factor unique to the asterisk address, whose use is limited to ORG, LTORG, and BASE1 statements. This special form is *+X00 (asterisk-plus-X-zero-zero). Its function in ORG and LTORG statements is to advance the processor's relative location counter to the address of the next hundreds position in core storage during program assembly (Figure 9). The BASE1 *+X00 value is resolved by the Linkage Loader.

For example, if the relative location counter contains 07214 when the processor encounters the statement in Figure 9, the counter will be automatically incremented to 07300, and subsequent entries will be assigned core-storage locations beginning at this address. The ORG and LTORG statements are discussed in the subsection, "Control Operation Codes." The BASE1 statement is discussed in the subsection, "Declarative Operation Codes."

NOTE: The BASE1 *+X00 statement must appear previous to the ORG *+X00 card or the "no-go" switch will be set.

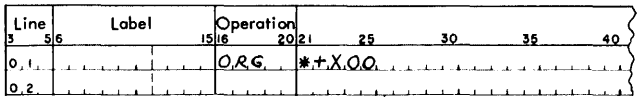


Figure 9. Use of the Special Form *+X00

Multiple Adjustment Factors

Basic A- and B-addresses can be address adjusted, and both can contain more than one address adjustment notation. The number of adjustment factors permitted is limited only by the length of the operand field (up to card column 72).

Indexing

Indexing is a form of address modification which takes effect at the time the object program is *executed.* Autocoder statements can be used to initialize or modify the contents of an index register, or indexing can be used to modify an address within a statement. (See either the 1410 or 7010 *Principles of Operation* reference manual, listed as prerequisites, for a complete description of index register usage.)

The following discussion explains how the Autocoder language can be used to (1) address an index register, (2) initialize or modify the contents of an index register, (3) index an address, and (4) combine indexing with address adjustment.

Addressing an Index Register

Index registers are symbolically referenced within the operand of an Autocoder statement by placing an X before its number. Thus, the *predefined* symbols X1 through X15 designate index registers 1 through 15, respectively.

Autocoder statements that initialize or modify the contents of an index register contain the notation for the index register as a basic address (Figure 10).

In Figure 10, the operation code "ZA" moves the contents of the location labeled EIGHTY into index register 10 and inserts high-order zeros. Thus, if EIGHTY contains "+80", the contents of index register 10 will become +00080, after the execution of the assembled instruction. (The plus sign will be placed in storage as AB bits over the units position.)

NOTE: When an index register is used as a basic address, it is assigned a NO relocation indicator.

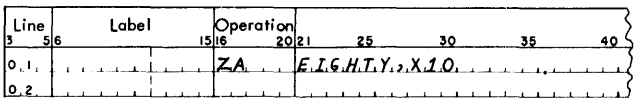


Figure 10. Entering a Numeric Value into an Index Register

Indexing an Address

A basic address is indexed by following the basic address with a plus sign and the notation for the index register (Figure 11).

In Figure 11, assume that the basic address MANNO is relocated to core-storage location 15000 (by the Linkage Loader). If index register 10 contains 00100 (or +00100), the effective address of MANNO+X10 is 15100. Thus, the operation code "MLC" will cause the data at location 15100 to be moved to the location labeled ACCUM. However, if index register 10 contains −00100, the effective address of MANNO+X10 is 14900. Thus, the data at location 14900 is moved to ACCUM by the execution of the assembled instruction.

Both the A-address and the B-address can be indexed.

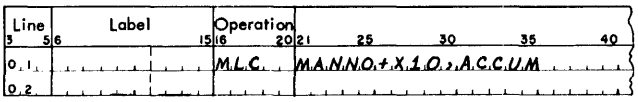


Figure 11. Indexing an Address

Indexing with Address Adjustment

Indexing and address adjustment are permitted in the same operand. The indexing indicator can follow or precede the adjustment factor notation (Figure 12).

The actual location represented by the A-address in Figure 12 will be the basic address (TOTAL), minus 12, plus the contents of index register 1. Assuming that TOTAL is the label for location 03101 and that index register 1 contains 00080, the address of TOTAL−12 +X1 will be 03101−12+00080, or 03169. However, the assembled instruction will be D030Y900140C, assuming that ACCUM is the label for location 00140. The "Y" in the tens position of the A-address is an 8-punch with a zero-zone punch. The zero-zone punch in the tens position is the tag for index register 1.

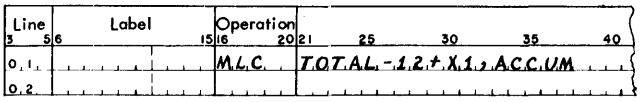


Figure 12. Address Adjustment and Indexing

NOTE 1: Index registers 14 and 15 are reserved for use by the System Monitor. If index registers 14 and 15 are used in an Autocoder program, the processor will properly assemble all references to them, but will flag these references with an "M" (multiple definition), unless the NOFLG parameter is supplied in the appropriate EXEQ or Option card.

By convention, index register 13 is designated as the index register used for linkage with subroutines. The reader will find further details in the publication, *System Monitor*.

NOTE 2: The programmer can assign a label to an index register and subsequently reference it, for the balance of the assembly run, by his own symbolic designation. (See "EQU – Equate," under "Declarative Operation Codes.")

NOTE 3: The special symbolic adjustment notation, X0, indicates to the processor that the associated basic address should be assembled *without* its tag. That is, if a label has been established in the program as relating to an address that is to be adjusted by an index register, it may be used without index register adjustment by following the symbolic address with +X0 (plus-X-zero). See "DA – Define Area," under "Declarative Operation Codes," for an example of index negation.

Literals

Literals permit the programmer to specify a previously-undefined constant within an imperative statement that refers to the constant. Autocoder's ability to process literals enables the programmer to specify a field of constant data, and in the same imperative statement perform a function using the specified field.

As the processor allocates core storage, it automatically reserves a field in which it constructs the constant data. When the imperative statement is translated into machine language, the address to which the constant data was assigned becomes a basic address referencing the field at its low-order position. The processor assigns a word mark to the high-order position of the constant field. Literals can be address adjusted and/or indexed.

All literals are assigned upward relocation indicators.

The four kinds of literals are discussed under the following headings:

Numeric Literal
Alphameric Literal
Area Defining Literal
Address Constant Literal

NOTE: Neither a literal nor its label can be used in the operand field of an EQU statement. (See "EQU – Equate," under "Declarative Operation Codes.")

Numeric Literal

A numeric literal represents a data field of numbers with a sign. Its form in a statement has the following characteristics:

1. It is coded as the basic A- or B-address.
2. It is preceded by a plus or minus sign. When it is assigned to core storage, the sign is placed over the units position.
3. It is an integer whose length is limited only by the available operand positions.
4. Blanks and other non-numeric characters are not permitted within a numeric literal.

In Figure 13, the instruction causes the value "+80" to be added to the contents of the core-storage location labeled TOTAL.

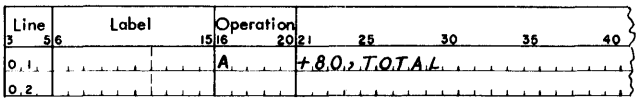


Figure 13. Use of the Numeric Literal

When a numeric literal does not exceed nine digits in length (excluding the sign), it is assigned a relative location only once per program segment (a program is separated into segments by LTORG, SPEND, or END statements), no matter how often it appears in the source program. Longer numeric literals are assigned a relative location each time they are encountered in

the source program. Consequently, to conserve core storage, in cases where multiple use of a "long literal" is necessary, it should be defined as a DCW. (See "DCW — Define Constant with Word Mark," in the section "Declarative Operation Codes.")

If an unsigned numeric literal is desired, it must be entered as an alphameric literal (see below).

Alphameric Literal

An alphameric literal can consist of one or more characters, including the blank. The literal must be preceded and followed by the @ character (4-8 punch). The @ character is permitted within the body of the literal itself; however, a comment on the same line must not contain the @ character. Also, only one alphameric literal can appear in the operand field. These considerations are necessary because the processor assumes everything between the initial and terminal @ characters in the operand field to be part of an alphameric literal.

NOTE: The word separator character (0-5-8 punch) must never be coded as the first character of an alphameric literal.

In Figure 14, the alphameric literal JUNE 14, 1964, is compared to the contents of the core-storage location labeled FLAGDAY.

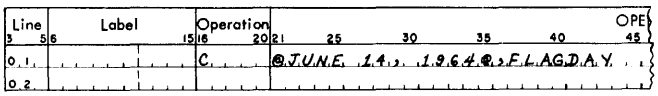


Figure 14. Use of the Alphameric Literal

An alphameric literal, one to nine characters in length (excluding preceding and terminal @ characters), is assigned a relative location only once in a program segment, no matter how often it is used in the source program. Longer alphameric literals are assigned a relative location each time they are encountered in the source program. Consequently, to conserve core storage, in cases where multiple use of a long literal is necessary, the literal should be entered as a DCW. (See "DCW — Define Constant with Word Mark.")

Area Defining Literal

The area defining literal affords a convenient method for simultaneously defining and labeling a field of blanks within the same Autocoder instruction in which it is required. The generated field is assigned a relative location along with other literals in the subprogram, and has a word mark in its high-order position. The field can be referenced by its associated label in other statements in the subprogram.

The area defining literal can be specified only as a basic address in an imperative statement (Figure 15). It consists of a user-created label, followed by a pound sign (#) character (3-8 punch), and a number specifying the length of the field required. Since the pound sign notation is used to define the label only once, it should not be attached to the label in other references.

Figure 15 illustrates how an area defining literal can be used in an Autocoder statement. This literal causes the processor to allocate ten successive positions of core storage, and label the area BUFFERTWO. Ten successive blanks will be loaded into storage at object program load time. Assuming that AMOUNT refers to location 00796, and BUFFERTWO refers to location 00596 (the low-order position of the field), the assembled machine-language instruction that moves the contents of AMOUNT to the area BUFFERTWO is D0079600596C.

NOTE: The following restrictions should be considered when using a label created by means of the area defining literal:

1. When the processor encounters a LTORG or SPEND statement, it terminates the availability of previously defined labels that were created by means of area defining literals. Thus, subsequent references to the label will not be effective. (See "LTORG — Literal Origin," and "SPEND — Subprogram End.")
2. A symbol defined by an area defining literal must never appear in the label field.
3. The area which can be reserved by an area defining literal is limited to 500 positions of core storage. If this limit is exceeded, the processor will reserve only the maximum (500 positions), and attach an "F" flag to the statement in which it appears.

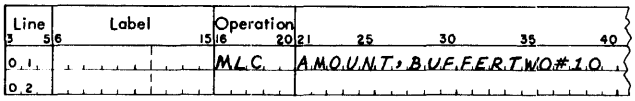


Figure 15. Use of the Area Defining Literal

Address Constant Literal

When a label is used in a source program, the machine address assigned to it by the processor can be defined as a constant (hereinafter referred to as an *address constant*), and used as such by the programmer. The label that is to be defined as an address constant is written as a symbolic basic address of an instruction, with a plus sign preceding this symbol. This signals the processor to create the address constant in a five-position area of core storage. The area contains the machine address assigned to the label by the processor. The relative location assigned to the address constant becomes the basic address represented by the address constant literal. (See (2), Figure 16.)

Figure 16 illustrates how an address constant literal can be used. (The numbers in column 36 correspond to the numbered references in the text which describe

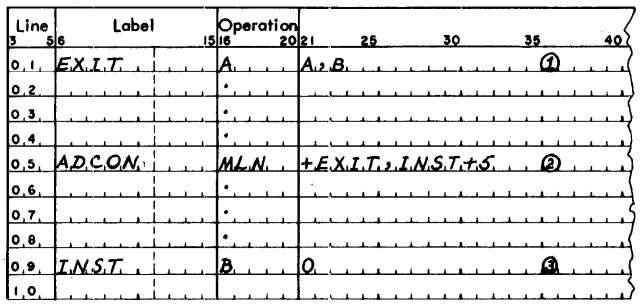


Figure 16. Use of an Address Constant Literal

the functions performed by the statements in the figure.)

1. Assume that the label EXIT is assigned by the processor to location 20600.

2. When the processor assembles the instruction labeled ADCON, the address of the symbolic operand +EXIT will be assigned a five-position area in core storage in a manner similar to other literals. That is, the address of EXIT (20600) becomes the constant data field addressed by the statement.

3. Assume that 32797 is the location of the address constant literal and the location of the instruction labeled INST is 11401. The assembled machine-language instruction of ADCON will be D3279711406A.

When the instruction labeled ADCON is *executed* in the object program, the address constant 20600 is moved to 11406. The instruction labeled INST now becomes J20600b. When the instruction is executed, an unconditional branch to EXIT takes place. Thus, the programmer can write an instruction which moves an actual address into the operand of another instruction at program execution time.

NOTE 1: If address adjustment and/or indexing are used in the operand with an address constant literal, they modify the address of the address constant literal, not the constant itself. (That is, in the example given, address 32797, not 20600.) Both addresses will be assigned upward relocation indicators.

NOTE 2: The address constant (20600) will be unsigned in core storage.

An address constant can also be created by means of the DCW statement. See "DCW — Define Constant with Word Mark."

Linkage Symbols

Relocatable subprograms that have been independently compiled, but designed to run together, require communication. This communication is supplied by the Linkage Loader when it combines the subprograms and converts them to absolute format. The use of linkage symbols facilitates the communication by indicating to the Linkage Loader that a reference address from a different subprogram is required.

The two formats of the linkage symbol are:

1. A *conventional label* (up to ten alphameric characters in length) which can be established as a linkage symbol *only* by appearing either as the name parameter in the TITLE card or as the label of a DEFIN statement.

It can be *referred to*, in an assembly, only by one of the following statements. These statements are discussed later, under their individual headings:

DCWF — Subprogram Address Constant
DCWS — Subprogram Branch Instruction
BASE1 — Base Address of the Subprogram
BASE2 — Base Address of COMMON
PRTCT — Protect
CALL — Subprogram Call

This type of linkage symbol is used to reference entire subprograms by means of operation codes mentioned in its description. Its specification is described in the discussion of statements permitting its use.

2. A *special five-position symbol* consisting of four alphameric characters (the first of which must be alphabetic) with a slash (/) in the fifth position. It is established as a linkage symbol by appearing as the label in a DEFIN statement. This type of linkage symbol can be used in the operand field of any Autocoder statement except EQU, ORG, LTORG, END and SPEND; it cannot be indexed or address adjusted.

It is imbedded within subprogram instructions, and is not assigned an address, but is placed in its corresponding position in the generated machine instruction, as supplied in the Autocoder statement.

This second type of linkage symbol can be used to communicate with a subprogram at a point designated by the symbol. When the instruction is processed by the Linkage Loader the linkage symbol is converted into an absolute machine address.

Figure 17 illustrates the use of a linkage symbol in an Autocoder instruction. The assembled instruction will be ALABE/00029. The Linkage Loader will determine and supply the absolute value for LABE/. Thus, the instruction will be loaded as Axxxxx00029, where xxxxx is the absolute address supplied by the Linkage Loader.

NOTE: Linkage symbols are assigned a NO relocation indicator.

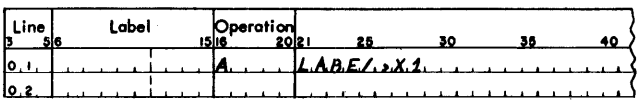


Figure 17. Use of the Five-Position Linkage Symbol

System Symbols

Various elements of the Resident Monitor (including the Resident IOCS) can be referenced in the Autocoder language by the use of their assigned system symbols. All system symbols have the format /ABC/, where ABC is the symbolic name of a location within the Resident Monitor. System symbols are discussed in detail in the publication, *System Monitor*.

System symbols are treated by the Autocoder processor in the same manner as linkage symbols. That is, they are passed intact to the Linkage Loader, where they are resolved into absolute addresses.

Figure 18 illustrates the use of a system symbol in an Autocoder instruction. The assembled instruction will be D/DAT/00029T. The Linkage Loader will search its table of system symbols and substitute the absolute value for /DAT/. Thus, the instruction will be loaded as Dxxxxx00029T, where xxxxx is the absolute value for the system symbol, /DAT/.

NOTE: A system symbol is assigned a NO relocation indicator. (The absolute addresses for system symbols are a permanent part of the Linkage Loader's symbol table.)

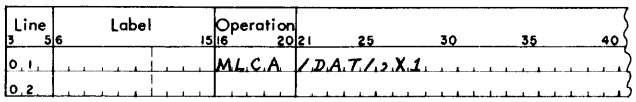


Figure 18. Use of the System Symbol

Miscellaneous

There are several additional elements that are valid in the operand field. These elements are discussed under the following headings:

Operation Modifiers (d-characters)
Blank Operand
Zero as a Basic Address
Special Operand Elements

Operation Modifiers (d-Characters)

The programmer using the Autocoder language is required to supply the operation modifiers associated with certain conditional branch instructions. They are indicated under "Logical Operations" in Appendix D of this publication. The reader is also directed to the appropriate sections in either of the *Principles of Operation* manuals listed as prerequisite reading.

Blank Operand

A blank operand is valid in the following types of operations:

1. In operations where valid A- and /or B-addresses are supplied by the chaining method. For example, in Figure 19 the second A (add) instruction is chained. It takes its addresses from the A- and B-address registers.

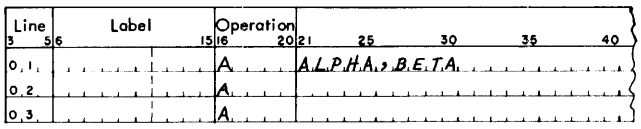


Figure 19. Blank Operand when Chaining

2. In operations that do not require an operand; for example, an ORG statement that directs the processor to use the address in the high assignment counter plus one. (See "ORG — Origin," under "Control Operation Codes"; also, "NOP" and "NOPWM" under "Imperative Operation Codes," for additional examples of the use of the blank operand.)

Zero as a Basic Address

A zero in the operand field is treated as the value 00000 by the processor. Address adjustment and indexing can be used. If this value is address adjusted by a negative factor, a complement number is created. For example, the operand 0−5 means address 99995 (Figure 20).

In Figure 20 the contents of the location indicated by the value 00000−5, plus the contents of index register 10, will be added to HOLD. The assembled machine instruction will be A99RR501000 (assuming HOLD is assigned address 01000). The Rs in the A-address are the result of B bits being placed over 9s in the tens and hundreds position to tag index register 10.

A NO relocation indicator will be assigned by the processor to a zero address.

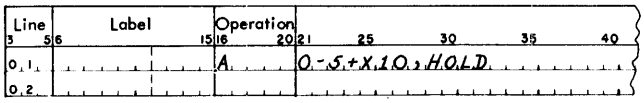


Figure 20. Use of the Zero Operand

Special Operand Elements

Some Autocoder statements require certain special types of information which must appear in the operand field in a specified manner. These statements are listed below. (The interested user is directed to the appropriate subsection of this publication for a full explanation of each statement's format requirements.)

1. DA and DAV statement parameters
2. DA and DAV subentries
3. TITLE statement
4. HEADR statement

This section explains the functions performed by Autocoder operation codes, and the permissible formats of the statements in which they appear. (Macro operation codes, because they are a special form, are explained separately in the next section.)

Every statement written in the Autocoder language must have a specified or implied operation code in the operation field (card columns 16-20). Every permissible operation code belongs to one of four major categories and is discussed in this section according to the functional grouping of the Autocoder statement it represents. The four major categories and their functional groupings are as follows:

Imperative Operation Codes
  Symbolic Machine Instructions
  Special Imperative Statements
Declarative Operation Codes
Linkage Loader Operation Codes
Control Operation Codes
  Assembly Control Statements
  Subprogram Control Statements

Imperative Operation Codes

Imperative operation codes appear in Autocoder statements that are translated by the processor into machine instructions.

Symbolic Machine Instructions

A symbolic machine instruction is an Autocoder statement that is translated by the processor into a machine-language instruction. Permissible symbolic machine instructions include all arithmetic and general data operations, as well as most miscellaneous and branch operations not related to input/output channel status indicators. The processor attaches an "R" flag to mnemonics which can violate the conventions of the Operating System. These mnemonics are listed in Figure 21, which groups and lists every operation code that can be handled by the 1410/7010 Autocoder processor.

The machine-language equivalents of all valid Autocoder operation codes are grouped and listed in Appendix D.

Details concerning the form and use of machine instructions and the operations they perform can be found in the following publications:

IBM 1410 Principles of Operation, Form A22-0526
IBM 7010 Principles of Operation, Form A22-6726

Special Imperative Statements

Two special imperative operation codes have unique meanings for the Autocoder processor, and are provided to add flexibility to program coding.

NOPWM — No Operation; Word Mark

The NOPWM operation code results in the creation of a NOP instruction, and directs the processor *not* to assign a word mark to the operation code of the next sequential instruction. The operand field of this statement is left blank (Figure 22).

At assembly time, the NOPWM instruction (Figure 22) causes the processor to insert in the object program, the machine operation code N (No Operation) with a word mark, followed by an unconditional branch instruction (J01950) without a word mark. The assembled instruction will be N̂J01950b. (Assume NEXT is assigned to location 01950.) At execution time the branch will be inoperative, and the machine will proceed to the next sequential instruction.

Other instructions in the subprogram can be used to set or clear the word mark over the operation code of the branch instruction, as needed. If there is no word mark, the branch instruction is ignored; if the word mark is present, the branch instruction is executed. Thus, the NOPWM operation permits the programmer to set "No Op" switches easily.

NOTE: The effect of NOPWM before a DCWS statement is the same as a simple NOP. That is, an N is generated and a word mark is assigned to the J operation code of the seven-position DCWS.

NOP — No Operation

The NOP operation code results in the creation of a one-character machine instruction. The operand field must be blank (Figure 23).

The NOP statement can also be used to define a program switch. As shown in Figure 23, the processor will translate the mnemonic NOP to N (No Operation) with a word mark. The unconditional branch instruction will be translated as Ĵ01950b, with a word mark over the "J" operation code. (Assume NEXT to be the

CONTROL OPERATION CODES

ASSEMBLY CONTROL CODES

Mnemonic	Meaning
HEADR	Reader Line
RESEQ	Resequence Object Deck
EJECT	Eject Page
PST	Print Symbol Table

SUBPROGRAM CONTROL CODES

ORG	Origin
LTORG	Literal Origin
END	End of Source Program
SPEND	Subprogram End

DECLARATIVE OPERATION CODES

SUBPROGRAM DECLARATIVE CODES

DA	Define Area
DCW	Define Constant with WM
DC	Define Constant
DS	Define Symbol
EQU	Equate

LINKAGE LOADER OPERATION CODES

TITLE	Title of Subprogram
BASE1	Base Address
BASE2	Base Address of COMMON
CALL	Call Subprogram
DEFIN	Definition
PRTCT	Protect Definitions

SUBPROGRAM LINKAGE CODES

DCWF	Subprogram Address Const
DCWS	Subprogram Branch Instr

COMMON DECLARATIVE CODES

DAV	Define Area in COMMON
RSV	Reserve Label in COMMON

IMPERATIVE OPERATION CODES

ARITHMETIC OPERATION CODES

A	Add
S	Subtract
M	Multiply
D	Divide
ZA	Zero and Add
ZS	Zero and Subtract
FA	Floating Add
FS	Floating Subtract
FM	Floating Multiply
FD	Floating Divide
FRA	Floating Reset and Add
FST	Floating Store

COMPARE AND LOOKUP OPERATIONS

C	Compare
LL	Lookup Low
LE	Lookup Equal
LLE	Lookup Low or Equal
LH	Lookup High
LEH	Lookup Equal or High
LLH	Lookup Low or High

IMPERATIVE CODES (Cont'd)

MOVE OPERATION CODES

Mnemonic	Meaning
MLNS	Move Left Numeric Single
MLZS	Move Left Zone Single
MLCS	Move Left Charac Single
MLWS	Move Left WM Single
MLNWS	Move Left Num and WM Single
MLCWS	Move Left Charac and WM Single
MLZWS	Move Left Zones and WM Single
MLNA	Move Left Num to A-Fld WM
MLZA	Move Left Zones to A-Fld WM
MLCA	Move Left Charac to A-Fld WM
MLWA	Move Left WMs to A-Fld WM
MLNWA	Move Left Num and WM to A-Fld WM
MLZWA	Move Left Zones WM to A-Fld WM
MLCWA	Move Left Charac and WM to A-Fld WM
MLNB	Move Left Num to B-Fld WM
MLZB	Move Left Zones to B-Fld WM
MLCB	Move Left Charac to B-Fld WM
MLWB	Move Left WM to B-Fld WM
MLNWB	Move Left Num and WM to B-Fld WM
MLZWB	Move Left Zones and WM to B-Fld WM
MLCWB	Move Left Charac and WM to B-Fld WM
MLN	Move Left Numeric
MLC	Move Left Characters
MLZ	Move Left Zones
MLW	Move Left Word Marks
MLNW	Move Left Num and WMs
MLZW	Move Left Zones and WMs
MLCW	Move Left Charac and WMs
MRN	Move Right Numeric
MRZ	Move Right Zones
MRC	Move Right Characters
MRW	Move Right Word Marks
MRNW	Move Right Num and WMs
MRZW	Move Right Zones and WMs
MRCW	Move Right Charac and WMs
MRNR	Move Rt Num to A-Fld RM
MRZR	Move Rt Zones to A-Fld RM
MRCR	Move Rt Charac to A-Fld RM
MRWR	Move Rt WMs to A-Fld RM
MRNWR	Move Rt Num and WMs to A-Fld RM
MRZWR	Move Rt Zones and WMs to A-Fld RM
MRCWR	Move Rt Charac and WMs to A-Fld RM
MRNG	Move Rt Num to A-Fld GM-WM
MRZG	Move Rt Zones to A-Fld GM-WM
MRCG	Move Rt Charac to A-Fld GM-WM
WRWG	Move Rt WM to A-Fld GM-WM
MRNWG	Move Rt Num and WM to A-Fld GM-WM
MRCWG	Move Rt Charac and WM to A-Fld GM-WM
MRZWG	Move Rt Zones and WM to A-Fld GM-WM
MRNM	Move Rt Num to RM or GM in A
MRZM	Move Rt Zones to RM or GM in A
MRCM	Move Rt Charac to RM-GM in A
MRWM	Move Rt WMs to RM-GM in A
MRNWM	Move Rt Num and WMs to RM-GM in A
MRZWM	Move Rt Zones and WMs to RM-GM in A
MRCWM	Move Rt Charac and WMs to RM-GM in A
MCS	Move Charac and Suppress Zeros
MCE	Move Characters and Edit

SCAN OPERATION CODES

SCNRR	Scan Rt to A-Fld RM
SCNRG	Scan Rt to A-Fld GM-WM
SCNRM	Scan Rt to A-Fld RM or GM-WM
SCNR	Scan Rt to WM in A- or B-Fld
SCNLA	Scan Left to A-Fld WM
SCNLB	Scan Left to B-Fld WM
SCNL	Scan Left to WM in A- or B-Fld
SCNLS	Scan Left to Single Position

IMPERATIVE CODES (Cont'd)

LOGICAL OPERATION CODES *

Mnemonic	Meaning
B	Branch Unconditionally
BU	Branch If Compare Unequal
BE	Branch If Compare Equal
BL	Branch If Compare Low
BH	Branch If Compare High
BCE	Branch If Charac Equal
BBE	Branch If Bit Equal
BAV	Branch If Arith Ovfl
BDV	Branch If Divide Ovfl
BZ	Branch If Zero Balance
BW	Branch If Word Mark
BZN	Branch If No Zones
BWZ	Branch If WM and No Zones
BXO	Branch If Exponent Ovfl
BXU	Branch If Exponent Unfl

MISCELLANEOUS OPERATION CODES

NOP	No Operation
NOPWM	No Opn, Suppress WM
SAR	Store A-Register
SBR	Store B-Register
SW	Set WM at A-Field
CW	Clear Word Mark
CS	Clear Storage
STC	Store Time Clock
SR	Store Register
STCPU	Store CPU Status
RSCPU	Restore CPU Status

RESTRICTED OPERATION CODES *

BEX1	Branch Ext Indic Chan 1
BEX2	Branch Ext Indic Chan 2
BEX3	Branch Ext Indic Chan 3
BEX4	Branch Ext Indic Chan 4
BOL1	Branch Ovlp Proc Chan 1
BOL2	Branch Ovlp Proc Chan 2
BOL3	Branch Ovlp Proc Chan 3
BOL4	Branch Ovlp Proc Chan 4
BB1	Branch If Binary Cd Chan 1
BB2	Branch If Binary Cd Chan 2
BPCB	Branch If Prntr Carr Busy
BPCB1	Branch If Prntr Busy Chan 1
BRCB2	Branch If Prntr Busy Chan 2
BCV	Branch If Prntr Carr Ovfl
BCV1	Branch If Carr Ovfl Chan 1
BCV2	Branch If Carr Ovfl Chan 2
BC9	Branch If Carr Chan 9
BC91	Branch If Carr Chan 9 Chan 1
BC92	Branch If Carr Chan 9 Chan 2
BXPA	Branch and Exit Priority Alert
BEPA	Branch and Enter Priority Alert
JID	Test and Branch
BPI	Priority Test and Branch
STATS	Store and Restore Status
SSF	Select Stackr and Feed
CC	Prntr Carriage Control
BSP	Backspace Tape
WTM	Write Tape Mark on Tape
RWD	Rewind Tape
RWU	Rewind and Unload Tape
CU	Control Unit
MU	Move Mode I/O Command
LU	Load Mode I/O Command
H	Halt

*See Appendix D

• Figure 21. Mnemonic Operation Codes

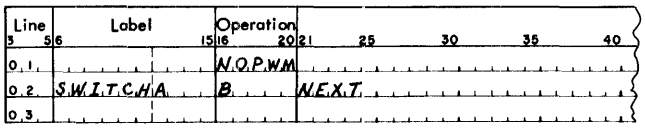


Figure 22. No Operation; Word Mark

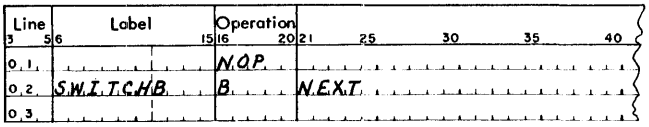


Figure 23. No Operation

label for location 01950.) Thus, the assembled instructions will be N̆J̆01950b, with word marks over N and J.

NOTE: The processor will automatically substitute a twelve-position NOP machine instruction for any statement in the source program containing an invalid operation code. This is done to permit patching of the object deck.

Declarative Operation Codes

Declarative operation codes are used in Autocoder statements that are translated by the processor into data areas, data constants, and address constants used by the object program, and to define and identify labels.

Declarative operations enable the programmer to refer to work areas and constants by their descriptive names (labels) without regard to their actual locations in core storage. For example, if the programmer wants to reserve 20 consecutive core-storage positions for accumulating a final sales total, a declarative operation enables him to reserve the area and refer to it by a label, without concern for the actual address of the field. In this case the label may be TOTAL or ACCUM, or some other label descriptive or meaningful to the programmer.

There are seven declarative operation codes. Their use, function and formats are discussed in the following order.

OPERATION CODE	STATEMENT FUNCTION
DA	Define Area
DAV	Define Area in COMMON
RSV	Reserve Assignment in COMMON
DCW	Define Constant with Word Mark
DC	Define Constant (no word mark)
DS	Define Symbol
EQU	Equate

DA — Define Area

The functions of the DA statement are to reserve and define areas of core storage, such as input, output, or work areas. Fields within each area can be defined by the use of successive statements (called DA subentries), with a blank operation code (columns 16-20). The label associated with the DA statement refers to the high-order address of the reserved area. The label associated with a subentry within the area refers to its low-order address.

DA Statement

The area or areas to be reserved must be defined in the operand field of the DA statement. The operand field of the DA statement contains a parameter of the form B x L, where B (blocking factor) is the number of identical areas to be defined, and L is the length of each area: B x L = Defined Area. In Figure 24, B = 1 and L = 80. Thus, 1 x 80 = 80 positions of core storage.

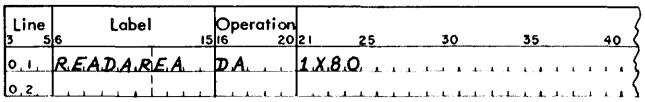


Figure 24. One Area of 80 Positions Defined in a DA Statement

Note that in Figure 25, B = 24 and L = 80. Thus, 24 x 80 = 1,920 positions of core storage to be reserved. The label refers to the high-order position of the total area defined. (The number of positions reserved will be increased if certain additional elements are specified in the operand field. These elements are explained in succeeding paragraphs.)

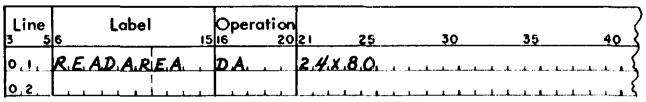


Figure 25. Defining 24 Identical Areas of 80 Positions Each

DA Subentries

The programmer frequently wants to process fields and subfields within a single area, or successive identical areas. These fields must be defined in the DA subentry statements immediately following the DA statement. The operation fields for these subsequent entries must be blank. If a label is used, it refers to the low-order position of the field.

The operand of each subentry must specify the relative location of the field within the defined area. The first location (high-order position) of each defined area is considered location 1. The high-order and low-order positions of the subentry field (relative to loca-

tion 1) are placed in the operand field. These two numbers must be separated by a comma. The processor places a word mark over the high-order position of each field thus defined.

If no word mark is desired, it is necessary to place only the relative low-order position in the operand field. The label will refer to this low-order position, and no word mark will be assigned by the processor. A word mark can, however, be associated with a single-position field by writing the relative location of the position twice, and separating the two numbers by a comma. This is illustrated in Figure 26.

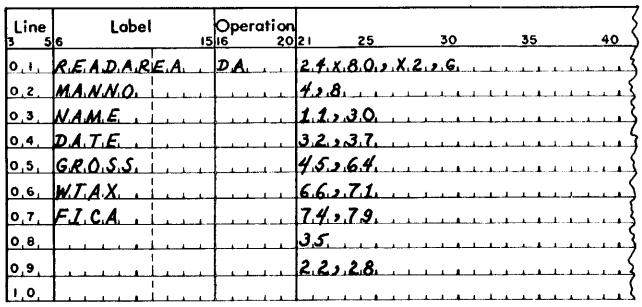


Figure 27. Indexing Fields in a DA Statement

DA Statement Parameters

Five optional parameters can be specified in the operand field subsequent to the B x L parameter in a DA statement. Any or all of these parameters can be used in any order; however, the B x L parameter must be first. These parameters are explained under the following headings:

Field Indexing
Group Mark with Word Mark
Record Mark
Relative to Zero Addressing
"No-Clear" Option

FIELD INDEXING

The labels of all DA subentries will be automatically indexed by the index register noted in the DA statement (Figure 27). However, the label of the DA statement can be indexed only when used as an operand. That is, the label of the DA statement must have the index register notation affixed to it, each time it is referenced, if index adjustment is desired.

In Figure 27, the labels defined in the DA subentries will be automatically indexed by the contents of index register 2. However, if one of the labels used as a symbol in the operand field of an instruction is followed by an indexing notation, this indexing overrides the indexing specified in the DA statement.

Overriding indexing, as explained above, is effective only for the instruction in which it appears. In subsequent instructions, the index indication of the DA statement will be effective. In Figure 28 GROSS is indexed by the contents of index register 3, regardless of the index register indicated in the DA statement in which it was defined.

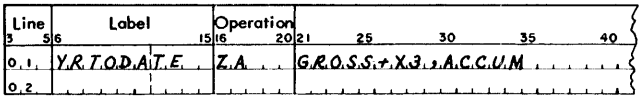


Figure 28. Overriding the DA Statement Index of a Field

The programmer can negate the effect of indexing in a field by putting an X0 (X-zero) in the operand of each instruction in which indexing is not wanted. Here, again, the original index indication is effective in subsequent instructions (Figure 29).

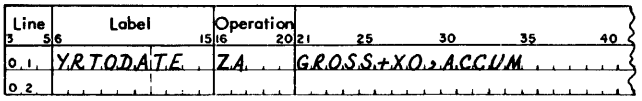
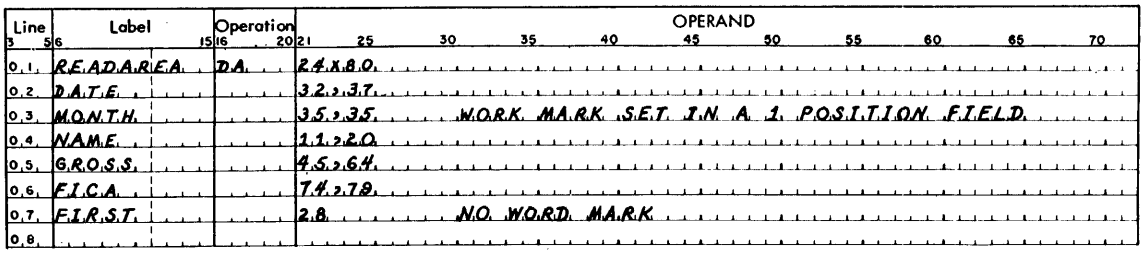


Figure 29. Negating the DA Statement Index of a Field

GROUP MARK WITH WORD MARK

If a group mark with word mark (≑) is desired after the *total* defined area, the character ≑ (12-7-8 punch), preceded by a comma; or the letter G, preceded by



• Figure 26. Areas and Fields Defined in a DA Statement

a comma, must be used as a parameter of the DA statement (Figure 30).

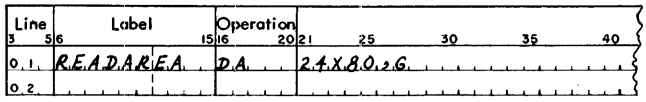


Figure 30. Group Mark with Word Mark after Total Area

RECORD MARK

If a record mark (=) is desired after *each area,* when multiple areas are defined, the character = (0-2-8 punch), preceded by a comma; or the letter R, preceded by a comma, must be used as a parameter of the DA statement (Figure 31).

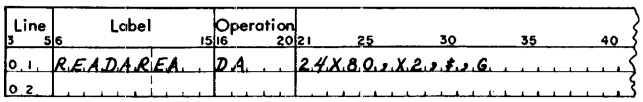


Figure 31. Indicating Record Marks in a DA Statement

The DA statement in Figure 31 will cause the processor to generate a record mark after each area. This means that there will be a total of 1,945 positions of core storage reserved: (24 x 80) + 24 record marks + group mark with word mark = 1,945 positions of core storage. When indexing this statement in a loop to process each area consecutively, 81 positions should be allowed to include the record mark with each area.

RELATIVE TO ZERO ADDRESSING

By writing the character zero, preceded by a comma (,0), as a parameter of the DA statement, the processor will assign addresses to the labels of fields as though the high-order position of the defined area were core-storage position zero. However, the label of the DA statement will still be assigned the address of the high-order position of the area actually reserved by the processor (Figure 32).

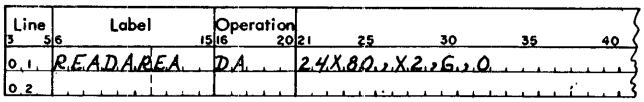


Figure 32. Relative to Zero Addressing

NOTE: If relative to zero addressing is used, the fields will be assigned a NO relocation indicator. Field indexing is required.

"NO-CLEAR" OPTION

The area reserved by the DA statement is normally cleared to blanks before setting word marks and/or record marks. By writing the character N preceded by a comma (,N) as a parameter of the DA statement, the area reserved by the DA statement will not be cleared to blanks when the program containing the area is loaded into core storage from the MJB. (See Figure 33 for an example.) This option is not effective when the program is loaded from the SOF.

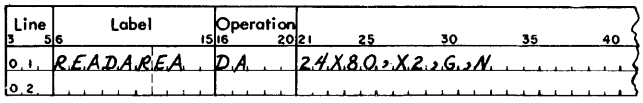


Figure 33. Negating the Clearing of an Area to Blanks

Sample Problem

In this problem, data is to be read from magnetic tape into an area of storage, where it is to be processed. This area, labeled READAREA, is indexed by index register 2 and will have a final group mark with word mark. This is a payroll operation, and each record refers to a different employee. The records are written on tape in blocks of 24. Each record is 80 characters long, and has the following format:

LABEL	POSITIONS	DATA
MANNO	4-8	Man Number
NAME	11-30	Employee name
DATE	32-37	Date
GROSS	45-64	Gross wages
WTAX	66-71	Withholding tax
FICA	74-79	FICA deductions
	35	Month
	22-28	Employee first name

The labels and their associated fields can be listed in any order. (Labels are not assigned to month and employee first name because they will not be needed in this problem.) One way of coding the required elements is illustrated in Figure 27.

The programmer can now, in his source program, write an IOCS macro-instruction to cause data to be read from tape into a storage area labeled READAREA. This causes a block of 24 eighty-character data records to be placed in the 1,920 reserved positions of core storage. This data can now be referred to by the labels DATE, NAME, FICA, etc., in the first of the 24 records that occupy READAREA. The IOCS controls the indexing used to reference and process the data in the subsequent records, 2 through 24.

After all the processing required in the first record is complete, index register 2 is incremented by 80. Because all labels defined by this DA statement are increased by the contents of index register 2, the routine now processes the second data record of the block read into core storage. This process is repeated until all the records have been processed.

Review

Figure 34 summarizes the main points covered in the preceding discussion of the DA statement parameters,

and shows the various entries which can be written in the DA statement. The B x L entry must be first; the other entries can be written in any order.

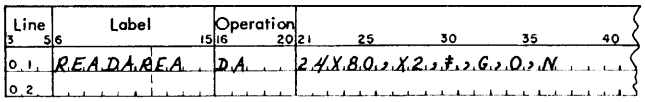


Figure 34. Possible Elements in a DA Statement

The following chart summarizes the main points of the processor's treatment of areas and subentries. Although the index register indication appears in the DA statement, only the subentries are indexed. The DA statement label can be indexed in any instruction in which it appears.

	WORD MARK SET	WORD MARK WHERE SET	LABEL REFERENCED
AREA(s)	NO	. . . . .	HIGH-ORDER
FIELD			
FORM(X, Y)	YES	HIGH-ORDER	LOW-ORDER
FORM(Y)	NO	. . . . .	LOW-ORDER
SINGLE POSITION FIELD			
FORM(X, X)	YES	LOCATION	LOCATION
FORM(X)	NO	. . . . .	LOCATION

DAV — Define Area in COMMON

The DAV statement is used to define an area in COMMON in a manner similar to the DA statement. However, labels of fields are automatically assigned downward relocation indicators (except when relative to zero addressed, resulting in a NO relocation indicator).

The format of the two statements is identical, except for the operation code. For a discussion of conventions for specifying a DAV statement, its subentries, and the five optional parameters available in the DAV operand field, see "DA — Define Area."

Before proceeding to examples illustrating the use of the DAV statement, the following discussion on COMMON is included for those users unfamiliar with its requirements and function.

The Autocoder processor makes the following assumptions when assembling the object deck:

1. The Linkage Loader will relocate downward all addresses referring to COMMON. The downward relocation factor applied will be such that the value contained at the system symbol /AMS/ will be the topmost limit of COMMON; or, if furnished in the program, the BASE2 statement will specify the topmost limit.

2. The topmost address in COMMON will be assigned the value 99999 by the processor, and go downward.

3. The processor will not assemble data or instructions to be loaded into COMMON. For example, a DC or a DCW cannot be used to load data into COMMON. Data can be placed into COMMON only through the execution of the object program.

4. The label COMMON is an indelible entry in the processor's symbol table. It has the address value of 99999, and has a downward relocation indicator attached. If the user defines COMMON as the label of a source statement, it will receive an "M" (Multiple Definition) flag.

Two steps must be taken by the programmer to make use of COMMON in his subprogram. The steps are discussed under the following subheadings:

Assignment of Data Areas in COMMON
Use of Labels Referencing COMMON

Assignment of Data Areas in COMMON

The programmer must define the assignment of data fields and areas within the COMMON data areas. Since COMMON starts at 99999 and goes downward, the programmer determines the total number of positions he requires and assigns space *upward* for each data field and area. The highest position he can use is 99998. (This technique is illustrated in Figures 61 and 62.)

If a program contains two or more subprograms, each of which contains references to the same data fields in COMMON, these subprograms must assign core storage in COMMON in the same way (or in a way compatible with each subprogram's needs). That is, the same data field must be assigned the same relative address in each of the subprograms.

One way of assuring compatible COMMON area referencing is to include identical COMMON area source statements in each of the subprograms involved (i.e., identical DAV, EQU, or RSV statements).

Labels referencing the COMMON Data Area can be defined in any of the following ways:

1. The EQU statement. For example, for the statement A EQU COMMON−10, the label (A) will be assigned the value 99989 (that is, 99999−10). Subsequent labels in the source program can be equated to this label (A). Under these conditions, this label and subsequent labels will be assigned downward relocation indicators.

2. The DAV statement and associated DAV subentry statements. (Normally DAV statements follow an ORG statement that places the DAV area within COMMON.) Labels used in the DAV and subentry statements will be assigned downward relocation indicators.

3. The RSV statement followed by an EQU statement for an actual address value in COMMON. For example:

```
          RSV     LABELB
LABELB    EQU     99985
```

If a subsequent label is to be equated to LABELB, the RSV statement must be used, with the subsequent label as the operand. For example:

```
          RSV     LABELC
LABELC    EQU     LABELB
```

Use of Labels Referencing COMMON

Once the programmer has defined labels referencing COMMON, he can use them as if they were labels assigned to fields and areas within the usual boundaries of the subprogram. Thus, data can be manipulated and operated upon, and IOCS can be employed to read data into and out of areas contained in COMMON.

RESTRICTION

The processor will not assemble data or instructions to be loaded into COMMON. Data can be placed into COMMON only through the execution of the object program. Hence, Autocoder declarative statements, such as the DC and DCW statements, cannot be used to enter data into COMMON.

The effect of the DAV statement is different in several respects from the DA statement, since the defined area is not located within the body of the subprogram, but within the COMMON data area. Hence the user of the DAV statement must note the following:

1. The programmer must specify the exact point within the COMMON data area where the defined area will start. This can be done by means of an ORG statement, the technique of which is illustrated in Figure 61. (The ORG statement is explained in the next section, under the subheading "ORG – Origin.")

2. The defined area in COMMON can be used by other subprograms; the "No-Clear" option enables the programmer to retain the desired contents of COMMON.

3. The DAV subentry statements can be used to set word marks in their respective fields.

4. The DAV statement can be used to clear the area it will occupy in COMMON, and to set record marks and a group mark with word mark.

The coding examples in Figure 61 illustrate the assignment of labels within COMMON. The relative addresses assigned are shown as comments. All labels are assigned downward relocation indicators.

The second part of Figure 35 illustrates all the elements that can appear in the DAV statement.

NOTE: A group mark with word mark will be placed at the topmost *usable* position in COMMON (99998).

An alternative way of assigning the same labels is illustrated in Figure 36. In this example, the fields are not cleared and no word marks are set automatically. Word marks are set during program execution by means of imperative statements.

RSV — Reserve Assignment in COMMON

The RSV statement is used to direct the processor to affix a downward relocation indicator to a label that references a field in COMMON. The label is used as the operand of the RSV statement (Figure 37). The RSV statement must *precede* the use of the label as an operand in the source program.

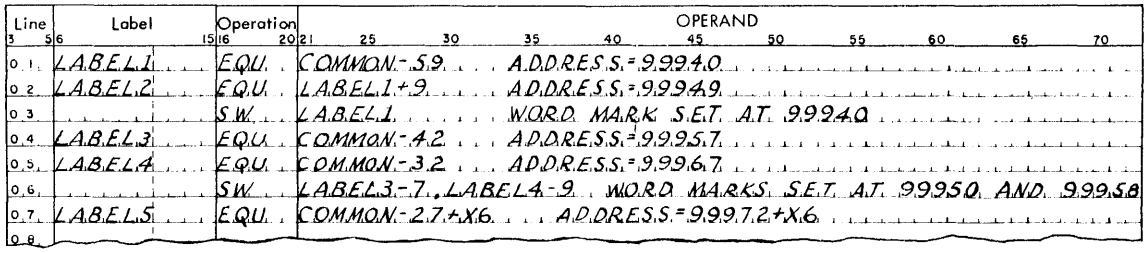
In Figure 37, the operand ALPHA will be assigned a downward relocation indicator each time it appears in a subsequent instruction.

DCW — Define Constant with Word Mark

A DCW statement is used to enter a numeric, alphameric, blank, or address constant into a core-storage

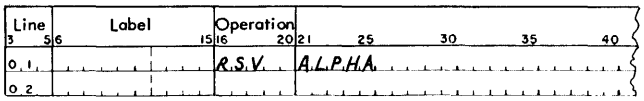
```
Line  Label      Operation  OPERAND
3 5   6      15  16   20    21  25  30  35  40  45  50  55  60  65  70
0 1                ORG      COMMON-59          ADDRESS=99940
0 2   LABEL1        DAV      1X10               LABEL1 ADDRESS=99940 (HIGH-ORDER)
0 3   LABEL2                 1,10               LABEL2 ADDRESS=99949 (LOW-ORDER)
0 4   *                                         ALSO SETS WORD MARK AT 99940
0 5                DAV      1X18               AREA IS FROM 99950 TO 99967
0 6   LABEL3                 1,8                LABEL3 ADDRESS=99957 (LOW ORDER)
0 7   LABEL4                 9,18               LABEL4 ADDRESS=99967 (LOW ORDER)
0 8   *                                         WORD MARKS ARE SET AT 99950 AND 99958
0 9                DAV      2X16,X6            AREA IS FROM 99968 TO 99998
1 0   LABEL5                 3,5                LABEL5 ADDRESS=99972+X6
1 1   LABEL6                 6,16               LABEL6 ADDRESS=99983+X6
1 2   *                                         ALSO WORD MARKS ARE SET AT
1 3   *                                         99968,99970,99973 AND 99984
1 4

Line  Label      Operation  OPERAND
3 5   6      15  16   20    21  25  30  35  40  45  50  55  60  65  70
0 1                ORG      COMMON-62
0 2   COMMONWRITE  DAV      2X30,X2,R,G,N,0    99999 MUST NOT BE USED
0 3   FIELD1                 1,10               SET AT 99998
0 4   FIELD2                 11,30
0 5
```

• Figure 35. Label Assignment in COMMON and DAV Statement Elements



Line	Label	Operation	OPERAND
0.1	LABEL1	EQU	COMMON-59 ADDRESS=99940
0.2	LABEL2	EQU	LABEL1+9 ADDRESS=99949
0.3		SW	LABEL1 WORD MARK SET AT 99940
0.4	LABEL3	EQU	COMMON-42 ADDRESS=99957
0.5	LABEL4	EQU	COMMON-32 ADDRESS=99967
0.6		SW	LABEL3-7,LABEL4-9 WORD MARKS SET AT 99950 AND 99958
0.7	LABEL5	EQU	COMMON-27+X6 ADDRESS=99972+X6
0.8			

• Figure 36. Label Assignment in COMMON



Line	Label	Operation	
0.1		RSV	ALPHA
0.2			

Figure 37. The RSV Statement

location assigned by the processor. The processor places a word mark over the high-order position of the defined constant.

The label of the DCW statement makes reference to the address of the low-order position of the constant. The high-order position will be referenced if the label is indented one column in the label field; that is, if it begins in column 7. In all cases the constant being defined must be left-justified in the operand field.

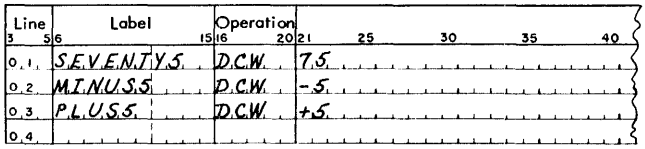
Five types of constants can be defined by means of the DCW statement. They are discussed under the following headings:

Numeric Constants
Alphameric Constants
Blank Constants
Address Constants
Signed Address Constants

Numeric Constants

A numeric constant, defined in a DCW statement by the numeric characters in the operand field, can be preceded by a plus or minus sign, or it can be unsigned. A plus sign causes A and B bits to be placed over the units digit. A minus sign causes a B bit to be placed over the units digit. Unsigned constants will be unsigned in storage (Figure 38).

The first blank column encountered in the operand field terminates a numeric constant. A numeric constant cannot be more than 51 characters long if it is signed; 52 if unsigned.



Line	Label	Operation	
0.1	SEVENTYS	DCW	75
0.2	MINUSS	DCW	-5
0.3	PLUSS	DCW	+5
0.4			

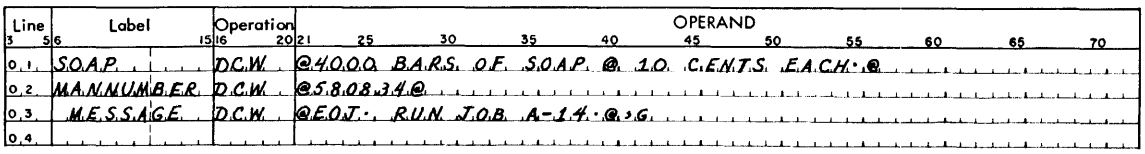
Figure 38. Numeric Constants Defined in DCW Statements

Alphameric Constants

An alphameric constant must be preceded and followed by the @ character (4-8 punch). Blanks and special characters, including the @ character itself, may be used in the body of the constant. The processor, in scanning the operand field, will consider everything to the left of the rightmost @ character a part of the constant being defined. For this reason, the @ character is not permissible in comments on the same line. An alphameric constant can contain up to 50 characters, excluding the initial and terminal @ characters. A comma preceding a G (,G) following the trailing @ character causes the processor to put a group mark with word mark in storage following the last character in the constant. (The label refers to the low-order position of the field, not the group mark position.)

In Figure 39, the group mark with word mark was used with a DCW statement to be written by the console printer. The label of the message was indented to reference the high-order position. A group mark with word mark is required to halt message typing on the console printer.

NOTE: The word separator character (0-5-8 punch) must never be coded as the first character of a DCW alphameric constant.



Line	Label	Operation	OPERAND
0.1	SOAP	DCW	@4000 BARS OF SOAP @ 10 CENTS EACH.@
0.2	MANNUMBER	DCW	@5808344@
0.3	MESSAGE	DCW	@EOJ. RUN JOB A-14.@,G
0.4			

Figure 39. Alphameric Constants Defined in DCW Statements

Blank Constants

A field of blanks can be reserved by placing a # character (3-8 punch) in column 21, followed by a number indicating how many consecutive blank core-storage positions are to be defined (Figure 40). A word mark is set in the high-order position of this field.

NOTE: The number of successive blank constants that can be reserved by a DCW statement is limited to 500 positions of core storage. If this limit is exceeded, the processor will reserve only the maximum (500 positions), and attach an "F" flag to the statement on the assembly listing.

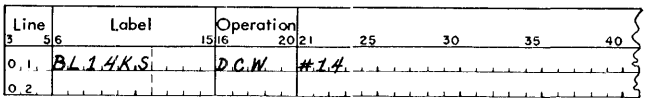


Figure 40. Field of 14 Blanks Defined in a DCW Statement

Address Constants

A DCW statement can be used to define an address constant. The constant is the address of the field whose label is written in the operand. For example (Figure 41), assume that the label MANNO is used in the symbolic program, and that it was assigned the address 00500 by the processor. The programmer can refer to the address of MANNO by using the symbolic label of the DCW statement.

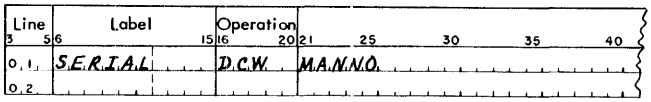


Figure 41. Address Constant

The five-character data field labeled SERIAL (Figure 41) will contain the address of the label MANNO (00500). The Linkage Loader will recognize address constants and adjust them by the proper relocation factor. Thus, SERIAL will contain the relocated address of MANNO.

If an address constant is address adjusted in a DCW statement, the constant is adjusted before it is assigned a storage location. In Figure 42, MANNO (actual address 00500) has been address adjusted by +12. Thus, the location labeled FICA will contain the address constant 00512.

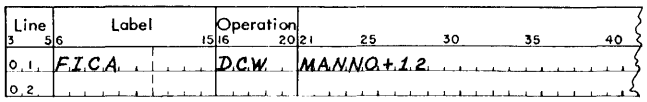


Figure 42. Address Constant with Address Adjustment Defined in DCW Statement

Address constants defined in a DCW statement can be indexed. The zone bit(s) indicating the specified index register becomes part of the constant.

NOTE 1: All address constants receive the same relocation indicators that were assigned to the symbol specified in the operand field.

NOTE 2: An address constant of a linkage or system symbol can be specified, and the desired address will be automatically supplied by the Linkage Loader. However, this form of address constant cannot be address adjusted or indexed.

Signed Address Constants

An address constant defined in a DCW statement can be signed. A and B bits will be generated by the processor over the units position, if the plus (+) sign was placed before the operand. The units position will contain a B bit if the minus (−) sign was used (Figure 43).

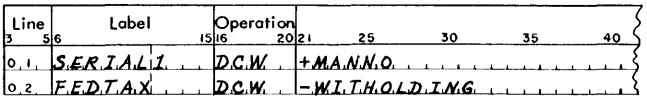


Figure 43. Signed Address Constants Defined in DCW Statement

Implied DCW Operation Codes

If a number of constants are to be defined in succession, only the first statement requires the mnemonic DCW, in the operation field (Figure 44).

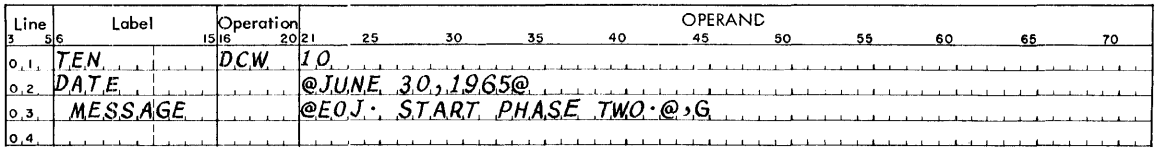


Figure 44. Successive DCW Statements with Blank Operation Columns

DC — Define Constant (no word mark)

The function performed by the DC statement, and the permissible forms of the constants, are identical to those described for the DCW statement. The only difference is that the word mark is absent when the constant is assigned to core storage (Figure 45).

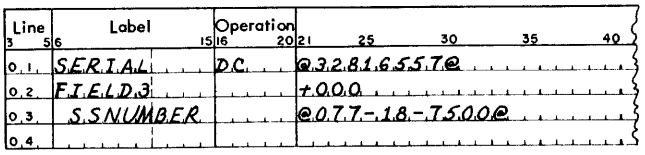


Figure 45. Successive DC Statements with Blank Operation Columns

NOTE: The restriction on the use of an initial word separator character in the DCW statement defining an alphameric constant does not apply to the DC statement.

DS — Define Symbol

The DS statement is used to label and define an area within the subprogram. No information is entered into the area, no word mark is assigned by the processor, and the area is not cleared prior to reservation. The programmer specifies the size of the area, and designates the symbolic label by which it will be referenced. The number of desired consecutive positions of core storage is written in the operand field (Figure 46). The label refers to the low-order position of the area. However, if the label is indented one place, that is, if it begins in column 7, the label will refer to the high-order position. A label is not mandatory.

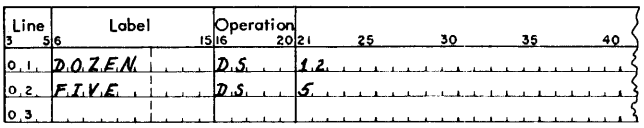


Figure 46. Defining Twelve-Position and Five-Position Areas in DS Statements

Figure 46 illustrates the form of the DS statement. The first entry, labeled DOZEN, defines an area twelve positions long. The second entry, labeled FIVE, defines an area five positions in length.

EQU — Equate

The EQU statement is used to define either a second symbol to reference a specific location, or a symbol for a location not previously labeled. The symbol to be defined is specified in the label field, and the representation of the location to be "equated" is specified in the operand field.

An EQU statement can be used to assign a symbolic label to each of the following:

Actual or symbolic address
Adjusted or modified address
Index register
Asterisk address

Actual or Symbolic Address

The symbol to be defined is specified in the label field. The operand field can contain an actual or symbolic address. If a symbolic address is specified in the operand field, it must have appeared as a label prior to this point in the subprogram. If this condition is not met, the label will *not* be defined.

SYMBOLIC ADDRESS

The EQU statement in Figure 47 will cause the processor to assign the same address to the label INDIVIDUAL that is assigned to the symbol MANNO. Thus, INDIVIDUAL has been equated to MANNO — both labels refer to the same core-storage location and are assigned the same relocation indicator by the processor.

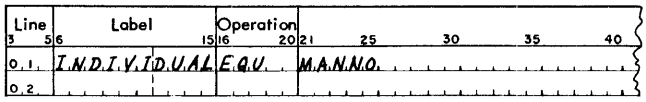


Figure 47. Equating a Symbolic Address

ACTUAL ADDRESS

The EQU statement in Figure 48 will cause the processor to assign the label ACCTNO to machine location 25000.

NOTE: Labels equated to actual addresses will be treated as absolute values and given a NO relocation indicator.

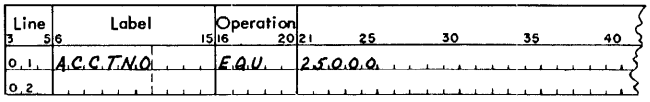


Figure 48. Equating an Actual Address

Adjusted or Modified Address

The operand of an EQU statement can be address adjusted or indexed. The same relocation indicators assigned to the address adjusted and/or indexed operand will be given to the defined label.

EQUATING TO AN ADDRESS ADJUSTED OPERAND

In Figure 49, the processor assigns the label WHTAX to a location ten storage positions lower than the location labeled FICA. That is, if FICA is assigned location 00890, WHTAX will be equated to FICA−10, or 00880.

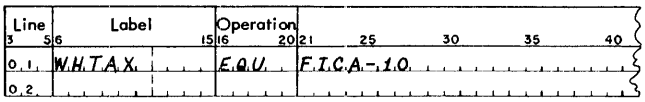


Figure 49. Equating with Address Adjustment

EQUATING TO AN INDEXED OPERAND

A label can be equated to an indexed operand. In Figure 50, CUSTNO is equated to JOB indexed by index register 3, not JOB alone. That is, CUSTNO will be assigned the address of JOB, with A and B bits over the tens position (the tag for index register 3).

An indexed operand can also be address adjusted.

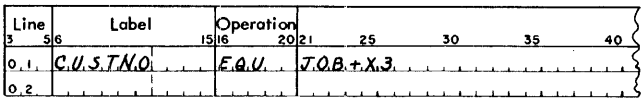


Figure 50. Indexing in an EQU Statement

EQUATING TO TWO LABELS

A label can be equated to the algebraic sum or difference of the values represented by two symbolic labels, in the form C EQU A±B (+ or −), where A and B are previously defined labels (Figure 51). (The treatment of the form C EQU A+B, where B is the label for an index register, is explained under the subheading "Index Register.")

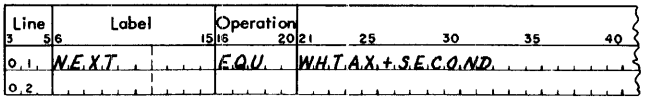


Figure 51. Equating a Label to Two Symbolic Labels

In Figure 51, if WHTAX references core-storage location 08000 and SECOND references location 01500, the label NEXT will be assigned the value equal to the sum of both; that is, 09500. If a minus sign is used instead of a plus sign, the label NEXT will be assigned a value equal to their difference, that is, 06500.

NOTE 1: No further adjustments (or indexing) are allowed.

NOTE 2: If either A or B is assigned a NO relocation indicator, C will be assigned the relocation indicator of the other label. If neither A nor B is assigned a NO relocation indicator, C will be assigned a NO relocation indicator.

Index Register

The label of an EQU statement can be defined as an alternative symbolic name for an index register. The processor's predefined label for the index register to be equated (X1-X15) is written in the operand field. Figure 52 illustrates this technique.

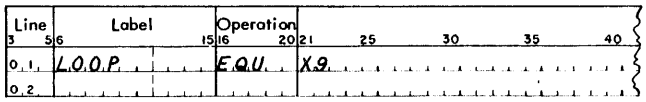


Figure 52. Labeling an Index Register with an EQU Statement

In Figure 52, the label LOOP will be assigned by the processor to index register 9. Thus, index register 9 can be referred to as LOOP, instead of X9. This use of a symbolic index register is illustrated in Figure 53.

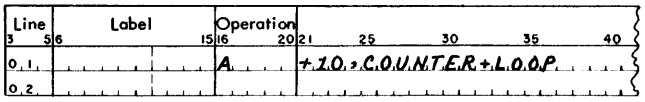


Figure 53. Use of a Symbolic Index Register

NOTE 1: Symbolic index registers must be equated *before* they can be used in an instruction.

NOTE 2: A label can be equated to an actual or symbolic operand indexed by a symbolic index register, in the form C EQU A+B. The label B represents the index register. The label C will be assigned the same relocation indicator assigned to A.

Asterisk

A label can be equated to an asterisk address, with or without address adjustment and/or indexing (Figures 54 and 55).

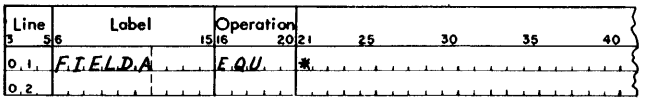


Figure 54. Equating an Asterisk Operand

In Figure 54, the label FIELDA refers to the location next available in the program. If the next available location in the program is 00698, FIELDA is equated to 00698.

Figure 55 illustrates the use of the EQU statement with an address adjusted asterisk operand. If the asterisk refers to location 00698, the label FIELDA is equated to 00710.

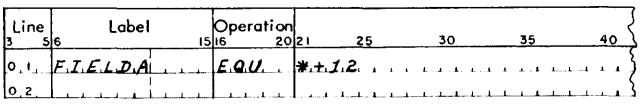


Figure 55. Asterisk Operand with Address Adjustment

Linkage Loader Operation Codes

The output of an Autocoder assembly is in the intermediate form of a relocatable object deck. It is during the execution of the Linkage Loader that the object deck is converted into absolute form. The absolute object program consists of relocated subprograms with linkages to each other, as well as to the Resident Monitor. In order to accomplish this conversion, the Linkage Loader requires certain specific information, presented according to the conventions of the System Monitor.

The Autocoder language provides eight statements by which the programmer can communicate the subprogram's requirements to the Linkage Loader. Although these statements can be added to the object deck of a subprogram after its assembly, their inclusion in the form of an Autocoder statement affords the following benefits:

1. Automatic conversion of the statement into the format required by the Linkage Loader
2. Error diagnosis of the statement as it relates to the subprogram being assembled
3. Automatic sequencing (card columns 73-75) and identification (card columns 76-80) of the statement within the relocatable object deck

The Linkage Loader statements permitted in a subprogram assembly contain the following operation codes:

OPERATION CODE	STATEMENT FUNCTION
TITLE	Title of Subprogram
BASE1	Base Address
BASE2	Base Address (COMMON Data Area)
CALL	Subprogram Call
DCWF	Subprogram Address Constant
DCWS	Subprogram Branch Instruction
DEFIN	Definition
PRTCT	Protect

In the following descriptions of these statements, only permissible formats and a brief description of their functions are given. The reader is directed to the publication, *System Monitor*, for details concerning the functions performed by these Linkage Loader control operations.

TITLE — Title

The TITLE statement is used to establish an identifying name for a subprogram, to indicate the size of the COMMON data area the subprogram will use, and to state the lowest origin point in the subprogram.

Entries

The name of the subprogram must be a conventional label (1-10 alphameric characters in length), and appears as the first entry in the operand field. This name can be used in DCWS, DCWF, BASE1, BASE2, PRTCT, and CALL statements. (These statements are individually explained in this publication.)

NOTE: All names of IBM-provided modules in the Operating System start with "IB". The user should be aware of this standard to avoid duplicating the name of a module when naming a relocatable subprogram.

The specified size of the COMMON data area required is written as the second operand entry in the form of an integer, one to five positions in length, and is optional. If it is omitted, the processor will leave the corresponding field in the object program's TITLE card blank, and the Linkage Loader can subsequently give no warning if the program and COMMON data area should overlap.

The lowest origin point is the third operand entry and can be omitted. If it is omitted, the processor will place into the object program's TITLE card the lowest address assigned during the assembly. If the third entry is included, the automatic computation of the processor is negated, and the value declared by the entry is passed on to the Linkage Loader through the TITLE card. This entry can be an actual value or a label within the assembly.

Although the third entry is normally omitted, it can be useful under the following conditions:

1. When a program being assembled contains one or more SPEND statements and,
2. When the low origin points of the subprograms are different and it is necessary that these differences be indicated.

Format Considerations

The entries are written in the operand field, and separated by commas. If the second entry is omitted and a third entry is used, the third entry must be separated from the first entry by two commas. If both the second and third entries are omitted, only the name is required, with no trailing commas (Figure 56).

NOTE 1: The TITLE card should be the first source statement in a subprogram, with the exception of any of the Assembly Control statements explained later.

NOTE 2: The TITLE card is the first card in the object deck, with the parameters rearranged to meet the requirements of the Linkage Loader.

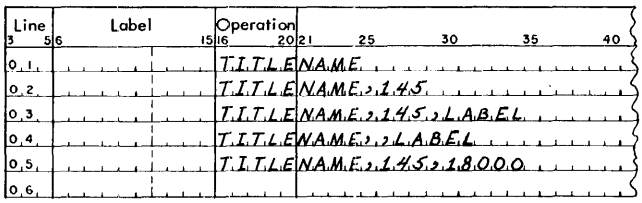


Figure 56. Permissible Forms of the TITLE Statement

NOTE 3: The contents of the system symbol /DAT/, the date currently stored in the Resident Monitor, will be placed in the date field (columns 6-10) of the rearranged TITLE card in the object deck. This date will also appear in the heading line of the assembly listing.

BASE1 — Base Address

The BASE1 statement can be used to control the Linkage Loader's relocation factor. The following operands are permissible in a BASE1 statement:

Actual
Symbolic
Asterisk plus X00 (*+X00)

Actual

The actual core-storage location is written in the operand field (Figure 57).

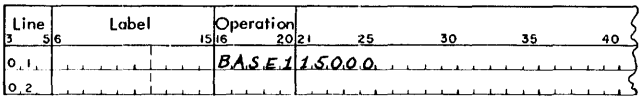


Figure 57. Use of an Actual Address in a BASE1 Statement

Symbolic

A symbolic address can be a subprogram name or a linkage symbol defined in a subprogram that is processed by the Linkage Loader *prior* to the subprogram in which the BASE1 statement appears. This symbolic address is defined by means of the TITLE or DEFIN statements (see below).

In Figure 58, the operand PROGRAM2 is the name of a subprogram that is defined by a TITLE statement.

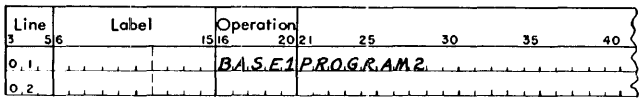


Figure 58. Symbolic Address in a BASE1 Statement

NOTE: A blank operand sets the relocation factor to base zero.

Asterisk Plus X00 (* +X00)

If an *+X00 is the operand of a BASE1 statement, the relocation factor will be incremented so that program loading will continue at the next even-hundred location. For example, if the asterisk has the *relocated* value of 18279 the relocation factor will be adjusted to 18300. If the value of the asterisk is already an even-hundred address, that value will remain the relocation factor.

NOTE: The *+X00 form of the BASE1 statement must be used in conjunction with an ORG statement containing *+X00 in the operand field. (See "ORG — Origin," under "Control Operation Codes.")

BASE2 — Base Address (COMMON Data Area)

The BASE2 card is used to set the upper limit of the COMMON data area. The operand of the BASE2 card, which can be either a linkage symbol or an actual address, is as described under the BASE1 statement. The form *+X00 is not valid in a BASE2 statement.

CALL — Subprogram Call

The CALL statement provides the Linkage Loader with the name of a subprogram that is to be loaded from the System Library file or from the GO file. The operand of the CALL card is the name of the subprogram to be processed by the Linkage Loader (Figure 59). When the CALL card produced by the processor is loaded with the object deck, the operand of the CALL statement is passed directly to the Linkage Loader.

In Figure 59, the linkage symbol IOMODULE must have been defined by a TITLE statement in the subprogram being called.

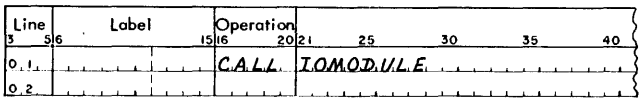


Figure 59. The CALL Statement

DCWF — Subprogram Address Constant

The DCWF statement is used to specify an address constant (similar to the DCW unsigned address constant) and to create an imbedded call for the named subprogram. The operand of the DCWF statement must be the name of the requested subprogram as specified in its TITLE card if the operand has not yet been located and processed (forward reference). Otherwise, it may be any linkage symbol.

The label of a DCWF always refers to the high-order position of the address constant. This label must not be indented.

The DCWF statement instructs the Linkage Loader to perform the following functions:

1. Include the named subprogram with the program that contains the DCWF statement.

2. Provide an address constant of the *relocated* origin point of the named subprogram. This address constant is placed into the position in which the DCWF statement appears in the source program (Figure 60).

This operation code assumes a higher mnemonic value if thought of as representing a DCW—Five-position address.

In Figure 60, assuming that the subprogram THIRD was relocated to start at location 10400, the address constant at object time will be ǂ10400.

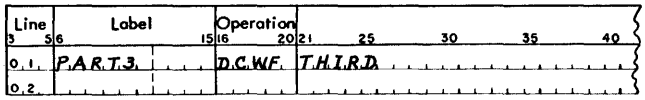


Figure 60. Use of the DCWF Statement

DCWS — Subprogram Branch Instruction

The DCWS statement has the same format as the DCWF statement (Figure 61); however, a seven-position unconditional branch is constructed instead of a five-position address constant. This statement causes the following:

1. The named subprogram will be located and processed by the Linkage Loader.

2. A seven-position unconditional branch instruction (branching to the relocated origin point of the named subprogram) will be constructed by the Linkage Loader, and placed into the position in the subprogram in which the DCWS appears.

This operation code assumes a higher mnemonic value if thought of as representing a DCW—Seven-position branch.

The DCWS statement in Figure 61 results in an unconditional branch instruction being placed into that position in the subprogram in which the DCWS appears. If the subprogram THIRD is relocated to the origin point 14000, the resulting branch instruction is ǂ14000b.

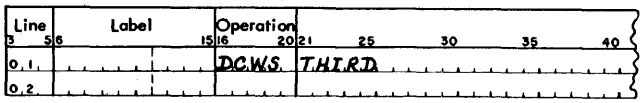


Figure 61. Use of the DCWS Statement

DEFIN — Definition

The DEFIN statement is used to define a linkage symbol. This linkage symbol can represent an entry point or data field within the subprogram being assembled. This symbol can be referenced by other subprograms, which may be assembled separately. The DEFIN statement can be used to establish linkage symbols of both formats: LABE/ or the conventional label type.

Figure 62 illustrates the format and use of the Autocoder DEFIN statement together with the object card produced. Consider the linkage symbol TABL/ to be an address appearing in one or more subprograms to be loaded with the subprogram containing the DEFIN statement. When PROG1 is loaded, the operand of the DEFIN card is relocated by the PROG1 relocation factor, producing an absolute address for each usage of TABL/. The Linkage Loader will replace every usage of TABL/ with the relocated DEFIN value. For example, if in another subprogram the instruction

```
MLCA    SAM#5,TABL/
```

appears, the object card produced by the Autocoder processor contains Ď00700TABL/T (assuming 00700 is the value of SAM). When the subprogram containing this instruction is loaded with PROG1, the Linkage Loader resolves both the A and B operands. The A operand is incremented by the upward relocation factor. The B operand, TABL/, is replaced by the relocated DEFIN value. If 14000 is the relocation factor for the subprogram containing TABL/ as the label of the DEFIN statement, and 15600 is the relocation factor for the program containing TABL/ as an operand, the instruction in storage is Ď1630015J?0T.

NOTE: The DEFIN statement is the only one in which the five-position linkage symbol LABE/ can appear in the label field.

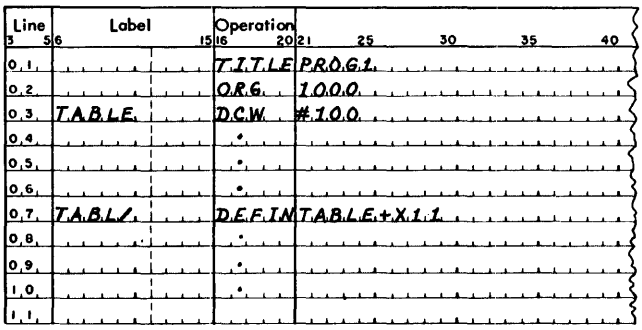


Figure 62. The DEFIN Statement

PRTCT — Protect

The PRTCT statement is used to set a limit for erasure of linkage symbols from the Linkage Loader's symbol table. The Linkage Loader will retain in its symbol table all linkage symbols equal to or higher than the address value specified by the operand of the PRTCT statement.

The operand of a PRTCT statement can be either a linkage symbol or an actual value (Figure 63). Neither indexing nor address adjustment is permitted.

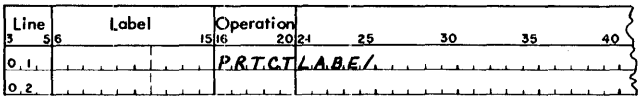


Figure 63. The PRTCT Statement

Control Operation Codes

Control operation codes are used in Autocoder statements that give directions to the processor in performing specified operations at assembly time. There are two types of control statements, discussed under the following subheadings:

Assembly Control Statements
Subprogram Control Statements

Assembly Control Statements

Assembly control statements are related to the assembly listing, object program cards, and the cross reference listing. They do not affect the subprogram being assembled. The assembly control operation codes are:

OPERATION CODE	STATEMENT FUNCTION
HEADR	Header Line
RESEQ	Resequence Object Cards
EJECT	Eject Listing Page
PST	Print Symbol Table

HEADR — Header Line

The HEADR statement (Figure 64) is used to direct the processor to perform the following functions:

1. Cause printing of specified information in the header line on each page of the assembly listing. The header line contains the contents of the HEADR card, columns 21-72. If the header card is absent from the source deck the processor will move blanks into the printing positions.
2. Cause punching of the identification in columns 76-80 of the HEADR card into the same columns of each output card in the object deck. This identification will also appear in the header line of the listing page.
3. Cause the card sequence count to be set at 001 in the object deck.
4. Cause printing to begin on a new page during listing.

The information printed in the header line can be written anywhere in the operand field, columns 21-72, of the HEADR statement.

The identification written in the identification field, columns 76-80, can consist of special, as well as alphameric, characters. This identification can be changed by a RESEQ statement. (See "RESEQ — Resequence.")

If another HEADR statement appears elsewhere in the source program, it causes printing to begin on a new page during listing; the new information will appear in the heading line and all subsequent heading lines, and in the object deck. The card sequence count of the subsequent program will start at 001. (See Figure 64.)

NOTE: The HEADR statement is permitted between the SPEND and TITLE statements, allowing the next subprogram in the assembly to be listed under its own page heading, and the object deck to contain the new information.

RESEQ — Resequence

The RESEQ statement allows the programmer to separate his object deck into logical groups or blocks by controlling the sequence number and identification field of the object program cards produced by the processor (Figure 65). In this respect it is similar to the HEADR statement.

The RESEQ control operation directs the processor to perform the following functions:

1. Punch the new identification supplied by this statement, card columns 76-80, into columns 76-80 of subsequent object cards, and replace the identification in the header line.
2. Set the card sequence count to 001 in the object deck.
3. Direct printing to begin on a new page during listing.

There are only two entries in a RESEQ statement: RESEQ in the operation code field, and the identification of columns 76-80 (Figure 65).

NOTE 1: A RESEQ statement can appear between the SPEND and TITLE statements.

NOTE 2: The execute card, produced by the processor from the END statement, contains the card sequence number 999. This permits the insertion of sequenced patch cards.

EJECT — Eject

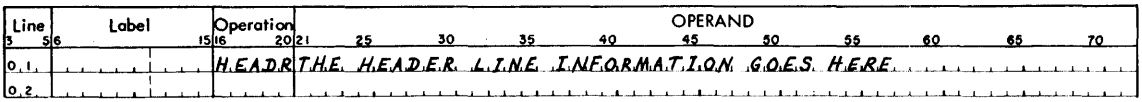
The EJECT statement causes printing to continue on a new page of the assembly listing, thereby separating routines or program sequences in the output listing. The statement consists of EJECT in the operation code field. The rest of the card remains blank (Figure 66).

NOTE: An EJECT statement can be placed between a SPEND and TITLE statement, if the next subprogram listing is to start at the top of a new page.

PST — Print Symbol Table (Cross Reference Listing)

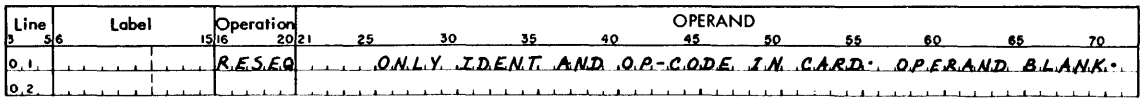
The PST statement (Figure 67) is used to indicate to the processor that a cross reference listing is desired of all symbols used in the program.

This cross reference listing appears after the assembly listing, and has the following features:



Columns 76-80 contain the identification of the subprogram that is printed in the heading line and punched into the object cards.

Figure 64. The HEADR Statement



Columns 76-80 contain the identification of the subprogram that is printed in the heading line and punched into the object cards.

Figure 65. The RESEQ Statement

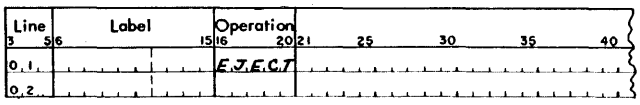


Figure 66. The EJECT Statement

1. It provides a list of every symbol used, in alphabetic order, followed by the sequence number of the statement in which it was defined and the sequence number of every statement referencing it within the program.
2. Undefined symbols and multiple definition of symbols are indicated.
3. It distinguishes between index registers used as basic addresses and those used for indexing.
4. It separates the literals according to program segment. (This literal separation is ineffective on programs containing more than nine segments.)

NOTE 1: Since the cross reference listing does not require previous definition of the symbolic operands in ORG, LTORG and EQU statements, the undefined indications for these statements will appear in the cross reference listing *only* if the label is not subsequently defined.

NOTE 2: The sequence numbers of equated index registers used as basic addresses are listed with the associated symbolic index registers. The sequence numbers of symbolic index registers used as modifiers are listed with the associated actual index registers (i.e., X1-X15).

The PST statement consists of the mnemonic operation code and a blank operand field (Figure 67).

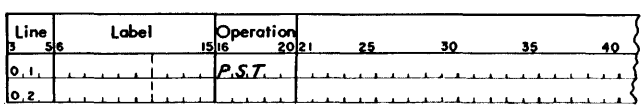


Figure 67. The PST Statement

Subprogram Control Statements

Subprogram control statements govern the form and sequence of subprograms, and supply the programmer with flexible control over the assembly process. The subprogram control operation codes are:

OPERATION CODE	STATEMENT FUNCTION
ORG	Origin
LTORG	Literal Origin
END	End Subprogram and Assembly
SPEND	End Subprogram

ORG — Origin

Sequential core-storage addresses are automatically assigned by the Autocoder processor to instructions, constants, and work areas. These assignments are ordinarily made in the order in which the source program is read during the assembly process. The ORG statement, however, can be used to instruct the processor to break the sequential order of address assignments, and continue from another specified address.

It should be noted that the ORG statement does not absolutely determine where the program will reside in core storage after relocation. The ORG statement controls address assignment during the assembly process, and the assembly listing indicates the *relative placement* of the various program elements within a subprogram. In this respect, the programmer retains the traditional freedom of controlling the relative locations of blocks of coding and data within a subprogram.

Unless a low origin is specified as the third parameter of the *source* TITLE card, the Autocoder processor determines the low origin and places it into the *object* TITLE card. It is this low origin, whether specified or automatically generated, which, in conjunction with the BASE1 value, determines the relocation factor to be applied to each upward-relocatable element in the object program.

The relocation factor is the BASE1 value minus the low origin value in the TITLE card. The address occu-

LOW ORG. FROM ORG. CARD	LOW ORG. FROM SOURCE TITLE CARD	LOW ORG. PLACED IN OBJECT TITLE CARD	BASE 1 VALUE	RELOCATION FACTOR	LOW ORG. IN CORE STORAGE
00000	None	00000	12000	12000	12000
05000	None	05000	12000	07000	12000
05000	07000	07000	12000	05000	10000
05000	None	05000	10000	05000	10000
12000	None	12000	12000	00000	12000

Figure 68. Determining the Load Address

pied by a unit of information in core storage is equal to the compiled address plus the relocation factor. See Figure 68 for representative examples.

If the low origin in the object TITLE card equals the true *compiled* low origin of the program (the normal case), the program will load at the BASE1 value.

If the BASE1 value equals the low origin, the program will load at the low origin, and the address within the program in core storage will equal the addresses in the assembly listing.

When assembling a program containing SPEND cards, the programmer can force the TITLE card to reflect the true low origin of each subprogram by specifying the respective low origin in each TITLE statement. Otherwise, Autocoder will place the lowest origin point of the entire set of subprograms into each TITLE card. (The use of identical TITLE statement values is useful in designing program overlays.)

If an ORG statement is not used, address assignment (by the Autocoder processor) will automatically begin at 00000.

The following types of operands are permissible in an ORG statement:

Actual
Symbolic
Blank
Asterisk
Asterisk plus X00 (*+X00)

ACTUAL

An actual address directs the processor to start assigning locations at the address specified. For example, in Figure 69 the address assignment will begin at 00500.

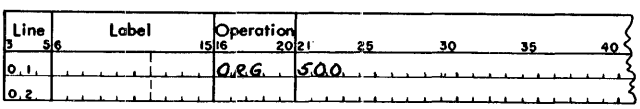


Figure 69. Actual Address in an ORG Statement

SYMBOLIC

A symbolic operand is permissible only if the symbol has been previously defined.

In Figure 70 the ORG statement will direct the processor to continue address assignments from the address labeled PHASEONE. Address adjustment is permitted.

NOTE: Neither linkage symbols nor system symbols are permitted in an ORG statement.

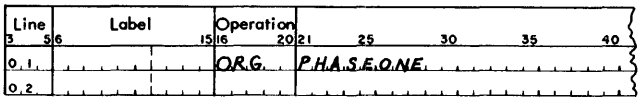


Figure 70. Symbolic Address in an ORG Statement

BLANK

An ORG statement with a blank operand instructs the processor to assign addresses to subsequent entries, beginning at the address that is one greater than the highest address thus far assigned by the processor.

ASTERISK

An asterisk operand can be address adjusted. The ORG statement in Figure 71 instructs the processor to assign storage locations consecutively, beginning 200 locations above the current address.

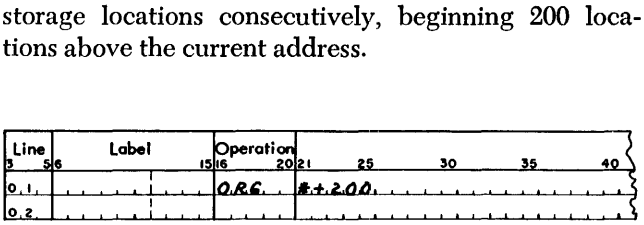


Figure 71. Asterisk Operand with Address Adjustment

ASTERISK + X00

The operand *+X00 instructs the processor to begin address assignment at the next available storage location whose address is a multiple of 100. For example,

in Figure 72, if the last address assigned was location 10926, address assignment would continue at core-storage location 11000.

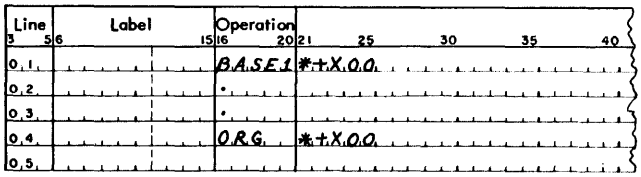


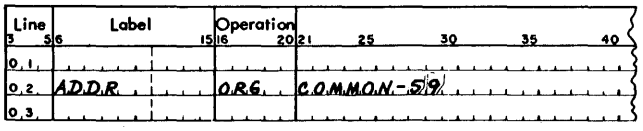
Figure 72. ORG Statement Advancing Address Assignment to Next Multiple of 100

NOTE: Unless the ORG *+X00 card is preceded by a BASE1 *+X00 card, the processor will assign a "W" flag to the ORG statement to warn the programmer that this subprogram must be loaded at an even-hundreds address.

LABELING AN ORG STATEMENT

The ORG statement permits the programmer to break the sequential assignment of a program temporarily, and to return subsequently to that point in the program sequence. This is done by labeling an ORG statement that breaks the sequence. When the programmer wants to return to the original point, an ORG statement with the label in the operand field can be used. The statements after the second ORG statement will be interpreted by the processor as though the sequence had never been interrupted.

The ORG statement, in Figure 73, shows how the programmer can direct the processor to save the address of the last storage allocated. The label ADDR is the symbolic address of the next available location before re-origin occurs. The processor will continue to assign addresses, beginning at the relative address of COMMON – 59.



• Figure 73. Saving the Address of Last Storage Allocated

The programmer can insert another ORG statement later in the source program to direct the processor to begin assigning storage at ADDR. This statement is shown in Figure 74.

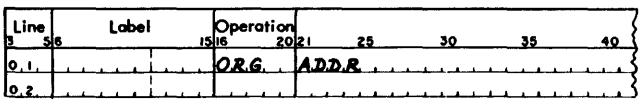


Figure 74. ORG Statement Referencing Last Storage Allocated

LTORG – Literal Origin

LTORG statements are coded in the same way as ORG statements. Their function is to direct the processor to assign storage locations to previously-encountered literals. Storage assignment begins at the address written in the operand field of the LTORG statement.

A LTORG statement can appear anywhere in the source program. If no LTORG statement appears (Figure 75), the processor begins assigning addresses to literals when it encounters an END or SPEND statement.

Figure 75 illustrates one way of directing the processor to assign storage to all literals that have previously appeared within the subprogram segment in which the LTORG statement appears.

NOTE: Since the LTORG statement signals the processor to assign storage to previously-defined literals, the programmer who wishes to use similar literals must re-create them. Thus, the programmer cannot use a previously-defined area defining literal, or its contents, after a LTORG. The area defining literal must be redefined.

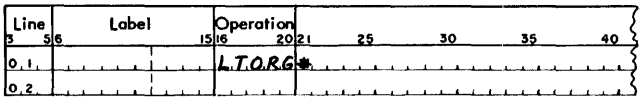


Figure 75. The LTORG Statement

END – End Subprogram and Assembly

The END statement must be the last card in the source program. The END statement directs the processor to start assigning all unassigned literals at this relative address in the subprogram.

If the operand field is blank, END signals the processor that all source program entries have been read. This form of the END statement is used to specify the end of a secondary subprogram.

If the operand field is not blank, it also specifies the end of a primary subprogram and indicates its entry point (Figure 76).

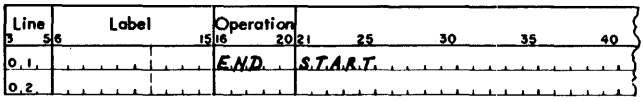


Figure 76. The END Statement

SPEND – End Subprogram

The SPEND statement is used when assembling two or more subprograms with the same symbol table used throughout the assembly. The SPEND statement indicates the end of a subprogram and directs the processor to process all unassigned literals at this point

of the subprogram. The SPEND statement implies that another subprogram to be assembled will follow.

The same formats used for the END statements are used for the SPEND statements. If the SPEND statement is written with an operand, the resultant output will be the same as that produced by an END statement with an operand (primary subprogram). If there is no operand, the output will be exactly the same as for the END statement without an operand (secondary subprogram). (See Figure 77.)

NOTE: The SPEND statement must be followed by the TITLE card of the next subprogram. (HEADR, RESEQ, and EJECT statements can intervene.)

If a source statement that can produce a load card in the object deck appears after a SPEND statement, and before a TITLE statement, the processor places a "W" flag on the first statement of the load card.

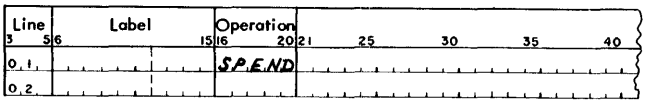


Figure 77. The SPEND Statement

The Macro System

The macro system enables the programmer to extract from a library of macro routines a sequence of instructions tailored by the processor to fit his particular program needs. This sequence of instructions is inserted automatically in the object program. This ability of Autocoder to process macro-instructions relieves the programmer of much repetitive coding.

Definitions of Terms

The special terms used in describing the requirements and characteristics of the macro system are defined below.

Macro-Instructions: A symbolic instruction written in the source program that causes a series of machine-language instructions to be inserted in the object program.

Object Routine: The specific machine-language instructions needed to perform the functions specified by a macro-instruction. An object routine is inserted directly into a program without a linkage or calling sequence. The routine is placed in the object program each time its associated macro-instruction is encountered by the processor.

Model Statement: Model statements appear in the macro library routines. They establish the conditions for inserting parameters in the object routine and define the basic structure of the symbolic program entries. They include pseudo-macro statements and symbolic entries.

Macro Routine: The complete set of model statements from which an object routine is developed by the processor. The form of an object routine depends upon the parameters given in the macro-instruction.

Macro Library: The macro library contains the complete set of macro routines stored on the System Operating File. Each routine has an identifying label.

Librarian: The librarian is that phase of the System Generator that produces and maintains the macro library on the System Operating File. The user should refer to the publication, *System Generation,* Form C28-0352, for information concerning the procedures for the production and maintenance of the macro library.

Parameters: Parameters are the elements in the operand fields of macro-instructions. Parameters can reference literals, actual addresses, or data fields to be inserted in symbolic program instructions generated from the model statements.

Pseudo-Macro Statements: A pseudo-macro statement appears only within the macro library. It is used internally by the processor to control the production of a series of object program instructions.

Macro Operations

The entries that will subsequently appear in the object program are placed on the library tape at system generation time. The function of the macro-instruction is to direct the processor in selecting the specific entries desired by the programmer. The entries selected become a routine designed to perform a specific function.

To illustrate the basic operation of the macro system, a hypothetical macro called CHECK, with a simple library routine, is used. The routine is designed to compare the contents of an input area to the contents of another area, test the compare indicator for a high, equal, or low condition, or any combination of the three.

Figure 78 shows the Library Coding Form used with the 1410/7010 Macro System.

Figure 79 shows the following:

1. The entries on the Library Coding Form.
2. The macro-instruction that specifies to the processor that all the instructions in the library routine are required and must appear in the object program.
3. The symbolic program entries generated by the processor. (The processor will subsequently translate these symbolic entries into machine language and insert them in the proper juncture of the object deck.)

General Description: Model statements are used to describe all entries in a macro routine. They include pseudo-macro as well as symbolic program statements.

The Programmer: The programmer plans and codes the following:

1. Designs a general routine to perform specific functions (depending upon the parameters supplied) when it is executed in the object program.
2. Writes the model statement as follows:
   a. If the entry is complete (no substitution), it is written on the library coding sheet as if it were an entry in a source program. This entry will be included in all object routines unless a bypass condition exists (see "BOOL"). This is illustrated in Figure 80.
   b. If the entry is incomplete, the programmer writes a special four-character code to indicate

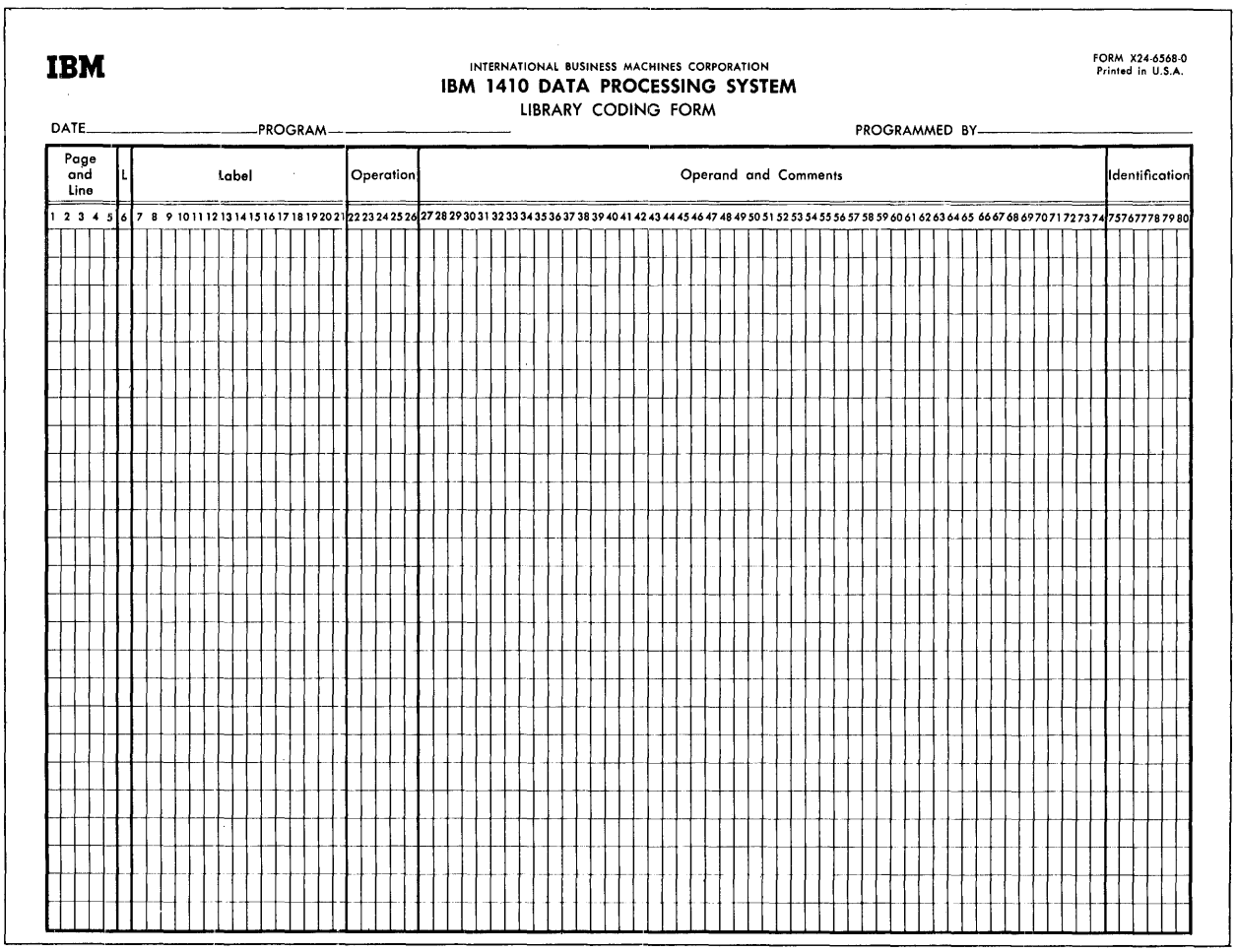


Figure 78. IBM 1410/7010 Library Coding Form

Library Entry

Page and Line	L	Label	Operation	Operand
			C	□001,□002
			BH	□00C
			BE	□00D
			BL	□00E

Macro-Instruction

Line	Label	Operation	OPE
0.1	ABCD	CHECK	PAR1,PAR2,PAR3,PAR4,PAR5
0.2			

Assembled Symbolic Program Entry

ABCD	C	PAR1,PAR2
	BH	PAR3
	BE	PAR4
	BL	PAR5

Figure 79. Macro Operations

Page and Line	L	Label	Operation	
		LABEL	BCE	A,B,C

Figure 80. Model Statement for a Complete Instruction

that a corresponding parameter from the macro-instruction operand field must be inserted in its place. This code is a □ followed by a number from 001 to 199, that indicates the position of the parameter in the macro-instruction. The macro-instruction in the source program will give the parameter entries to be inserted in the object routine. The model statement is illustrated in Figure 81.

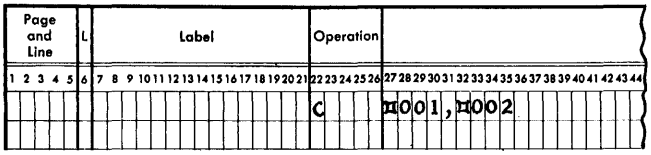


Figure 81. Model Statement for an Incomplete Instruction with Required Parameters

c. If the entry is incomplete, the programmer writes a □ followed by a number from 001 to 199 with AB bits over the units position (parameter 001 is □00A, parameter 2 is □00B, etc.). This indicates that the entry is to be included in the object routine only if the parameter is specified by the macro-instruction. For example, if parameter 003 does not appear in the macro-instruction, the instruction shown in Figure 82 will be deleted from the object routine.

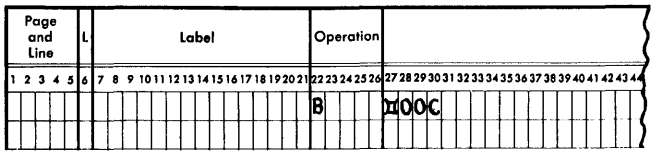


Figure 82. Model Statement for an Incomplete Instruction with Conditional Parameters

Labeling: If the model statement represents an instruction entry point for a branch instruction elsewhere in the program, it should have a label.

If additional external labels are required and specified as parameters in the macro-instruction they can be inserted in the label field of the symbolic program entry by using the □001-199 code.

The label of the macro-instruction causes the generation of an equate statement in the assembled object routine. The label is equated to an *, as shown in Figure 83.

Another example is shown in Figure 84.

Symbolic Addressing within the Library Routine: To allow a symbolic reference to other instructions in a library routine a □ followed by a number from 001 to 199 with a B bit over the units position (□00J = symbolic address 1, □00K = symbolic address 2, etc.) can be used. For example, the processor generates the symbolic address if the code □00J is used as a label for one entry and as an operand of at least one other entry in the same library routine.

Internal labels within flexible routines are generated in the form □nnnmmm, where nnn is the code (00J-09R), and mmm is the number of the macro within the source program. This is done to avoid duplicate address assignments for labels.

Example: Use the generated symbolic address of □00J as an operand for entry 3 and as the label for entry 6. UPDAT is the 23d macro encountered in the source program (Figure 85).

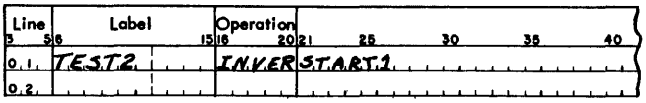
Address Adjustment and Indexing: The parameters in a macro-instruction and the operands in partially complete instructions in a library routine can have address adjustment and indexing.

If address adjustment is used in both the parameter and the instruction, the assembled instruction will be adjusted to the algebraic sum of the two. For example, if the address adjustment on one is +7 and the other is −4, the assembled instruction will have address adjustment equal to +3.

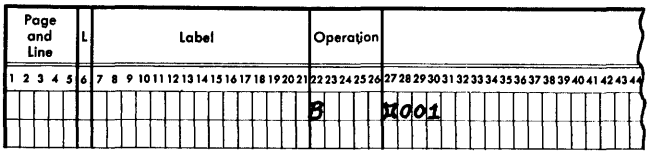
Model statement operands can be indexed. This indexing takes precedence over any indexing of a parameter supplied by a macro-instruction. The model statement index is used.

Literals: Operands of instructions in library routines may use literals as required. However, these literals may not contain the @ symbol within an alphameric literal.

Macro Instruction (Source Program)



Model Statement

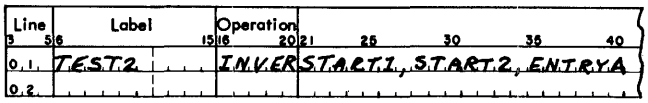


Assembled Symbolic Program Entry

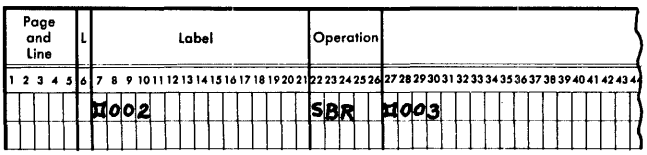
```
TEST2      EQU     *
           B       START1
```

Figure 83. Labeling

Macro Instruction (Source Program)



Model Statement

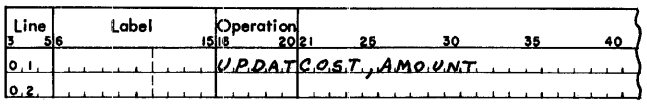


Assembled Symbolic Program Entry

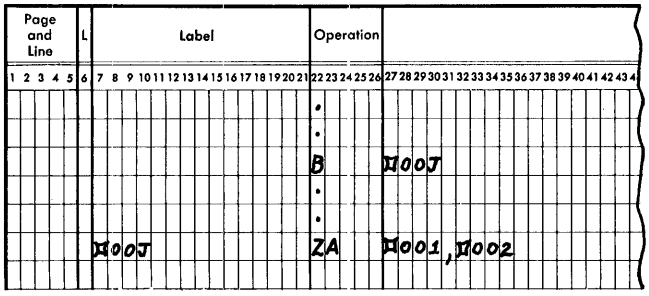
```
TEST2      EQU     *
START2     SBR     ENTRYA
```

Figure 84. Additional External Labels

Macro Instruction (Source Program)



Model Statement



Assembled Symbolic Program Entry

```
                .
                .
                B       □00J023
                .
                .
□00J023         ZA      COST,AMOUNT
```

Figure 85. Internal Labels

NOTE 1: A model statement in the library routine for a macro-instruction may not be another macro-instruction.

NOTE 2: END statements cannot be used in library routines.

The Processor enters model statements in the library tape immediately following the header statement during System Generation.

Result: Any library routine can be extracted by writing the associated macro-instruction in the source program.

Figure 86 is a summary of the codes that can be used in the model statements of library routines.

CODE	POSITION	FUNCTION
□001 - □199	Statement	Substitute parameter (parameter must be present)
□00A - □19I	Statement	Substitute parameter (if parameter is missing, delete statement)
□00J - □19R	Label Field and Operand Field	Assign internal label

Figure 86. Model Statement Codes

General Description: A macro-instruction is the entry in the source program that causes a series of instructions to be inserted in a program.

The Programmer:

1. Writes the name of the library routine in the operation field.

2. Writes the label that is to reference the first assembled model statement. A LABEL EQU * is generated to do this.

3. Writes the parameters that are required for the particular object routine desired. These parameters, used by the model statements, are written as follows:

   a. Parameters must be written in the sequence in which they are to be used by the codes in the model statements. For example, if cost is parameter 001, it must be written first so that it will be substituted wherever a □001 or □00A appears as a label, operation code, or operand of a model statement.

   b. As many parameters may be used as can be contained in the operand fields of five or fewer coding sheet lines. If more than one line is needed for a macro-instruction, the label and operation fields of the additional lines must be left blank. Parameters must be separated by a comma. They cannot contain blanks or commas unless they appear between @ symbols. The @ symbol itself cannot appear between @ symbols. Also, the @ symbol can be used only in pairs as a literal identifier. It cannot be used in any other way; e.g., a single @ symbol could not be used to represent the d modifier of a macro-instruction. If parameters for a single macro-instruction require more than one coding sheet line, the last parameter in each line must be followed immediately by a comma. The last parameter in a macro-instruction should not be followed by a comma.

   c. Parameters that are not required for the particular object routine desired can be omitted from the operand field of the macro-instruction. However, if a parameter is omitted, the comma that would have followed the parameter must be included, unless the omitted parameter is behind the last parameter which is included in the macro-instruction. These commas are necessary to count parameters up to the last included parameter. All parameters between the last included parameter and parameter 199 are assumed by the processor to be absent.

Figures 87, 88, 89 and 90 show how parameters can be omitted. The hypothetical macro-instruction called EXACT is used. EXACT can have as many as nine parameters.

The Processor extracts the library routine and selects the model statements required for the object routine as specified by the parameters in the macro-instructions, and by substitution and switches set by BOOL or COMP in the model statements.

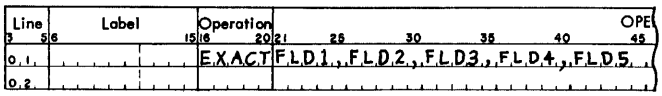


Figure 87. Parameter for EXACT. 006-199 Missing

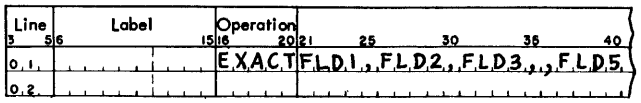


Figure 88. Parameters 004 and 006-199 Missing

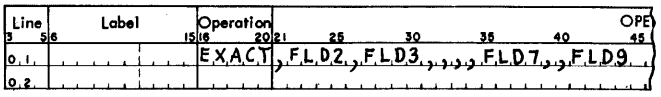


Figure 89. Parameters 001, 004-006, 008 and 010-199 Missing

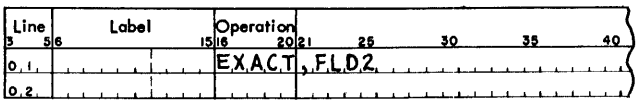


Figure 90. Parameters 001 and 003-199 Missing

Pseudo-Macro Instructions

These statements never appear in a user's source program or in the output listing of an assembled Autocoder program. However, they are used in library routines to signal the processor that certain conditions exist which can affect the assembly of an object routine. For example, the presence of a pseudo-macro-instruction in a library routine can cause a group of model statements to be deleted. Thus, pseudo-macros provide the writer of library routines with a coding flexibility which exceeds the limitations of the substitution and condition codes described previously.

Pseudo-macro-instructions may be written anywhere in a library routine. The five pseudo-macros incorporated in the Autocoder processor are MATH, BOOL, COMP, NOTE, and MEND.

Permanent and Temporary Switches

The MATH, BOOL, and COMP pseudo-macros use internal indicators (switches) to signal the processor of existing status conditions.

There are 099 permanent and 199 temporary switches available for recording status conditions. Each switch occupies one core-storage position during the macro generator phase of Autocoder. If a storage position contains the character A (AB1 bits), the switch is ON; if it contains a ? (CAB 82 bits), the switch is OFF. At the beginning of assembly all switches are OFF.

Permanent Switches: Permanent switches retain status conditions during the entire macro generator phase unless changed by a pseudo-macro. They are addressed by using a # symbol followed by the three-digit number of the switch to be set or tested. For example, #001 addresses permanent switch 001; #002 addresses switch 002; and #099 addresses switch 099.

Temporary Switches: When the processor encounters a macro-instruction, the temporary switches are set to the condition (presence or absence) of the parameters in the operand of the macro field. If the parameter is present, the corresponding switch is set ON. If the parameter is missing, the switch is set OFF. For example, if parameter 001 is present, temporary switch 001 is turned ON. If parameter 002 is missing from the macro-instruction, temporary switch 002 is OFF. Temporary switches retain status throughout the processing of a macro-instruction unless changed by a pseudo-macro. After the macro-instruction has been completely processed, all temporary switches are set OFF. Temporary switches are addressed by using a □ symbol followed by the three-digit number of the switch to be set or tested. For example, □001 addresses temporary switch 001; □002 addresses switch 002; and □199 addresses switch 199.

If a macro with a maximum of nine parameters is encountered, the processor sets the first nine temporary switches to indicate the presence or absence of these nine parameters. Temporary switches 010-199, which are OFF, can be used by the pseudo-macros to communicate conditions to the processor while it is working on this particular macro-instruction. This use of temporary switches is recommended because it reserves the permanent switches for communicating information from one macro to another.

MATH — For Solving Algebraic Expressions

A MATH pseudo-macro contains as operands: sum boxes, arithmetic expressions, and sign switches.

Sum Boxes: A sum box is a group of five core-storage positions used to store the result of an arithmetic expression. Autocoder makes available 20 such sum boxes. A sum box is addressed by using a # symbol followed by the three-digit number (ending in zero or five) of the sum box to be referenced. For example, the address of the first sum box is #005; the address of the second sum box is #010; and the address of the twentieth sum box is #000.

At the beginning of the macro phase, a sum box contains 00000. Any number may be placed in a sum

box or added to its contents. The units position of the sum box always contains the sign of the result. Sum boxes retain information placed in them throughout the macro phase, and their contents may be used and/or changed from one macro-instruction to another.

Sum boxes can be used by model statements as well as by a pseudo-macro. For example, in Figure 91, assume that sum box #005 contains 12345 and sum box #010 contains 00015.

NOTE: ZA FLD1+0001N,FLD2 is processed as ZA FLD1−15,FLD2.

Macro Instruction

Line	Label	Operation	
01		REORG	FLD1,FLD2
02			

Model Statement

Page and Line	L	Label	Operation	
			ORG	#005
			ZA	¤001+#010,¤002

Assembled Symbolic Program Entry

```
ORG     1234E
ZA      FLD1+0001N,FLD2
```

Figure 91. Sum Boxes

Arithmetic Expressions: Arithmetic expressions within the MATH pseudo-macro use add (+), subtract (−), multiply (*), and divide (/). An @ symbol represents both the left and right parentheses if they are required for the expression. For example,
(001+12−5) 20 is written: @001+12−5@*20.

Multiplication and division are done before addition and subtraction by the MATH pseudo-macro, unless otherwise indicated by the use of @s. The quotient resulting from the divide operation is *not* half-adjusted, and the remainder is lost. At the end of a multiplication operation the five low-order positions of the product are used for the result. (The high-order digits are lost.) An overflow is ignored.

The result of the arithmetic expression is inserted with its sign in the designated sum box.

Sign Switches: Permanent and temporary switches may be used to store the sign of the result of an arithmetic expression. The first switch specified in the operand field of the pseudo-macro represents a positive result; the second represents a zero result; and the third represents a negative result. Consequently, one switch is on and the other two are off if the result is either positive or negative. A zero result causes both the zero and positive switches to be set ON. It is not necessary to specify all three switches. However, if a switch code is omitted from the operand field, the comma that would have followed the switch code must be present. (This is the same rule that applies to omitted parameters in a macro-instruction.)

The Programmer:

1. Writes the name of the pseudo-macro (MATH) in the operation field.
2. Writes in the operand field:
   a. The code for the sum box in which the result of the arithmetic expression is to be stored.
   b. The arithmetic expression.
   c. The code for the switch in which the sign(s) of the result are to be stored.

NOTE: A comma must follow the sum box code, the arithmetic expression, and the individual sign-switch codes. Figure 92 shows the format for a MATH pseudo-macro.

The Processor:

1. Produces the result of the arithmetic expression.
2. Stores the result in the sum box.
3. Sets the sign switches.

Example: The MATH pseudo-macro shown in Figure 93 multiplies parameter 07 by 401 and adds 12 to the

Page and Line	L	Label	Operation	Operand and Comments	Identification
			MATH	SUMBOX,ARITHMETIC EXPRESSION,PLUS,ZERO,MINUS	

Figure 92. Format for the MATH Pseudo-Macro

Page and Line	L	Label	Operation	Operand and Comments	Identification
			MATH	#030,12+¤007*401,#004,#006,#009	

Figure 93. MATH Pseudo-Macro

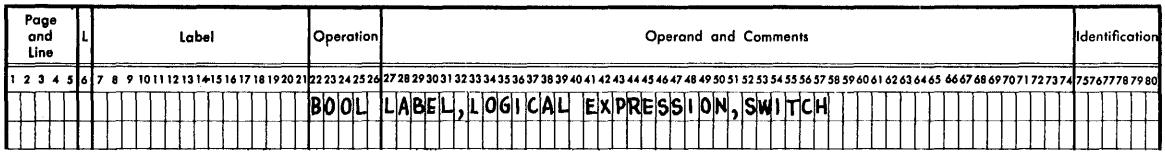


Figure 94. Format for the BOOL Pseudo-Macro

result. The answer is stored in SUMBOX 6 (#30). If the result is positive, permanent switch 04 is set ON; if the result is zero, switches 04 and 06 are set ON; if the result is negative, switch 09 is set ON.

BOOL — For Solving Logical Expressions

General Description: The BOOL pseudo-macro can be used to set a permanent or temporary switch as the result of a logical expression, or to cause the processor to skip over certain model statements if the logical expression is false. If the statement is true, the processor goes to the next sequential model statement.

The Programmer:

1. Writes the name of the pseudo-macro (BOOL) in the operation field.

2. May write a special one-character label, the logical expression (statement), and a switch code in the operand field in the format shown in Figure 94.

Labeling: A special one-character label permits skipping *forward* in the library routine as the object routine is being assembled by the processor. This one-character label is written in the first position of the operand field of the BOOL pseudo-macro and also in the label position (column 6 of the library coding form) of the first model statement (or command) to be examined after the skip has been initiated. Skipping occurs only if the logical statement is false. The label may be omitted if a skip is not desired, but the comma that would have followed the label must be written in the BOOL statement to indicate that the label is missing. The label can be any alphabetic or numeric character. Special characters are not permitted.

Logical Expression: The BOOL pseudo-macro can have any combination of three logical operations: * (and), + (or), and − (not). The operators are defined in Figure 95. The combination of these operators and the switches to be tested make up the logical expression. (See, for example, Figure 96.)

The @ symbol is used to represent both the left and right parentheses.

*	+	-
1 * 1 = 1 1 * 0 = 0 0 * 1 = 0 0 * 0 = 0	1 + 1 = 1 1 + 0 = 1 0 + 1 = 1 0 + 0 = 0	- 1 = 0 - 0 = 1

Figure 95. Table of Operators

Switches: Either a permanent or temporary switch may be used to store the result of the logical expression. If the expression is true, the specified switch will be set ON. If the expression is false, the specified switch is set OFF. If no switch setting is desired, a comma must be used to indicate that the switch is missing.

The Processor:

1. Examines the status switches to determine whether all conditions specified in the logical expression are satisfied. If they are, the expression is true. If the logical condition is not met, the expression is false.

2. Sets the specified status switch to ON or OFF to reflect the true or false condition.

3. If a false condition exists and a label appears in the BOOL operand, the processor skips forward to the command or model statement containing a corresponding label in its label position.

To determine if a logical expression is true or false:

a. Consider all ON conditions true and all OFF conditions false.

b. Let 1 = true and 0 = false.

c. Calculate the logical value of the expression.

If the logical value of the expression is zero, the expression is false. If the logical value is one, the expression is true. For example, if switches 001, 002, 003 and 004 are on, the expression

@□001*□002@+@□003*□004@

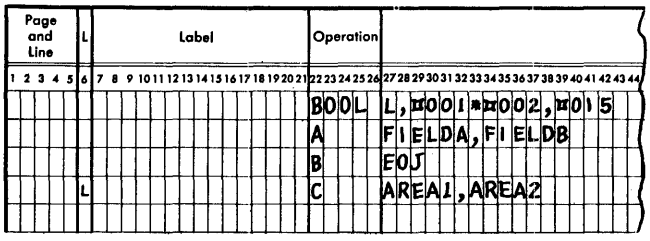
is true because:

(ON*ON)+(ON*ON) =
(1*1)+(1*1) =
1+1 = 1

Examples: Figure 96 shows how the BOOL pseudo-macro can be used. The BOOL entry states:

1. If temporary switches 001 and 002 are ON, the statement is true. Therefore, set temporary switch 015 ON.

2. However, if either temporary switch 001 or 002



• Figure 96. Using the BOOL Pseudo-Macro

is OFF, the statement is false. Therefore, set temporary switch 015 OFF and skip to statement 004.

The example shown in Figure 97 states:

1. If both temporary switches 001 and 002 or both temporary switches 003 and 004 are ON, the statement is true. Therefore, set temporary switch 015 ON.
2. However, if either temporary switch 001 or 002 and either temporary switch 003 or 004 is OFF, the statement is false. Therefore, set temporary switch 015 OFF and skip to the model statement whose label is L.

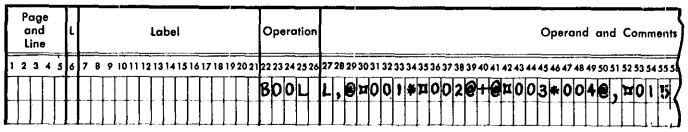


Figure 97. BOOL Pseudo-Macro

Figure 98 is a table showing all conditions that will cause the BOOL statement shown in Figure 97 to be true.

Figure 99 is a table showing all conditions that will cause the BOOL statement shown in Figure 97 to be false.

SWITCHES

CONDITIONS	001	*	002	+	003	*	004	=	LOGICAL VALUE
	OFF 0	*	OFF 0	+	OFF 0	*	OFF 0	=	0
	ON 1	*	OFF 0	+	OFF 0	*	OFF 0	=	0
	OFF 0	*	ON 1	+	OFF 0	*	OFF 0	=	0
	OFF 0	*	OFF 0	+	ON 1	*	OFF 0	=	0
	OFF 0	*	OFF 0	+	OFF 0	*	ON 1	=	0
	OFF 0	*	ON 1	+	OFF 0	*	ON 1	=	0
	ON 1	*	OFF 0	+	ON 1	*	OFF 0	=	0
	OFF 0	*	ON 1	+	ON 1	*	OFF 0	=	0
	ON 1	*	OFF 0	+	OFF 0	*	ON 1	=	0

FALSE

Figure 99. False Conditions

COMP — To Compare Two Fields

General Description: The COMP pseudo-macro compares an A-field to a B-field (maximum of 15 characters), and sets permanent or temporary switches to indicate the result of the comparison.

The Programmer:

1. Writes the name of the pseudo-macro (COMP) in the operation field.

SWITCHES

CONDITIONS	001	*	002	+	003	*	004	=	LOGICAL VALUE
	ON 1	*	ON 1	+	OFF 0	*	OFF 0	=	1
	OFF 0	*	OFF 0	+	ON 1	*	ON 1	=	1
	ON 1	*	ON 1	+	ON 1	*	ON 1	=	1
	ON 1	*	ON 1	+	ON 1	*	OFF 0	=	1
	OFF 0	*	ON 1	+	ON 1	*	ON 1	=	1
	ON 1	*	ON 1	+	OFF 0	*	ON 1	=	1
	ON 1	*	OFF 0	+	ON 1	*	ON 1	=	1

TRUE

Figure 98. True Conditions

2. Writes the operand field in the format shown in Figure 100. The first and second entries are the A- and B-fields. The A- and B-fields may be any of the parameters 001-199, sum boxes #005-#000, or literals. They cannot be switches.

NOTE 1: For the COMP pseudo-macro, alphameric literals are not enclosed by @ symbols. Entries 3, 4, and 5 are high, equal, and low switches.

NOTE 2: The codes for the two fields to be compared must be present in all COMP pseudo-macro-instructions. Codes for the switches may be omitted if they are not needed to store the result of the compare operation. However, if a switch is omitted, the comma that would have followed it must be included in the operand field.

NOTE 3: B-field controls compare. (High-order position of B-field ends compare.)

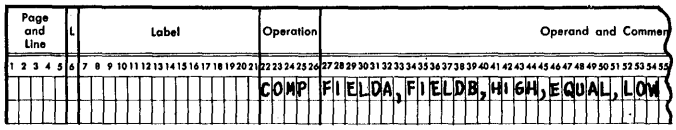


Figure 100. Format for COMP Pseudo-Macro

The Processor:

1. Compares the A-field to the B-field.
2. Sets one status switch ON and two switches OFF to reflect the result of the comparison.
   a. The first switch is set ON, if the value of the B-field is greater than that of the A-field.

b. The second switch is set ON, if the B-field is equal to the A-field.

c. The third switch is set ON, if the value of the B-field is less than that of the A-field.

Examples: Figure 101 shows a COMP pseudo-macro which states:

1. Compare parameter 002 of the macro statement to WORKAREA.
2. If parameter 002 is equal to WORKAREA, turn on temporary switch 25.
3. If WORKAREA is less than parameter 002, turn on temporary switch 26.

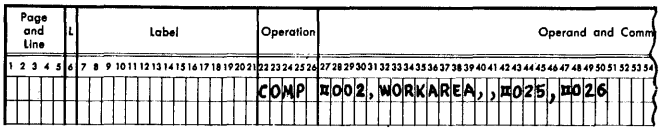


Figure 101. COMP Pseudo-Macro

Figure 102 shows a COMP pseudo-macro which states:

1. Compare the contents of sum box 005 to parameter 003 of the macro statement.
2. If the result is HIGH, set temporary switch 024 ON.
3. If the result is EQUAL, set temporary switch 025 ON.
4. If the result is LOW, set temporary switch 026 ON.

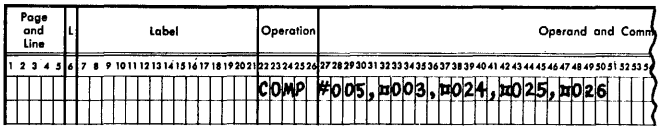


Figure 102. Comparing a Parameter to the Contents of a Sum Box

NOTE: Standard 1410/7010 collating sequence determines HIGH, EQUAL, or LOW conditions. Comparisons are controlled by the B-field. Thus, the statement shown in Figure 103 will cause temporary switch 025 to be set ON if the low-order position of parameter 002 is a 3.

NOTE — To Produce a Message

General Description: The NOTE pseudo-macro is used to write messages concerning conditions that can arise during the processing of a macro-instruction.

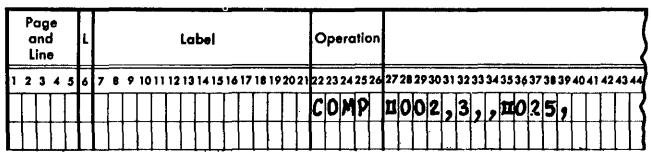


Figure 103. Checking for a Single Character

The message is printed in line on the output device (tape or on-line printer). In addition, an "N" will be automatically inserted in the flag field of the assembly listing.

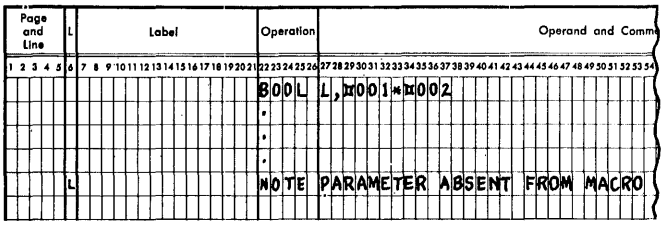
The Programmer:

1. Writes the name of the pseudo-macro (NOTE) in the operation field.
2. Writes the message in the operand field.

The Processor: Prints the message on the Standard Print Unit (tape or on-line printer).

Examples: Figure 104 shows how the NOTE pseudo-macro can be used in combination with the BOOL pseudo-macro. The BOOL pseudo-macro tests to ensure that parameters 001 and 002 are present in the macro-instruction. If either parameter is missing, the processor skips to the NOTE pseudo-macro and prints:

PARAMETER ABSENT FROM MACRO.



• Figure 104. NOTE Pseudo-Macro

MEND — End of Routine

General Description: This pseudo-macro signals the end of generation for a macro-instruction. It may appear anywhere in a library routine.

The Programmer:

1. Writes the name of the pseudo-macro (MEND) in the operation field.
2. Leaves the operand field blank.

The Processor: Stops processing the macro-instruction when it encounters a MEND statement. Figure 105 shows a MEND pseudo-macro.

NOTE: A BOOL pseudo-macro can be used to skip over a MEND pseudo-macro which appears within the library routine if conditions indicate that more model statements must be processed.

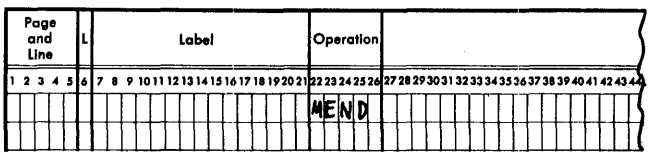


Figure 105. MEND Pseudo-Macro

Pseudo-Macro Coding Example

Example: Figure 106 shows the library entry for a hypothetical macro called PRLIT. This library routine uses all of the five pseudo-macros. It illustrates the effect of the pseudo-macros on the processing of a macro-instruction. The meaning of each line in the library routine is:

Entry 1: If parameter 001 is present, set temporary switch 050 OFF and go to entry 3. If parameter 001 is missing, go to entry 2.

Entry 2: Print the note: OPERAND 001 ABSENT.

Page and Line	L	Label	Operation	
01001			BOOL	A,-□001,□050
01002			NOTE	OPERAND 001 ABSENT
01003	A		BOOL	A,#010
01004			ORG	#005
01005	A		MATH	#005,#005+100,,,
01006			SBR	□00K+5
01007			MLCA	@00000@,□003
01008			A	+3,□003
01009			B	□004
01010			COMP	□002,@,,□051,
01011			BOOL	B,□051,
01012			MLCA	□002,□001
01013			S	□002,□00F
01014			MLC	□003,□005
01015		□00K	B	0
01016			BOOL	C,-□051,
01017			MLCA	@□002@,□001
01018	C		MEND	

• Figure 106. PRLIT Library Routine

Entry 3: If permanent switch 010 is OFF, go to entry 5. If permanent switch 010 is ON, take entry 4.

Entry 4: ORG at the contents of sum box #005.

Entry 5: Put the contents of sum box #005 plus 100 in sum box #005.

Entry 6: Store the contents of the B-address register in an address equal to the address assigned to the internal label (□00K) +5.

Entry 7: Move five zeros to the field whose symbolic address is parameter 003 of the macro-instruction.

Entry 8: Add the literal +3 to the field specified by the parameter 003.

Entry 9: Branch to parameter 004.

Entry 10: If parameter 002 is a literal, the EQUAL switch (□051) is set ON.

Entry 11: If the EQUAL switch (temporary switch 51) is OFF, skip to entry 15. If the EQUAL switch is ON go to entry 12.

Entry 12: Move parameter 002 to parameter 001.

Entry 13: Subtract parameter 002 from parameter 006. (If parameter 006 is missing, this statement will be bypassed.)

Entry 14: Move parameter 003 to parameter 005.

Entry 15: Branch to 0

Entry 16: If temporary switch 051 is ON, skip to entry 18. If temporary switch 051 is OFF, go to entry 17.

Entry 17: Insert parameter 002 as a literal, and move it to the field indicated by parameter 001.

Entry 18: End of library routine.

Assume that:

1. The macro shown in Figure 107 is encountered in the source program.
2. Permanent switch 010 is ON.
3. Sum box #005 contains 12345.

Macro Instruction

Line	Label	Operation	OPERAND
0.1		PRLIT	AREAA,@42AB@,FIELD1,EXIT1,WORKAREA
0.2			

Assembled Symbolic Program Entry

```
        ORG     12345
        SBR     □ 00K023+5
        MLCA    @00000@, FIELD1
        A       +3, FIELD1
        B       EXIT1
        MLCA    @42AB@,AREAA
        MLC     FIELD1, WORKAREA
        B       0
```

• Figure 107. Using the PRLIT Routine

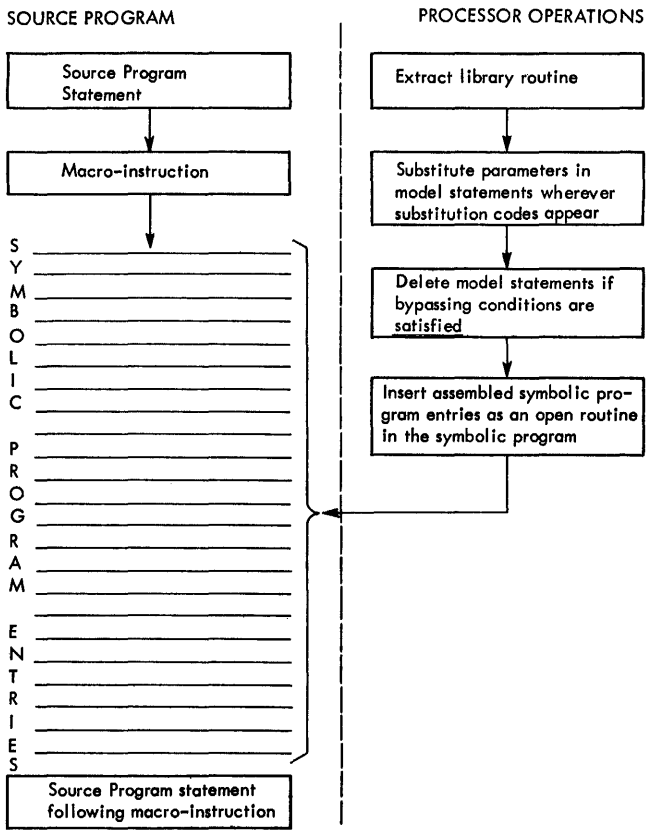


Figure 108. Macro Processing

Appendix A: Processor Error Diagnostic Procedures

The following chart lists the seven flag codes which the Autocoder processor will affix to an Autocoder statement upon diagnosing an error. The circumstances which will produce each flag are also described.

FLAG	DEFINITION	CIRCUMSTANCES
F	Format Invalid	The operand of each statement is analyzed for correct format. Invalid format testing is not 100% exhaustive. The list included below indicates the areas in which format testing is conducted. (See NOTE 2.)
M	Multiple Definition	A label (or labels) in the operand has been defined more than once in the source program. The address of the first encountered label is used. NOTE 1: See indexing restriction of X14 and X15.
N	Note Generated	This flag is caused by a macro generation and is not necessarily a programming error. (See "Note – To Produce a Message" under "The Macro System.")
O	Operation Invalid	The operation code of the statement is a mnemonic that does not exist in the Autocoder language and is not one of the macro-instructions contained in the macro library. (See "Mnemonic Operation Codes," Figure 21.)
R	Restricted Operation	The operation code of the statement is one of the restricted operation codes. (See Appendix D.) The "R" flag does not set the "no-go" switch.
U	Undefined Label	A label (or labels) in the operand has not been defined by another statement in the source program. (See NOTE 2.)
W	Warning	This flag is used to indicate improper program conditions. For example, the processor must encounter a TITLE card after a SPEND card before it generates another card in the object deck. Another example is: an ORG * +X00 must be preceded by a BASE1 * +X00.

NOTE 1: If a label is defined as an area defining literal and as a label in another Autocoder statement, this error will not receive the "M" flag. The cross reference listing, however, will diagnose this condition and give notice of the multiple definition.

NOTE 2: Certain labels are exempt from these requirements. They are linkage symbols, system symbols, index register symbols (X1-15, +X0) and common.

Invalid Operand Relationships Causing the "F" Flag:

1. No operand information in a statement that requires additional information.

2. Comments not separated from operand elements by at least two blanks.

3. Incorrect DA or DAV header configuration.

4. A label used as an address adjustment is not equated to an index register, and the statement is not a DC or DCW entry.

5. Invalid sequence of sensitive characters. Sensitive characters in the operand are:

BLANK	- b
COMMA	- ,
PLUS	- +
MINUS	- –
At Symbol	- @
Asterisk	- *
Pound Sign	- #

6. Incorrect information following a sensitive character, such as
+b or *A.

7. A pound sign not followed by a pure numeric count less than or equal to 500.

8. A literal without a literal character, for example, @@.

Appendix B: Autocoder Messages and Limits

This appendix contains a listing of all the messages produced by Autocoder, a description of the circumstances causing the message, and a table of the limits of the Autocoder processor.

The processor can produce the following messages on the console printer and on the /SPR/ file. When the message is found necessary, the GO mode will be cancelled and the assembly terminated at the point where the condition occurred. If the job is not in the TEST mode, this will cause the System Monitor to cancel the remainder of the job.

10701 AUTOCODER NOT COMPLETED. INCORRECTABLE I/O ERROR (ON MWn) TERMINATED IN PHASE X

This message will have been preceded by the standard IOCS message on the console printer indicating a data check or wrong-length record. In some cases the work file number is given.

10701 AUTOCODER NOT COMPLETED. UNUSUAL (OR UNEXPECTED) END OF FILE ON MW1 TERMINATED IN PHASE X

(A similar message may be produced which indicates MW2 and MW3.) This message will be produced on an unexpected end-of-file or end-of-reel indication when the system is using tape work files, or an end of the tracks assigned to the given 1301 disk work file.

10701 AUTOCODER NOT COMPLETED. LIBRARY DIRECTORY NOT FOUND TERMINATED IN PHASE 1

This message will be produced when the library directory is not contained in the SOF.

10701 AUTOCODER NOT COMPLETED. MACRO LIBRARY NOT FOUND TERMINATED IN PHASE 1

This message will be produced when the macro library is not contained in the SOF.

10701 AUTOCODER NOT COMPLETED. RECURSION CAPACITY EXCEEDED TERMINATED IN PHASE 3

This message will be produced when the maximum number of literals is exceeded in a source program (see chart below). The source program will have to be altered before assembling.

10701 AUTOCODER NOT COMPLETED. MAXIMUM NUMBER OF DTFS EXCEEDED TERMINATED IN PHASE 1

This message will be produced when the maximum number of DTF's has been exceeded in a source program (see chart below). The source program will have to be altered before assembling.

10701 AUTOCODER NOT COMPLETED. ERROR IN MACRO LIBRARY READ TERMINATED IN PHASE 2

This message will be produced when an error is read in the Macro Library.

10701 AUTOCODER NOT COMPLETED. MACRO XXXXX NOT FOUND ON DISK TERMINATED IN PHASE 2

This message will be produced when the specified Macro Library routine, xxxxx, is not found on the disk file.

10702 NOGO SWITCH SET

This message is produced when a source statement is flagged (except for the "R" flag).

AUTOCODER LIMITS

Affecting	Maximum	Reason	Results when Exceeded
Macro-Instructions	240 different macros in the macro library	Table size is limited	Diagnostic message and System Generation termination
	9999 macro-instruction usages within one assembly run	Limit of generated labels	Possibility of multiple label definitions
	25 DTF usages in one assembly run	Table size is limited	Diagnostic message and assembly termination
Autocoder Statements	The total number of Autocoder statements (source and generated) is limited by one reel of tape, or by the number of disk cylinders assigned to each work file.	No multi-reel processing on tape, or end of tracks assigned to a work file on disk	Diagnostic message and assembly termination
Literals	Approximate literal limit $= \frac{x - y - z}{16}$ Where: x = contents of /AMS/ y = address of IBAU30SUBR defined in the System Generation memory map. z = 3,000 plus 16 times the number of EQU, ORG, and LTORG statements containing symbolic operands within each assembly.	Work area in core storage of the SOF has been exceeded	Diagnostic message and assembly termination

Appendix C: 1410/7010 Autocoder Sample Program

The following sample program is provided both as a test deck for the user's System Operating File and as a teaching aid. It is an Autocoder source program with a set of dummy input which is combined with control cards to constitute a Monitor job run. The card deck (Exhibit I) is loaded via the Standard Input Unit. The job run includes the assembly of the source program by the Autocoder processor, the creation of the absolute program by the Linkage Loader, and the execution of the sample program which results in output on the Standard Punch Unit.

The assembled object program will perform the following functions:

1. Reproduce card decks in card columns 6-75.
2. Sequence the deck, punching the sequence number into columns 1-5.
3. Gang punch the contents of columns 76-80 of the first card into the corresponding columns of the remainder of the deck.
4. Type the message "EOJ" on the console printer at end of job.

The exhibits are as follows:

Exhibit I	The assemble-and-go input deck.
Exhibit II	The console printer output during the run.
Exhibit III	A listing of the object program deck.
Exhibit IV	The Standard Print Unit output of the job run, showing: A. Monitor Control Cards B. Assembly Listing C. Cross Reference Listing D. Linkage Loader Control Cards
Exhibit V	A listing of the cards which are the output of the executed sample program.

```
     MON$$      DATE 64015
     MON$$      JOB  SAMPLE
     MON$$      ASGN MJB,A1
     MON$$      MODE GO
     MON$$      ASGN MGO,A6
     MON$$      EXEQ AUTOCODER
0001            HEADRSAMPLE PROGRAM USING 1410/7010 AUTOCODER                SAMPL
0002            TITLERESEQUENCE                                              SAMPL
0003            BASE1*&X00                                                   SAMPL
0004 *                                                                       SAMPL
0005 *      THIS  PROGRAM  WILL  RESEQUENCE  SOURCE  DECKS                   SAMPL
0006 *      RENUMBERING  FROM  0001  IN  THE  FIRST  FOUR                    SAMPL
0007 *      COLUMNS  AND  WITH  THE  IDENTIFICATION  FIELD                   SAMPL
0008 *      SUPPLIED BY THE FIRST CARD GANG-PUNCHED INTO EACH                SAMPL
0009 *      CARD PRODUCED.                                                   SAMPL
0010 *                                                                       SAMPL
0011 START      B    READ                                                    SAMPL
0012            MLCB AR80,IDENT#5                                            SAMPL
0013            B    *&8                                                     SAMPL
0014 LOOP       B    READ                                                    SAMPL
0015            B    CHNG                                                    SAMPL
0016            B    PUNCH                                                   SAMPL
0017            B    LOOP                                                    SAMPL
0018 *                                                                       SAMPL
0019 READ       SBR  RDXT&5                                                  SAMPL
0020            STDIOREAD,AREA,EOF,ERROR,M                                   SAMPL
0021 RDXT       B    0                                                       SAMPL
0022 *                                                                       SAMPL
0023 CHNG       SBR  CHNGX&5                                                 SAMPL
0024            MLCA IDENT,AR80                                              SAMPL
0025            MLCA @   @                                                   SAMPL
0026            MLCS @   @,AR04&1                                            SAMPL
0027            A    &1,CNT#4                                                SAMPL
0028            MLCA CNT,AR04                                                SAMPL
0029 CHNGX      B    0                                                       SAMPL
0030 *                                                                       SAMPL
0031 PUNCH      SBR  PCHXT&5                                                 SAMPL
0032            STDIOPUNCH,AREA                                              SAMPL
0033 PCHXT      B    0                                                       SAMPL
0034 *                                                                       SAMPL
0035 EOF        IOCTLTYPE,EOJ                                                SAMPL
0036            B    /EOP/                                                   SAMPL
0037 *                                                                       SAMPL
0038 ERROR      IOCTLTYPE,ERR                                                SAMPL
0039            B    /UEP/                                                   SAMPL
0040 *                                                                       SAMPL
0041            LTORG*                                                       SAMPL
0042 *                                                                       SAMPL
0043 AREA       DA   1X80,G                                                  SAMPL
0044 AR04            4                                                       SAMPL
0045 AR80            80                                                      SAMPL
0046 *                                                                       SAMPL
0047  EOJ       DC   @  EOJ@,G                                               SAMPL
0048  ERR       DC   @ERROR@,G                                               SAMPL
0049            PST                                                          SAMPL
0050            END  START                                                   SAMPL
     MON$$      EXEQ LINKLOAD
                PHASESAMPLE
                CALL RESEQUENCE


     MON$$      EXEQ SAMPLE,MJB
           AUTOCODER SAMPLE PROGRAM DECK OF DUMMY INPUT CARDS               XXXXX
     THIS DECK OF CARDS IS  PROVIDED WITH THE SAMPLE PROBLEM TO SUPPLY A
     FEW DUMMY INPUT CARDS FOR OPERATING IN THE ASSEMBLE-AND-GO MODE.
     NOTE THAT THE FIRST CARD - THREE CARDS EARLIER IN THIS DECK - CONTAINS
     FIVE CHARACTERS IN CARD COLUMNS 76 THROUGH 80.  AFTER EXECUTION OF THE
     SAMPLE PROGRAM, ALL THE CARDS PRODUCED WILL CONTAIN THESE CHARACTERS
     IN THE CORRESPONDING CARD COLUMNS.  NOTE ALSO THAT EACH OF THE CARDS
     PRODUCED WILL CONTAIN A SEQUENCE NUMBER IN CARD COLUMNS 1 THROUGH 5.
     MON$$      END
```

Exhibit I

```
# ØØØØØ

S 149ØØ 1bbbb 11622 bb bbb bbbb

D ØØØØØ
D bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

A AČ%BØØØØ12$Ñ

R DATE 64Ø15
R SØ1 JOB  SAMPLE
R ASGN MJB,A1
R MODE GO
R ASGN MGO,A6
R EXEQ AUTOCODER
R 1Ø1Ø1 NR1 M@4Ø144ØØW
R EXEQ LINKLOAD
R EXEQ SAMPLE,MJB
R   EOJ
R END
R END SIU
R ENTER B MESSAGES

# ØØØØØ
```

Exhibit II

```
                                                                    PAGE  001

     64015      TITLERESEQUENCE00000                                    5001SAMPL
                BASE1**+X00                                             4002SAMPL
  00000 00049 J00054  D00394003O6L J00033  J00054  J00118          ))))D)1003SAMPL
  00040 00035 J00185  J00026  G00116B J/RSI/  J00216               ) ))))1004SAMPL
  00075 00043 W00259/MCS/M D/CRD/00104/ D00000000315$ J00000          AAC1005SAMPL
  00118 00037 G00183B D00306003947 D00309 D00309003193              D-D)1006SAMPL
  00155 00044 A00310003l4 D00314003l8T J00000  G00214B Y/PCH/X       ) D41007SAMPL
  00199 00005                                                           1008SAMPL
  00204 00036 00315 J00000  V00216/CTB/2 Y/CNC/X 00396             N C N1009SAMPL
  00240 00043 V00240/CTB/2 J/EOP/  V00259/CTB/2 Y/CNC/X 00402      N C C1010SAMPL
  00283 00019 V00283/CTB/2 J/UEP/                                     C1011SAMPL
  00302 00013           A                                             1012SAMPL
  00315 00395                                                        NY013SAMPL
  00395 00001                                                           1014SAMPL
  00396 00012  EOJ  ERROR                                               1015SAMPL
E 00000                                                                 3999SAMPL
```

Exhibit III

```
MON$$ SO1 JOB  SAMPLE
MON$$     ASGN MJB,A1
MON$$     MODE GO
MON$$     ASGN MGO,A6
MON$$     EXEQ AUTOCODER
```

Exhibit IV, Part A

```
      64015                    SAMPLE PROGRAM USING 1410/7010 AUTOCODER                    PAGE    1              SAMPL
SEQNO  PGLIN  LABEL     OPCOD  OPERAND                                  REL  CT  ADDRS  INSTRUCTION          CARD  FLAG

    1  0002              TITLE  RESEQUENCE                                                                    001
    2  0003              BASE1  *&X00                                                                        002
    3  0004  *
    4  0005  *     THIS  PROGRAM  WILL  RESEQUENCE  SOURCE  DECKS
    5  0006  *     RENUMBERING  FROM  0001  IN  THE  FIRST  FOUR
    6  0007  *     COLUMNS  AND  WITH  THE  IDENTIFICATION  FIELD
    7  0008  *     SUPPLIED BY THE FIRST CARD GANG-PUNCHED INTO EACH
    8  0009  *     CARD PRODUCED.
    9  0010  *
   10  0011  START       B      READ                                     ¤    7  00000  J 00054              003
   11  0012              MLCB   AR80,IDENT#5                             D   12  00007  D 00394 00306 L       003
   12  0013              B      *&8                                      ¤    7  00019  J 00033              003
   13  0014  LOOP        B      READ                                     ¤    7  00026  J 00054              003
   14  0015              B      CHNG                                     ¤    7  00033  J 00118              003
   15  0016              B      PUNCH                                    ¤    7  00040  J 00185              004
   16  0017              B      LOOP                                     ¤    7  00047  J 00026              004
   17  0018  *
   18  0019  READ        SBR    RDXT&5                                   ¤    7  00054  G 00116         B    004
   19  0020              STDIO  READ,AREA,EOF,ERROR,M                    L
   20 G 00090            B      /RSI/                                    T    7  00061  J /RSI/              004
   21 G 00100            B      EOF                                      ¤    7  00068  J 00216              004
   22 G 00110            BBE    ERROR,/MCS/,M                            C   12  00075  W 00259 /MCS/ M      005
   23 G 00140            MLNA   /CRD/,*&6                                A   12  00087  D /CRD/ 00104 /       005
   24 G 00150            MRCG   0,AREA                                   A   12  00099  D 00000 00315 $      005
   25  0021  RDXT        B      0                                        L
                                                                         T    7  00111  J 00000              005
   26  0022  *
   27  0023  CHNG        SBR    CHNGX&5                                  ¤    7  00118  G 00183         B    006
   28  0024              MLCA   IDENT,AR80                               D   12  00125  D 00306 00394 T      006
   29  0025              MLCA   @   @                                    @    6  00137  D 00309              006
   30  0026              MLCS   @   @,AR04&1                            D   12  00143  D 00309 00319 3      006
   31  0027              A      &1,CNT#4                                 4   11  00155  A 00310 00314        007
   32  0028              MLCA   CNT,AR04                                 D   12  00166  D 00314 00318 T      007
   33  0029  CHNGX       B      0                                        L
                                                                         T    7  00178  J 00000              007
   34  0030  *
   35  0031  PUNCH       SBR    PCHXT&5                                  ¤    7  00185  G 00214         B    007
```

Exhibit IV, Part B

```
    64015                    SAMPLE PROGRAM USING 1410/7010 AUTOCODER                    PAGE    2              SAMPL
SEQNO  PGLIN  LABEL     OPCOD   OPERAND                          REL  CT  ADDRS  INSTRUCTION      CARD  FLAG

   36    0032           STDIO   PUNCH,AREA
   37 G  00320          BXPA    /PCH/                            LT   7  00192  Y /PCH/       X  007
   38 G  00330          DCW     #5                                    5  00203                    008
   39 G  00340          DCW     AREA                             N    5  00208    00315          009
   40    0033  PCHXT    B       0                                LT   7  00209  J 00000          009
   41    0034  *
   42    0035  EOF      IOCTL   TYPE,EOJ
   43 G  0035  EOF      EQU     *                                             00216
   44 G  01510          BZN     *-11,/CTB/                       CLT  12  00216  V 00216 /CTB/ 2  009
   45 G  01520          BXPA    /CNC/                            LT   7  00228  Y /CNC/       X  009
   46 G  01530          DCW     EOJ                              N    5  00239    00396          009
   47 G  01580          BZN     *-11,/CTB/                       CLT  12  00240  V 00240 /CTB/ 2  010
   48    0036           B       /EOP/                            LT   7  00252  J /EOP/          010
   49    0037  *
   50    0038  ERROR    IOCTL   TYPE,ERR
```

```
    64015                    SAMPLE PROGRAM USING 1410/7010 AUTOCODER                    PAGE    3              SAMPL
SEQNO  PGLIN  LABEL     OPCOD   OPERAND                          REL  CT  ADDRS  INSTRUCTION      CARD  FLAG

   51 G  0038  ERROR    EQU     *                                             00259
   52 G  01510          BZN     *-11,/CTB/                       CLT  12  00259  V 00259 /CTB/ 2  010
   53 G  01520          BXPA    /CNC/                            LT   7  00271  Y /CNC/       X  010
   54 G  01530          DCW     ERR                              N    5  00282    00402          010
   55 G  01580          BZN     *-11,/CTB/                       CLT  12  00283  V 00283 /CTB/ 2  011
   56    0039           B       /UEP/                            LT   7  00295  J /UEP/          011
   57    0040  *
   58    0041           LTORG   *                                      00302
   59          IDENT            #0005                                 5  00306                    012
   60                           @  @                                  3  00309                    012
   61                           @A@                                   1  00310                    012
   62          CNT              #0004                                 4  00314                    012
   63    0042  *
   64    0043  AREA     DA      1X80,G                                81  00315
   65    0044  AR04             4                                        00318
   66    0045  AR80             80                                       00394
   67    0046  *
   68    0047   EOJ     DC      @  EOJ@,G                             6  00396                    015
   69    0048   ERR     DC      @ERROR@,G                             6  00402                    015
   70    0049           PST
   71    0050           END     START                                         00000

                                NUMBER OF FLAGGED STATEMENTS  NONE

                          1410/7010 AUTOCODER...SYSTEM /MID/ 0001
```

```
                    *****C R O S S    R E F E R E N C E    L I S T I N G*****
SAMPLE PROGRAM USING 1410/7010 AUTOCODER
SYMBOL     LABEL SEQNO          OPERAND SEQUENCE NUMBERS
/CNC/                      0045 0053
/CRD/                      0023
/CTB/                      0044 0047 0052 0055
/EOP/                      0048
/MCS/                      0022
/PCH/                      0037
/RSI/                      0020
/UEP/                      0056
AREA          0064         0024 0039
AR04          0065         0030 0032
AR80          0066         0011 0028
CHNG          0027         0014
CHNGX         0033         0027
CNT           0062         0031 0032
EOF           0043         0021
EOJ           0068         0046
ERR           0069         0054
ERROR         0051         0022
IDENT         0059         0011 0028
LOOP          0013         0016
PCHXT         0040         0035
PUNCH         0035         0015
RDXT          0025         0018
READ          0018         0010 0013
START         0010         0071


SYMBOL     LABEL SEQNO          OPERAND SEQUENCE NUMBERS FOR LITERALS IN SEGMENT 1
@  @          0060         0029 0030
@A@           0061         0031
```

Exhibit IV, Part C

```
     MON$$     EXEQ LINKLOAD
MESSAGE  CARD    NAME      VALUE     SYMBOL  VALUE     SYMBOL  VALUE     SYMBOL  VALUE     SYMBOL  VALUE


        PHASE  SAMPLE      18324
        BASE1              18400

        TITLE  RESEQUENCE  18400 REL FACTOR  18400

        UNRESOLVED ENTRIES              NONE
     MON$$     EXEQ SAMPLE,MJB
     MON$$     END
```

Exhibit IV, Part D

```
0001         AUTOCODER SAMPLE PROGRAM DECK OF DUMMY INPUT CARDS              XXXXX
0002 THIS DECK OF CARDS IS  PROVIDED WITH THE SAMPLE PROBLEM TO SUPPLY A     XXXXX
0003 FEW DUMMY INPUT CARDS FOR OPERATING IN THE ASSEMBLE-AND-GO MODE.        XXXXX
0004 NOTE THAT THE FIRST CARD - THREE CARDS EARLIER IN THIS DECK - CONTA     XXXXX
0005 FIVE CHARACTERS IN CARD COLUMNS 76 THROUGH 80.  AFTER EXECUTION OF     XXXXX
0006 SAMPLE PROGRAM, ALL THE CARDS PRODUCED WILL CONTAIN THESE CHARACTER     XXXXX
0007 IN THE CORRESPONDING CARD COLUMNS.  NOTE ALSO THAT EACH OF THE CARD     XXXXX
0008 PRODUCED WILL CONTAIN A SEQUENCE NUMBER IN CARD COLUMNS 1 THROUGH 5     XXXXX
```

Exhibit V

Appendix D: Autocoder Operation Codes

This appendix contains a complete listing of Autocoder language statements and, where applicable, their machine-language equivalents. In this listing, A-addresses in operands have been equated to location 12345, B-addresses have been equated to location 34567, and I-addresses have been equated to location 56789. C-addresses (for the Store Register instructions) have been equated to 45678. The character "D" in an operand indicates that an appropriate machine-language d-modifier is to be coded in the Autocoder source statement.

Included in the listing are several generalized forms for source statements. For example, the programmer can cause the generation of any machine-language instruction with an operation code of "J" by using the generalized form JID I,D. Place JID in the op code field, and the branch address (I-address) and appropriate d-modifier in the operand field. The generalized forms permit the coding of instructions for which there are no specific Autocoder mnemonics. Addresses in the operand of the generalized forms can be specified symbolically with labels, and can have address adjustment and indexing.

No diagnostics are performed on the d-modifiers associated with generalized mnemonics.

```
                    AUTOCODER MNEMONIC OPERATION CODES
LABEL       OPCOD  OPERAND                                        INSTRUCTION

*             PROCESSOR CONTROL OPERATIONS
            ORG    10000     ORIGIN
            LTORG  *         LITERAL ORIGIN
*           HEADR  THIS INFORMATION WILL BE THE HEADING OF LISTING
*           RESEQ           RESEQUENCE AT 001, NEW PAGE, NEW IDENT
*           EJECT           CONTINUE LISTING ON NEW PAGE
*           SPEND  START    END PRIMARY SUBPROGRAM
*           SPEND           END SECONDARY SUBPROGRAM
*           END    START    END PRIMARY SUBPROGRAM AND AUTOCODER RUN
*           END             END SECONDARY SUBPRGRM AND AUTOCODER RUN
            NOPWM             NO OPERATION WORD MARK                N
*             LINKAGE LOADER CONTROL OPERATIONS
*           BASE1  12000   CONTROLS RELOCATION FACTOR
*           BASE2  38000   UPPER LIMIT FOR COMMON DATA AREA
*           TITLE  COSINE,20,0  DECLARES NAME OF SUBPROGRAM,
*                     SIZE OF COMMON, ORIGIN POINT
SINE/       DEFIN  COSINE+42   DECLARES LINKAGE SYMBOL
*           PRTCT  ABCD/   PREVENTS ERASURE OF LINKAGE SYMBOLS
*                        FROM LINKAGE LOADER TABLE
*           CALL   THIRD   SUBPROGRAM CALL
*              DECLARATIVE OPERATIONS
A           EQU    12345    THE EQUATE INSTRUCTION
B           EQU    34567
C           EQU    45678
I           EQU    56789
```

AUTOCODER MNEMONIC OPERATION CODES

LABEL	OPCOD	OPERAND		INSTRUCTION
	DA	1X2,G	DEFINE AREA	3 10001
	DAV	1X2,G	DEFINE AREA IN COMMON AREA	3 10004
	DCW	- -	DEFINE CONSTANT WITH WORD MARK	1 10007
	DC	+2	DEFINE CONSTANT	1 10008
	DS	1	DEFINE SYMBOL	1 10009
	DCWF	NAME	ADCON FOR ENTRY POINT OF NAMED SUBPROGRAM	5 10010
	DCWS	NAME	BRANCH TO NAMED SUBPROGRAM	7 10015
	RSV	LABEL	APPLY DOWNWARD RELOCATION TO LABEL	
*		ARITHMETIC OPERATIONS		
	A	A,B	ADD A-FIELD TO B-FIELD	A 12345 34567
	S	A,B	SUBTRACT A FROM B	S 12345 34567
	ZA	A,B	ZERO AND ADD A TO B	+ 12345 34567
	ZS	A,B	ZERO AND SUBTRACT A FROM B	- 12345 34567
	M	A,B	MULTIPLY	- 12345 34567
	D	A,B	DIVIDE	(12345 34567
*		MOVE RIGHT TO LEFT COMMANDS		
*		MOVE SINGLE POSITION		
	MLNS	A,B	MOVE LEFT NUMERIC SINGLE	D 12345 34567 1
	MLZS	A,B	ZONES SINGLE	D 12345 34567 2
	MLCS	A,B	CHARACTERS SINGLE	D 12345 34567 3
	MLWS	A,B	WORD MARKS SINGLE	D 12345 34567 4
	MLNWS	A,B	NUMERIC AND WORD MARK SINGLE	D 12345 34567 5
	MLZWS	A,B	ZONE AND WORD MARK SINGLE	D 12345 34567 6
	MLCWS	A,B	CHARACTER AND WORD MARK SINGLE	D 12345 34567 7
*		STOP MOVE AT WORD MARK IN A-FIELD		
	MLNA	A,B	MOVE LEFT NUMERIC TO A-FIELD WORD MARK	D 12345 34567 /
	MLZA	A,B	ZONES TO A-FIELD WORD MARK	D 12345 34567 S
	MLCA	A,B	CHARACTERS TO A-FIELD WORD MARK	D 12345 34567 T
	MLWA	A,B	WORD MARKS TO A-FIELD WORD MARK	D 12345 34567 U
	MLNWA	A,B	NUMERIC AND WM TO WORD MARK IN A	D 12345 34567 V
	MLZWA	A,B	ZONES AND WM TO WORD MARK IN A	D 12345 34567 W
	MLCWA	A,B	CHARACTERS AND WM TO WORD MK IN A	D 12345 34567 X
*		STOP MOVE AT WORD MARK IN B-FIELD		
	MLNB	A,B	MOVE LEFT NUMERIC TO B-FIELD WORD MARK	D 12345 34567 J
	MLZB	A,B	ZONES TO B-FIELD WORD MARK	D 12345 34567 K
	MLCB	A,B	CHARACTERS TO B-FIELD WORD MARK	D 12345 34567 L
	MLWB	A,B	WORD MARKS TO B-FIELD WORD MARK	D 12345 34567 M
	MLNWB	A,B	NUMERIC AND WM TO WORD MARK IN B	D 12345 34567 N
	MLZWB	A,B	ZONES AND WM TO WORD MARK IN B	D 12345 34567 O
	MLCWB	A,B	CHARACTERS AND WM TO WORD MK IN B	D 12345 34567 P
*		STOP MOVE AT WORD MARK IN A- OR B-FIELD		
	MLN	A,B	MOVE LEFT NUMERIC	D 12345 34567 A
	MLZ	A,B	ZONES	D 12345 34567 B

AUTOCODER MNEMONIC OPERATION CODES

LABEL	OPCOD	OPERAND		INSTRUCTION	
	MLC	A,B	CHARACTERS	D 12345 34567	C
	MLW	A,B	WORD MARKS	D 12345 34567	D
	MLNW	A,B	NUMERIC AND WORD MARKS	D 12345 34567	E
	MLZW	A,B	ZONES AND WORD MARKS	D 12345 34567	F
	MLCW	A,B	CHARACTERS AND WORD MARKS	D 12345 34567	G
*		MOVE LEFT TO RIGHT COMMANDS			
*		STOP MOVE AT WORD MARK IN A- OR B-FIELD			
	MRN	A,B	MOVE RIGHT NUMERIC	D 12345 34567	9
	MRZ	A,B	ZONES	D 12345 34567	0
	MRC	A,B	CHARACTERS	D 12345 34567	=
	MRW	A,B	WORD MARKS	D 12345 34567	-
	MRNW	A,B	NUMERIC AND WORD MARKS	D 12345 34567	.
	MRZW	A,B	ZONES AND WORD MARKS	D 12345 34567	G T
	MRCW	A,B	CHARACTERS AND WORD MARKS	D 12345 34567	T M
*		STOP MOVE AT RECORD MARK IN A-FIELD			
	MRNR	A,B	MOVE RIGHT NUMERIC TO RECORD MARK IN A-FLD	D 12345 34567	Z
	MRZR	A,B	ZONES TO RECORD MARK IN A-FIELD	D 12345 34567	#
	MRCR	A,B	CHARACTERS TO RECORD MARK IN A	D 12345 34567	,
	MRWR	A,B	WORD MARKS TO RECORD MARK IN A	D 12345 34567	(W
	MRNWR	A,B	NUMERIC AND WM TO RM IN A-FIELD	D 12345 34567	S
	MRZWR	A,B	ZONES AND WM TO RM IN A-FIELD	D 12345 34567	' Q
	MRCWR	A,B	CHARACTERS AND WM TO RM IN A	D 12345 34567	T
	MRNG	A,B	MOVE RIGHT NUMERIC TO GM-WM IN A-FIELD	D 12345 34567	R -
	MRZG	A,B	ZONES TO GM-WM IN A-FIELD	D 12345 34567	0
	MRCG	A,B	CHARACTERS TO GM-WM IN A-FIELD	D 12345 34567	$
	MRWG	A,B	WORD MARKS TO GM-WM IN A-FIELD	D 12345 34567	* R
	MRNWG	A,B	NUMERIC AND WM TO GM-WM IN A	D 12345 34567	P .
	MRZWG	A,B	ZONES AND WM TO GM-WM IN A-FIELD	D 12345 34567	, D
	MRCWG	A,B	CHARACTERS AND WM TO GM-WM IN A	D 12345 34567	L
*		STOP AT RM OR GM-WM IN A-FIELD			
	MRNM	A,B	MOVE RIGHT NUMERIC TO RM OR GM-WM	D 12345 34567	I +
	MRZM	A,B	ZONES TO RM OR GM-WM	D 12345 34567	0
	MRCM	A,B	CHARACTERS TO RM OR GM-WM	D 12345 34567	.
	MRWM	A,B	WORD MARKS TO RM OR GM-WM	D 12345 34567	) L
	MRNWM	A,B	NUMERIC AND WM TO RM OR GM-WM	D 12345 34567	P L
	MRZWM	A,B	ZONES AND WM TO RM OR GM-WM	D 12345 34567	T G
	MRCWM	A,B	CHARACTERS AND WM TO RM OR GM-WM	D 12345 34567	M
*		SCAN LEFT AND RIGHT COMMANDS			
	SCNRR	A,B	SCAN RIGHT TO RM IN A-FIELD	D 12345 34567	Y
	SCNRG	A,B	TO GM-WM IN A-FIELD	D 12345 34567	Q
	SCNRM	A,B	TO RM OR GM-WM IN A-FIELD	D 12345 34567	H
	SCNR	A,B	TO WORD MARK IN A- OR B-FIELD	D 12345 34567	8

AUTOCODER MNEMONIC OPERATION CODES

LABEL	OPCOD	OPERAND		INSTRUCTION	C
	SCNLA	A,B	SCAN LEFT TO WORD MARK IN A-FIELD	D 12345 34567	T
	SCNLB	A,B	TO WORD MARK IN B-FIELD	D 12345 34567	-
	SCNL	A,B	TO WM IN A- OR B-FIELD	D 12345 34567	+
	SCNLS	A,B	SINGLE POSITION	D 12345 34567	
*		SPECIAL MOVE COMMANDS			
	MCS	A,B	MOVE CHARACTERS AND SUPPRESS ZEROS	Z 12345 34567	
	MCE	A,B	MOVE CHARACTERS AND EDIT	E 12345 34567	
*		COMPARE AND LOOKUP COMMANDS			
	C	A,B	COMPARE B-FIELD TO A-FIELD	C 12345 34567	
	LL	A,B	LOOKUP LOW	T 12345 34567	1
	LE	A,B	LOOKUP EQUAL	T 12345 34567	2
	LLE	A,B	LOOKUP LOW OR EQUAL	T 12345 34567	3
	LH	A,B	LOOKUP HIGH	T 12345 34567	4
	LLH	A,B	LOOKUP LOW OR HIGH	T 12345 34567	5
	LEH	A,B	LOOKUP EQUAL OR HIGH	T 12345 34567	6
*		LOGICAL OPERATIONS			
	BW	I,B	BRANCH TO I-ADDR IF WORD MARK AT B-ADDRESS	V 56789 34567	1
	BZN	I,B	BRANCH TO I IF B HAS NO AB-BITS	V 56789 34567	2
	BZN	I,B,AB	IF B HAS A-BIT AND B-BIT	V 56789 34567	B
	BZN	I,B,+	A-BIT AND B-BIT	V 56789 34567	B
	BZN	I,B,A	IF B HAS A-BIT AND NO B-BIT	V 56789 34567	S
	BZN	I,B,C T	A-BIT AND NO B-BIT	V 56789 34567	S
	BZN	I,B,B	IF B HAS B-BIT AND NO A-BIT	V 56789 34567	K
	BZN	I,B,-	B-BIT AND NO A-BIT	V 56789 34567	K
	BWZ	I,B	BRANCH TO I IF B HAS WM AND NO AB-BITS	V 56789 34567	3
	BWZ	I,B,AB	AND AB-BITS	V 56789 34567	C
	BWZ	I,B,+	AND AB-BITS	V 56789 34567	C
	BWZ	I,B,A	AND A-BIT	V 56789 34567	T
	BWZ	I,B,C T	AND A-BIT	V 56789 34567	T
	BWZ	I,B,B	AND B-BIT	V 56789 34567	L
	BWZ	I,B,-	AND B-BIT	V 56789 34567	L
	BCE	I,B,D	BRANCH TO I IF CHARACTER AT B EQU D-MOD	B 56789 34567	D
	BBE	I,B,D	BRNCH IF ANY BIT AT B MATCHES BIT IN D-MOD	W 56789 34567	D
	B	I	UNCONDITIONAL BRANCH	J 56789	
	BU	I	BRANCH IF COMPARE UNEQUAL	J 56789	/
	BE	I	EQUAL	J 56789	S
	BL	I	LOW	J 56789	T
	BH	I	HIGH	J 56789	U
	BZ	I	BRANCH IF ZERO BALANCE	J 56789	V
	BAV	I	BRANCH IF ARITHMETIC OVERFLOW	J 56789	Z
	BDV	I	BRANCH IF DIVIDE OVERFLOW	J 56789	W

AUTOCODER MNEMONIC OPERATION CODES

LABEL	OPCOD	OPERAND		INSTRUCTION	
*		MISCELLANEOUS OPERATIONS			
	SAR	C	STORE A-REGISTER	G 45678 A	
	SBR	C	STORE B-REGISTER	G 45678 B	
	SW	A	SET WORD MARK AT A	, 12345	
	SW	A,B	SET WORD MARK AT A AND B	, 12345 34567	
	CW	A	CLEAR WORDMARK AT A	) 12345	
	CW	A,B	CLEAR WORD MARK AT A AND B	) 12345 34567	
	CS	B	CLEAR STORAGE	/ 34567	
	CS	I,B	CLEAR STORAGE AND BRANCH	/ 56789 34567	
	NOP		NO OPERATION	N	
	STC	A	STORE TIME CLOCK	G 12345	T
	SR	C,D	GENERALIZED STORE REGISTER	G 45678	D
	STCPU	I	STORE CPU STATUS	$ 56789	S
	RSCPU	I	RESTORE CPU STATUS	$ 56789	R
*		FLOATING POINT ARITHMETIC INSTRUCTIONS			
	FRA	A	FLOATING RESET ADD	= 12345	R
	FST	A	FLOATING STORE	= 12345	L
	FA	A	FLOATING ADD	= 12345	A
	FS	A	FLOATING SUBTRACT	= 12345	S
	FM	A	FLOATING MULTIPLY	= 12345	M
	FD	A	FLOATING DIVIDE	= 12345	D
	BXO	I	BRANCH EXPONENT OVERFLOW	J 56789	Y
	BXU	I	BRANCH EXPONENT UNDERFLOW	J 56789	X

NOTE: The remainder of this listing presents instructions that are (or can be) related to input/output functions, including the use of priority and overlap. They are "restricted" in that special care must be given to their use, since they are potential hazards to such Resident Monitor control functions as input/output scheduling and assignment of input/output units. Because all programs within the Operating System are provided with input/output facilities by the system's IOCS, the majority of these instructions will not normally be used in coding an Autocoder program. The programmer who wishes to use any of these instructions is advised to be familiar with the extended use of the IOCS, as explained in the publication IBM 1410/7010 Operating System: Basic Input/Output Control System, Form C28-0322.

*		CONDITIONAL BRANCHES FOR I/O, OVERLAP, AND PRIORITY			
	BEX1	I,D	BRANCH EXTERNAL INDICATOR - CHANNEL 1	R 56789	D
	BEX2	I,D	- CHANNEL 2	X 56789	D
	BEX3	I,D	- CHANNEL 3	3 56789	D
	BEX4	I,D	- CHANNEL 4	1 56789	D

NOTE: The Branch External Indicator instructions must not be used in any form, for any purpose, by dependent programs within the Operating System. These instructions reset certain interrupt indicators and can result in either an I/O interlock error or failure of IOCS functions. The BEX mnemonic is included in this listing merely as an aid for reading assembly listings of the Resident IOCS.

	BOL1	I	BRANCH OVERLAP IN PROCESS - CHANNEL 1	J 56789	1
	BOL2	I	- CHANNEL 2	J 56789	2
	BOL3	I	- CHANNEL 3	J 56789	3
	BOL4	I	- CHANNEL 4	J 56789	4
	BB1	I	BRANCH IF BINARY CARD - CHANNEL 1	J 56789	M
	BB2	I	- CHANNEL 2	J 56789	(

AUTOCODER MNEMONIC OPERATION CODES

LABEL	OPCOD	OPERAND		INSTRUCTION	
	B	I	BRANCH PRINTER CARRIAGE BUSY - CHANNEL 1	J 56789	R
	BPCB1	I	- CHANNEL 1	J 56789	R
	BPCB2	I	- CHANNEL 2	J 56789	L
	BCV	I	BRANCH CARRIAGE OVERFLOW - CHANNEL 1	J 56789	-
	BCV1	I	- CHANNEL 1	J 56789	-
	BCV2	I	- CHANNEL 2	J 56789	)
	BC9	I	BRANCH CARRIAGE CHANNEL 9 - CHANNEL 1	J 56789	9
	BC91	I	- CHANNEL 1	J 56789	9
	BC92	I	- CHANNEL 2	J 56789	0
	BXPA	I	BRANCH AND EXIT PRIORITY ALERT	Y 56789	X
	BEPA	I	BRANCH AND ENTER PRIORITY ALERT	Y 56789	E
*	JID	I,D	GENERALIZED TEST AND BRANCH		
*		THE ABOVE IS A GENERALIZED FORM, PERTAINING TO			
*		THE J OPCODE. THE PROPER D MODIFIER MUST BE			
*		SUBSTITUTED BY THE USER FOR THE D SHOWN IN THE OPERAND			
*	BPI	I,D	GENERALIZED PRIORITY TEST AND BRANCH		
*		THE ABOVE IS A GENERALIZED FORM, PERTAINING TO			
*		THE Y OPCODE. THE PROPER D MODIFIER MUST BE			
*		SUBSTITUTED BY THE USER FOR THE D SHOWN IN THE OPERAND			

NOTE: The Priority Test and Branch instructions reset indicators tested by the Resident IOCS and, therefore, should be used with special care. A thorough knowledge of the internal functions of the IOCS is prerequisite for use of these instructions in a dependent program.

	STATS	I,D	GENERALIZED STORE AND RESTORE STATUS	$ 56789	D
*		THE ABOVE IS A GENERALIZED FORM. THE PROPER			
*		D MODIFIER MUST BE SUBSTITUTED FOR THE D SHOWN.			
*		FOUR EXAMPLES OF THIS USAGE ARE			
	STATS	I,E	STORE CHANNEL 1 STATUS	$ 56789	E
	STATS	I,G	STORE CHANNEL 3 STATUS	$ 56789	G
	STATS	I,1	RESTORE CHANNEL 1 STATUS	$ 56789	1
	STATS	I,4	RESTORE CHANNEL 4 STATUS	$ 56789	4
	SSF	0	SELECT STACKER 0 AND FEED - CHANNEL 1	K	0
	CC	1	CARRIAGE CONTROL I/O CHANNEL 1	F	1
	BSP	(B1	BACKSPACE TAPE	U (B1	B
	WTM	(B1	WRITE TAPE MARK	U (B1	M
	RWD	(B1	REWIND	U (B1	R
	RWU	(B1	REWIND AND UNLOAD	U (B1	U
	CU	(B1,W	CONTROL UNIT	U (B1	W
	MU	(B1,B,D	TO BUILD MOVE MODE I/O COMMAND	M (B1 34567	D
	LU	(B1,B,D	TO BUILD LOAD MODE I/O COMMAND	L (B1 34567	D
	H	I	HALT AND BRANCH	. 56789	

NOTE: The Halt instruction, although not necessarily related to input/output functions, is included in the "restricted" category because its use is in opposition to the Operating System convention that dependent programs should not interrupt Monitor control with machine halts. This convention is especially significant for dependent programs that run under control of a Resident Monitor that includes the Tele-processing Supervisor.

EOJ	END	

Index

IBM **Technical Newsletter**

File Number	1410/7010-22
Re: Form No.	C28-0326-2
This Newsletter No.	N27-1223
Date	June 24, 1965
Previous Newsletter Nos.	None

IBM 1410/7010 AUTOCODER

This Technical Newsletter amends the publication IBM 1410/7010 Operating System; Autocoder, Form C28-0326-2, to include new information concerning the DCW statement, the chaining of instructions and the use of zoned switches, and to correct minor errors.

The attached replacement pages (11-14, 29-30, and 41-42) should be substituted for the pages currently in the publication. Text changes are indicated by a vertical line at the left of the affected text.

In addition, the following changes should be made to the publication:

Page	Amendment
22	Change the Meaning of the mnemonic "HEADR" (under "Assembly Control Codes") to read Header Line
26	Under the heading "Assignment of Data Areas in COMMON", the first paragraph refers to "Figures 61 and 62". Change this reference to read: Figures 35 and 36
27	In the section "Use of Labels Referencing COMMON," there are two references to "Figure 61". Change these references to read: Figure 35
47	The first complete sentence on the page ends with "skip to statement 004". Change this to read: skip to the statment labeled L.
51	The second sentence in Note 2 ends with the word "common". The word "common" should be capitalized to read: COMMON

Please file this cover letter at the back of the publication. It provides a method of determining that all changes have been received and incorporated into the publication.

IBM Corporation, Programming Systems Publications, Dept. 637, Neighborhood Road, Kingston, N.Y. 12401

PRINTED IN U.S.A.

1. SEQNO — Sequence Number: The sequence number of statements as they appear in the assembly listing.

2. PGLIN — Page and Line Number: The page and line number as it appears in columns 1 through 5 of the cards in the source deck. Page and line numbers must consist of five non-blank characters and must appear in ascending sequence.

Statements generated by the macro generator will have a page and line number in this field supplied by the generator. These numbers have no relationship to the numbers of the hand-coded statements; they represent the order in which the statements appear in the Macro Library.

The space between the SEQNO and PGLIN columns of the listing are used by the processor to contain either an "S" or a "G," under the following conditions.

S — The page and line number of the statement is not in ascending sequence in relation to the preceding source statement. This is only a warning to the programmer that his source statements may be out of sequence.

G — This character differentiates statements produced by the macro generator from the hand-coded source statements.

3. LABEL — Label: The contents of the label field, columns 6 through 15, of the Autocoder statement.

4. OPCOD: The Operation Code, columns 16 through 20, of the Autocoder statement.

5. OPERAND: The contents of the operand field, columns 21 through 72, of the Autocoder statement.

6. REL — Relocation Indicator: This is a code character that indicates to the Linkage Loader the type of relocation to be applied to the element(s) in the statement.

7. CT — Character Count: The length in characters of the assembled imperative statement, or the number of core-storage locations reserved for a constant defined in a declarative statement.

8. ADDRS: The relative address assigned by the processor to the instruction or constant. This address is subject to relocation.

9. INSTRUCTION: The assembled machine-language instruction or constants from which the object deck is constructed.

10. CARD — Card Number: The sequence number of the card in which the associated constants or instructions appear in the object deck. This sequence number is automatically computed and placed in columns 73-75 of each card in the object deck, in ascending order.

11. FLAG: An alphabetic character indicating an actual or possible programming error. As many as five flags can be assigned to one Autocoder statement. The flags provided are as follows:

F — invalid statement Format

M — Multiple definition of a label

N — macro generation Note

O — invalid Operation code

R — Restricted operation code (if not generated by a macro)

U — Unidentified label in the operand

W — Warning, general classification of error

Details concerning the above flags can be found in Appendix A. The total number of flagged statements is indicated at the end of the assembly listing, followed by a line which contains the sequence number of each flagged statement, to a maximum of 20 numbers. The presence of any flag except "R" causes the processor to set the "no-go" switch during assembly. This setting of the "no-go" switch can cause a bypassing of all the source cards up to the next job. See the *System Monitor* publication.

The assembly listing can be supplemented by a cross reference listing at the option of the user, by means of the PST statement. This listing analyzes the subprogram(s) just assembled, and lists each label, followed by the sequence number of the statement in which it was defined, and the sequence number of each statement in which the label is used as a reference address. See "PST — Print Symbol Table," in the subsection "Control Operation Codes," for a more detailed explanation.

NOTE: The system symbol /LIN/ controls the line count on the listing page. However, if this system symbol calls for the printing of less than 30 lines per page the processor will reject this direction and print the assembly listing at the normal 55 lines per page. See the *System Monitor* publication for details concerning this system symbol.

Replacement Codes

The Autocoder processor utilizes a second line (normally blank) in the assembly listing, for the representation of non-printable characters. Each of these characters is represented by two characters, one printed above the other, at the appropriate place in the listing. These two-character substitutions are called replacement codes, and they appear most frequently as relocation indicators or operation modifiers.

The two-character replacement codes with their conventional graphic representations, card codes, and names are listed in Figure 3.

Replacement Code	Graphic	Card Code	Name
‡0	?	12-0	Plus Zero
-0	!	11-0	Minus Zero
GM	‡	12-7-8	Group Mark
QT	+++	0-7-8	Segment Mark
WS	⌒	0-5-8	Word Separator
DL	Δ	11-7-8	Delta
CT	¢ or ♭	2-8	Cent Sign or Substitute Blank
LP	[	12-5-8	Left Bracket
RP	]	11-5-8	Right Bracket
TM	√‾	7-8	Tape Mark
LT	<	12-6-8	Less Than
GT	>	6-8	Greater Than
;	;	11-6-8	Semicolon
:	:	5-8	Colon
"b	\	0-6-8	Backslash

Figure 3. Replacement Codes

Coding Sheet

The Autocoder Coding Sheet (Figure 4) provides a convenient form for coding source program statements. Column numbers on the coding sheet have a one-for-one correspondence to the columns on the card used to punch the source statements (*Autocoder Input Card*, Form A36199).

Each line of the coding sheet is punched into a separate card. The source deck, therefore, consists of a sequenced set of punched cards containing a line-by-line representation of the coding sheets.

The following paragraphs explain the function of each field. The heading information, *Program, Programmed By,* and *Date,* are only for documentation, and are not punched.

Identification (Card Columns 76-80)

This five-position field can contain a name created by the programmer to identify the program. This identification will be punched into 76-80 of the object deck only if it appears in a HEADR or RESEQ control card. (See "Control Operation Codes.") However, the identification is not checked on the other Autocoder statements, and serves only to identify the program to which the card belongs. Special, as well as alphameric, characters are permitted.

Page Number and Line Number (Card Columns 1-5)

The page number (columns 1 and 2), in conjunction with the line number (columns 3-5), provides a means of sequencing the cards in the source deck. This enables the programmer to identify and correlate the entries on the coding sheet and assembly listing with the entries in the source deck. Alphabetic, as well as numeric, characters can be used. (If the standard collating sequence is not followed, the processor will place a sequence (S) flag next to the PGLIN field in the assembly listing, as previously explained.)

Label (Card Columns 6-15)

This field, if used, contains the label being defined in this statement.

Operation Code (Card Columns 16-20)

This field contains the operation code.

Operand (Card Columns 21-72)

This field, if used, contains the operand element(s) of the statement.

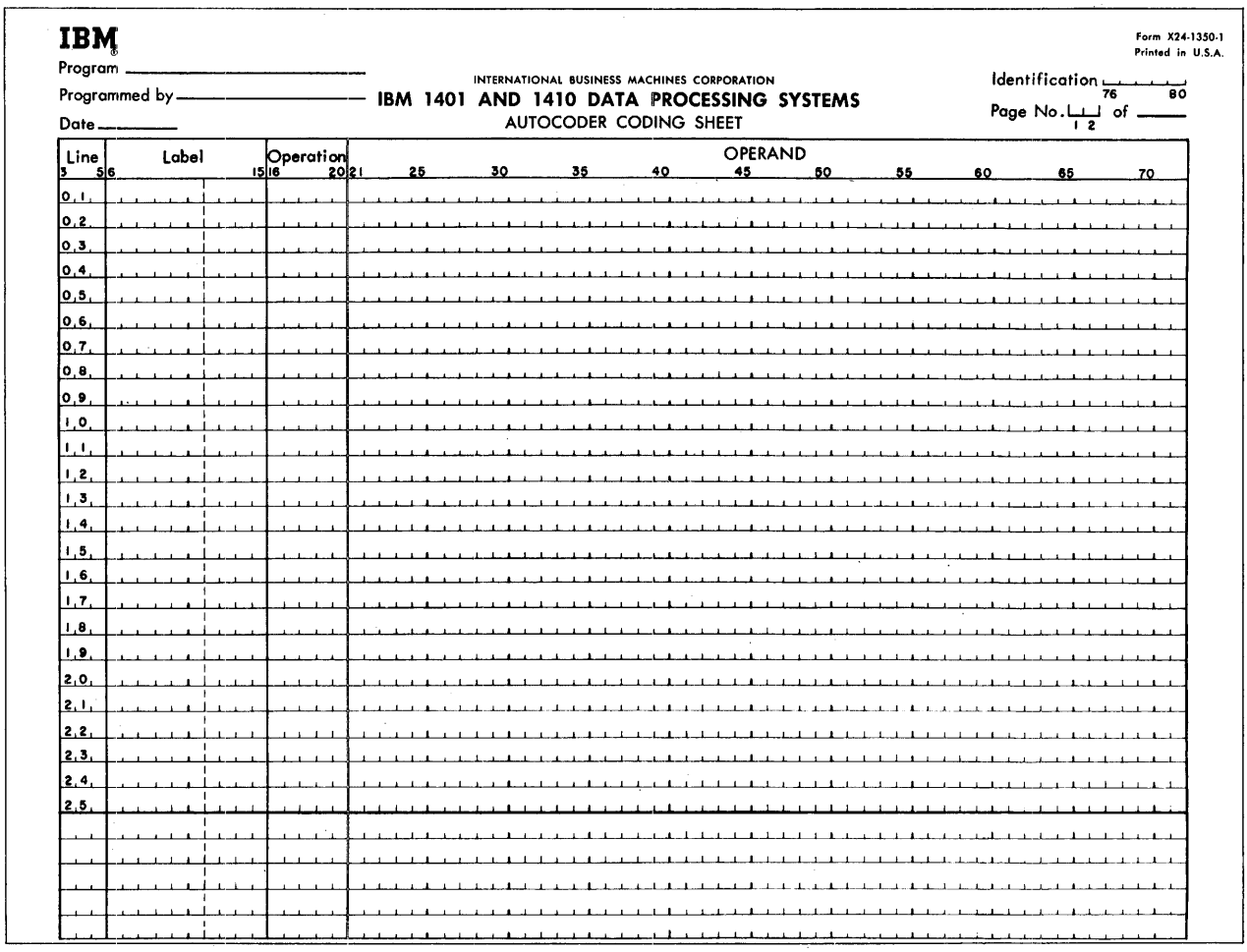
NOTE: Columns 73-75 should be left blank.

COMMENTS

Comments are remarks or notes written by the programmer in the operand field. At least two blank spaces must separate a comment from the last character of the statement. The comment, punched in the source deck, appears in the assembly listing but is not contained in the object deck, and has no effect on the object program.

COMMENTS CARD

It may, at times, be helpful to insert an entire line of descriptive information. This is done by placing an asterisk in column 6 and using the balance of the line (up to column 72) for comments. When this line of information is punched into a card of the source deck, the asterisk will identify it to the processor as a comments card. The comments will be printed in a single line of the assembly listing at the point of encounter, which can be anywhere in the source deck, except as


IBM

Form X24-1350-1
Printed in U.S.A.

Program _____

Programmed by _____

Date _____

INTERNATIONAL BUSINESS MACHINES CORPORATION

IBM 1401 AND 1410 DATA PROCESSING SYSTEMS

AUTOCODER CODING SHEET

Identification 76 80

Page No. 1 2 of _____

Line	Label	Operation	OPERAND
3 5	6 15	16 20	21 25 30 35 40 45 50 55 60 65 70
0,1			
0,2			
0,3			
0,4			
0,5			
0,6			
0,7			
0,8			
0,9			
1,0			
1,1			
1,2			
1,3			
1,4			
1,5			
1,6			
1,7			
1,8			
1,9			
2,0			
2,1			
2,2			
2,3			
2,4			
2,5			

Figure 4. The Coding Sheet Form

noted under "Implied DCW Operation Codes." Comments cards have no effect on the object program and are not included in the object deck. However, a comments card inserted in a series of chained Autocoder operation codes will break the chain. It is, therefore, necessary to restate the operation code and to provide the proper operand on the first source card statement following the comments card, in order to resume a chained action. In summary, a chain which is broken by a comment must be made into two chains.

Types of Operand Entries

This section explains the form and use of the various entries permitted in the operand field of imperative, declarative, Linkage Loader, and control statements.

The operand field of an Autocoder statement is used to specify a variety of information to the processor. The function of a specific entry is dependent upon the type of Autocoder statement in which it appears. The normal operand usage with each of the five types of Autocoder statements is as follows.

STATEMENT TYPE	OPERAND CONTENTS
IMPERATIVE	Symbolic address(es) to be operated upon by the machine instruction, and a d-modifier, when required
DECLARATIVE	Constants, symbols, and/or control parameters necessary to declare the desired fields
LINKAGE LOADER	Symbolic (or actual) addresses and/or control parameters required to convert the object deck into absolute format
CONTROL	Symbolic (or actual) addresses and constant information indicated by the operation code
MACRO	Parameters of the macro statement (These parameters are discussed in the section entitled, "The Macro System.")

All permissible operand entries are explained and illustrated under the following headings:

Basic Addresses
Address Adjustment
Indexing
Literals
Linkage Symbols
System Symbols
Miscellaneous

Basic Addresses

Basic addresses contained in the operand field of an Autocoder statement are the primary elements of information conveyed to the processor. They can be altered or modified by means of additional elements contained in the operand field.

A basic address is the symbolic or actual representation of a core-storage location of the data field or instruction referred to by the Autocoder statement.

A basic address can be in one of three forms:

Symbolic
Asterisk
Actual

Symbolic

A symbolic address is an operand entry that appears elsewhere in the source program as a label. As a rule, this symbol can be defined as a label either before or after the Autocoder statement in which it appears as an address. The exceptions to this rule are as follows:

1. All symbolic operands appearing in ORG, LTORG, and EQU statements must have been *previously* defined within the same program.

2. The symbolic address appearing in an RSV statement must *precede* any other use of this symbol in a program. (See "RSV — Reserve.")

3. The symbolic representations of index registers (X0, X1-X15) and the common data area (COMMON), must never appear in the label field. They cannot be defined by the user because they are predefined labels in the symbol table maintained by the Autocoder processor.

The instruction in Figure 5 illustrates the use of symbolic addresses. The symbols TOTAL and ACCUMULATE are defined as labels elsewhere in the program. The assembled instruction will cause the contents of the core-storage area labeled TOTAL to be moved to the area labeled ACCUMULATE.

NOTE: A symbolic address will receive upward, downward, or NO relocation, depending on the manner in which the symbol is defined.

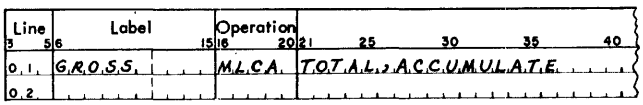


Figure 5. Autocoder Instruction with Symbolic Addresses

Asterisk (*)

An asterisk (11-4-8 punch) can be used as a basic address in an Autocoder statement. When compiling the object program, the processor will replace the asterisk with the relative core-storage address of the last character of the instruction or data field created by the statement in which it appears. However, if an asterisk address is used in a statement that does not cause the generation of an instruction or data area in the object program, the value substituted for the asterisk will be the current location in the object program.

Blank Constants

A field of blanks can be reserved by placing a # character (3-8 punch) in column 21, followed by a number indicating how many consecutive blank core-storage positions are to be defined (Figure 40). A word mark is set in the high-order position of this field.

NOTE: The number of successive blank constants that can be reserved by a DCW statement is limited to 500 positions of core storage. If this limit is exceeded, the processor will reserve only the maximum (500 positions), and attach an "F" flag to the statement on the assembly listing.

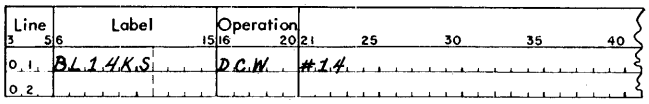


Figure 40. Field of 14 Blanks Defined in a DCW Statement

Address Constants

A DCW statement can be used to define an address constant. The constant is the address of the field whose label is written in the operand. For example (Figure 41), assume that the label MANNO is used in the symbolic program, and that it was assigned the address 00500 by the processor. The programmer can refer to the address of MANNO by using the symbolic label of the DCW statement.

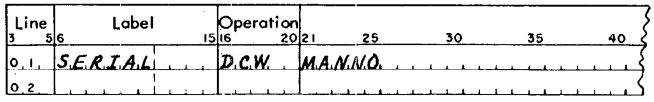


Figure 41. Address Constant

The five-character data field labeled SERIAL (Figure 41) will contain the address of the label MANNO (00500). The Linkage Loader will recognize address constants and adjust them by the proper relocation factor. Thus, SERIAL will contain the relocated address of MANNO.

If an address constant is address adjusted in a DCW statement, the constant is adjusted before it is assigned a storage location. In Figure 42, MANNO (actual address 00500) has been address adjusted by +12. Thus, the location labeled FICA will contain the address constant 00512.

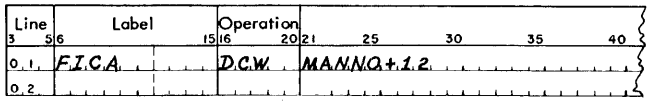


Figure 42. Address Constant with Address Adjustment Defined in DCW Statement

Address constants defined in a DCW statement can be indexed. The zone bit(s) indicating the specified index register becomes part of the constant.

NOTE 1: All address constants receive the same relocation indicators that were assigned to the symbol specified in the operand field.

NOTE 2: An address constant of a linkage or system symbol can be specified, and the desired address will be automatically supplied by the Linkage Loader. However, this form of address constant cannot be address adjusted or indexed.

Signed Address Constants

An address constant defined in a DCW statement can be signed. A and B bits will be generated by the processor over the units position, if the plus (+) sign was placed before the operand. The units position will contain a B bit if the minus (−) sign was used (Figure 43).

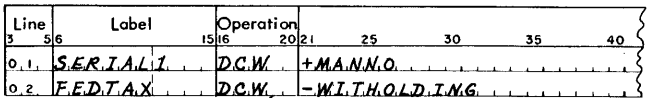


Figure 43. Signed Address Constants Defined in DCW Statement

Implied DCW Operation Codes

If several constants are to be defined in succession, only the first statement (or any statement preceded by a comments card) requires the mnemonic DCW in the operation field (Figure 44).

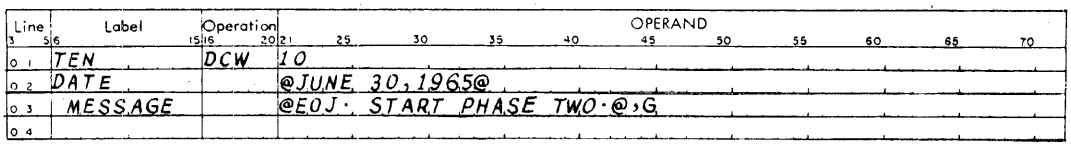


Figure 44. Successive DCW Statements with Blank Operation Columns

DC — Define Constant (no word mark)

The function performed by the DC statement, and the permissible forms of the constants, are identical to those described for the DCW statement. The only difference is that the word mark is absent when the constant is assigned to core storage (Figure 45).

Line	Label	Operation	
0 1	SERIAL	DC	@32816555T@
0 2	FIELD3		+000
0 3	SSNUMBER		@077-18-7500@
0 4			

Figure 45. Successive DC Statements with Blank Operation Columns

NOTE: The restriction on the use of an initial word separator character in the DCW statement defining an alphameric constant does not apply to the DC statement.

DS — Define Symbol

The DS statement is used to label and define an area within the subprogram. No information is entered into the area, no word mark is assigned by the processor, and the area is not cleared prior to reservation. The programmer specifies the size of the area, and designates the symbolic label by which it will be referenced. The number of desired consecutive positions of core storage is written in the operand field (Figure 46). The label refers to the low-order position of the area. However, if the label is indented one place, that is, if it begins in column 7, the label will refer to the high-order position. A label is not mandatory.

Line	Label	Operation	
0 1	DOZEN	DS	12
0 2	FIVE	DS	5
0 3			

Figure 46. Defining Twelve-Position and Five-Position Areas in DS Statements

Figure 46 illustrates the form of the DS statement. The first entry, labeled DOZEN, defines an area twelve positions long. The second entry, labeled FIVE, defines an area five positions in length.

EQU — Equate

The EQU statement is used to define either a second symbol to reference a specific location, or a symbol for a location not previously labeled. The symbol to be defined is specified in the label field, and the representation of the location to be "equated" is specified in the operand field.

An EQU statement can be used to assign a symbolic label to each of the following:

Actual or symbolic address
Adjusted or modified address
Index register
Asterisk address

Actual or Symbolic Address

The symbol to be defined is specified in the label field. The operand field can contain an actual or symbolic address. If a symbolic address is specified in the operand field, it must have appeared as a label prior to this point in the subprogram. If this condition is not met, the label will *not* be defined.

SYMBOLIC ADDRESS

The EQU statement in Figure 47 will cause the processor to assign the same address to the label INDIVIDUAL that is assigned to the symbol MANNO. Thus, INDIVIDUAL has been equated to MANNO – both labels refer to the same core-storage location and are assigned the same relocation indicator by the processor.

Line	Label	Operation	
0 1	INDIVIDUAL	EQU	MANNO
0 2			

Figure 47. Equating a Symbolic Address

ACTUAL ADDRESS

The EQU statement in Figure 48 will cause the processor to assign the label ACCTNO to machine location 25000.

NOTE: Labels equated to actual addresses will be treated as absolute values and given a NO relocation indicator.

Line	Label	Operation	
0 1	ACCTNO	EQU	25000
0 2			

Figure 48. Equating an Actual Address

Adjusted or Modified Address

The operand of an EQU statement can be address adjusted or indexed. The same relocation indicators assigned to the address adjusted and/or indexed operand will be given to the defined label.

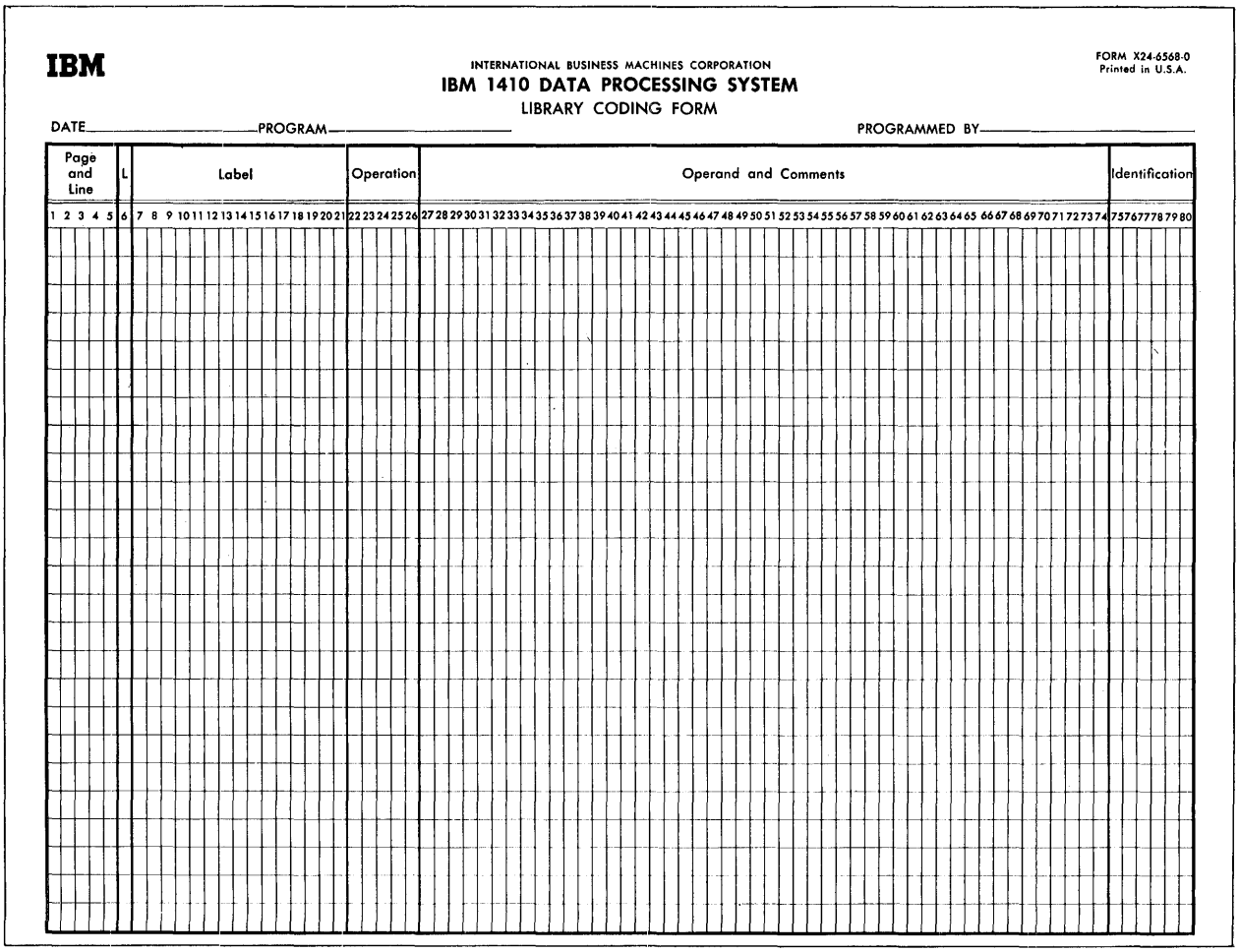


Figure 78. IBM 1410/7010 Library Coding Form

Library Entry

Page and Line	L	Label	Operation	Operand
			C	X001,X002
			BH	X00C
			BE	X00D
			BL	X00E

Macro-Instruction

Line	Label	Operation	Operand
0.1	ABCD	CHECK	PAR1,PAR2,PAR3,PAR4,PAR5
0.2			

Assembled Symbolic Program Entry

ABCD	C	PAR1,PAR2
	BH	PAR3
	BE	PAR4
	BL	PAR5

Figure 79. Macro Operations

Page and Line	L	Label	Operation	
		LABEL	BCE	A,B,C

Figure 80. Model Statement for a Complete Instruction

that a corresponding parameter from the macro-instruction operand field must be inserted in its place. This code is a □ followed by a number from 001 to 199, that indicates the position of the parameter in the macro-instruction. The macro-instruction in the source program will give the parameter entries to be inserted in the object routine. The model statement is illustrated in Figure 81.

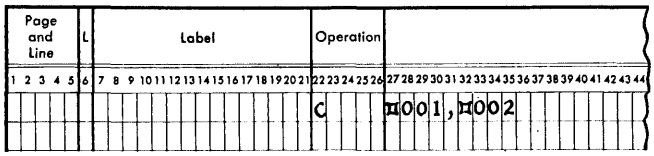


Figure 81. Model Statement for an Incomplete Instruction with Required Parameters

c. If the entry is incomplete, the programmer writes a □ followed by a number from 001 to 199 with AB bits over the units position (parameter 001 is □00A, parameter 2 is □00B, etc.). This indicates that the entry is to be included in the object routine only if the parameter is specified by the macro-instruction. For example, if parameter 003 does not appear in the macro-instruction, the instruction shown in Figure 82 will be deleted from the object routine.

NOTE: If parentheses are used, the programmer cannot use zoned switches in a MATH or BOOL statement.

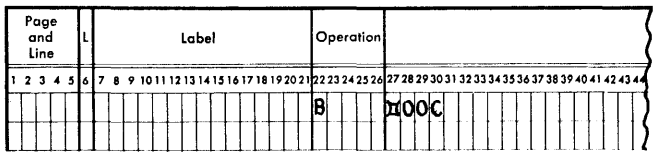


Figure 82. Model Statement for an Incomplete Instruction with Conditional Parameters

Labeling: If the model statement represents an instruction entry point for a branch instruction elsewhere in the program, it should have a label.

If additional external labels are required and specified as parameters in the macro-instruction they can be inserted in the label field of the symbolic program entry by using the □001-199 code.

The label of the macro-instruction causes the generation of an equate statement in the assembled object routine. The label is equated to an *, as shown in Figure 83.

Another example is shown in Figure 84.

Symbolic Addressing within the Library Routine: To allow a symbolic reference to other instructions in a library routine a □ followed by a number from 001 to 199 with a B bit over the units position (□00J = symbolic address 1, □00K = symbolic address 2, etc.) can be used. For example, the processor generates the symbolic address if the code □00J is used as a label for one entry and as an operand of at least one other entry in the same library routine.

Internal labels within flexible routines are generated in the form □nnnmmm, where nnn is the code (00J-09R), and mmm is the number of the macro within the source program. This is done to avoid duplicate address assignments for labels.

Example: Use the generated symbolic address of □00J as an operand for entry 3 and as the label for entry 6. UPDAT is the 23d macro encountered in the source program (Figure 85).

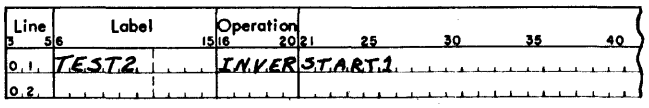
Address Adjustment and Indexing: The parameters in a macro-instruction and the operands in partially complete instructions in a library routine can have address adjustment and indexing.

If address adjustment is used in both the parameter and the instruction, the assembled instruction will be adjusted to the algebraic sum of the two. For example, if the address adjustment on one is +7 and the other is −4, the assembled instruction will have address adjustment equal to +3.

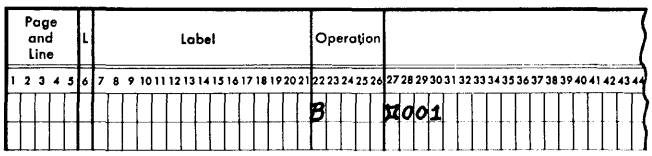
Model statement operands can be indexed. This indexing takes precedence over any indexing of a parameter supplied by a macro-instruction. The model statement index is used.

Literals: Operands of instructions in library routines may use literals as required. However, these literals may not contain the @ symbol within an alphameric literal.

Macro Instruction (Source Program)



Model Statement

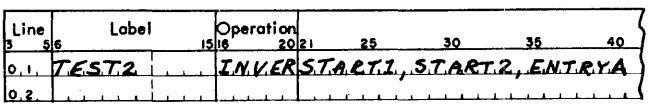


Assembled Symbolic Program Entry

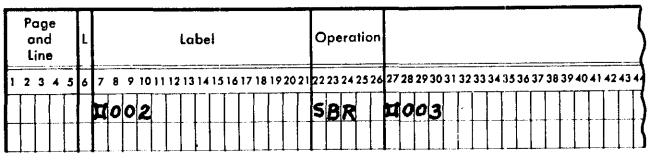
```
TEST2    EQU    *
         B      START1
```

Figure 83. Labeling

Macro Instruction (Source Program)



Model Statement



Assembled Symbolic Program Entry

```
TEST2    EQU    *
START2   SBR    ENTRYA
```

Figure 84. Additional External Labels

IBM **Technical Newsletter**

File Number	1410/7010-22
Re: Form No.	C28-0326-2
This Newsletter No.	N27-1267
Date	December 30, 1966
Previous Newsletter Nos.	N27-1223

IBM 1410/7010 AUTOCODER

This Technical Newsletter amends the publication IBM 1410/7010 Operating System; Autocoder, Form C28-0326-2, to include new information concerning the NOTE and MACRO statements, and to make other necessary changes and additions.

The attached replacement pages (9-10, 41-44, 47-48) should be substituted for the corresponding pages now in the publication. Text changes are indicated by a vertical line to the left of the affected text; figure changes are indicated by a bullet (•) to the left of the affected figure caption.

Please file this cover letter at the back of the publication. It provides a method of determining if all changes have been received and incorporated into the publication.

IBM Corporation, Programming Publications, Dept. 637, Neighborhood Road, Kingston, N.Y. 12401

PRINTED IN U.S.A.

as the reference address of COMMON during the assembly process. All relocatable addresses of data in COMMON are relative to 99999. For example, the 15th location downward in COMMON is assigned the value 99985, and appears as the same relative address in all subprograms. Labels referencing COMMON are assigned downward relocation indicators for absolute adjustment by the Linkage Loader.

Absolute adjustment involves changing the relative values of the labels (assigned to them by the processor) to absolute values in the relocated COMMON data area. The adjustment factor applied is the difference between the value of COMMON in the assembly process (99999) and the absolute value of COMMON determined by the Linkage Loader. Normally, the Linkage Loader will place COMMON at the location represented by the value of the system symbol /AMS/ (Absolute Memory Size). However, the programmer can specify a different absolute location for COMMON by means of a BASE2 statement. (The interested reader will find a fuller discussion of this subject in the publication, *System Monitor.*)

The steps necessary to use COMMON in a subprogram are discussed under "DAV – Define Area in COMMON," in the subsection "Declarative Operation Codes."

Processing Options

There are four processing options which can be exercised by the user:

1. He can suppress the printing of the assembly listing (on the Standard Print Unit).
2. He can suppress the punching of the object deck (on the Standard Punch Unit).
3. If there are no macro statements in the source deck, he can speed up the assembly process by indicating this fact.
4. He can suppress the diagnostic generation of an "M" flag for uses of index registers 14 and 15 when there is no true multiple definition. (See NOTE 1, under "Indexing with Address Adjustment.")

These options are indicated by means of additional parameters in the EXEQ card that calls the Autocoder processor.

The four parameters are:

NOPRT – Suppress printing
NOPCH – Suppress punching
NOMAC – No macros present
NOFLG – Suppress "M" flag

Any or all of these parameters may be used in the EXEQ card. They can appear in any order immediately following the EXEQ parameters required by the System Monitor. (See the publication, *System Monitor,* for details concerning the EXEQ card.)

Specification of parameters in the EXEQ card is concluded by the first blank encountered in the operand field. The following examples illustrate the format:

LABEL	OPERATION CODE	OPERAND
MON$$	EXEQ	AUTOCODER, SOF, SIU, NOPRT
MON$$	EXEQ	AUTOCODER, , , NOMAC, NOPCH, NOFLG
MON$$	EXEQ	AUTOCODER, , , NOFLG, NOPRT, NOMAC, NOPCH

Autocoder Multiple Compilation

Autocoder can compile any number of programs with a single MON$$ EXEQ AUTOCODER card. The output is the same as if it were produced by several separate compilations.

Input for a multiple compilation consists of the MON$$ EXEQ AUTOCODER card followed by the source decks to be compiled. No control cards are necessary between the END statement of one program and the first card of the next program if the programmer wants the subsequent compilation to receive standard treatment; that is, printing, punching, and normal macro and flag processing.

A different set of processing options (NOPRT, NOPCH, NOMAC or NOFLG) can be specified for an ensuing program in a multiple compilation by placing an Option card after the preceding END statement. This card has the same requirements and options as the MON$$ EXEQ AUTOCODER card except that the label and operation fields, card columns 6-20, must contain blanks (instead of MON$$ EXEQ). The processing options specified in this Option card will be applied until the next Autocoder END card is read by the processor.

Autocoder multiple compilation has two potential advantages:

1. It enables the programmer to process a series of source decks from the Alternate Input Unit as well as the Standard Input Unit.
2. It bypasses the monitor processing which normally is necessary between compilations.

Terminating the Object Program

The object progam must terminate execution by means of one of the following instructions:

B	/EOP/	Normal End of Program
B	/UEP/	Unusual End of Program

Both forms of termination are shown in Figure 2. Full details can be found in the publication, *System Monitor.*

```
     64015                    SAMPLE SUBPROGRAM USING THE 1410/7010 AUTOCODER                PAGE    1              SAMPL
SEQNO  PGLIN  LABEL       OPCOD  OPERAND                                    REL   CT  ADDRS  INSTRUCTION      CARD  FLAG

   1   AA020              TITLE  SEQUENCE                                                                        001
   2   AA030  * THIS SUBPROGRAM CHECKS THE SEQUENCE OF THE PGLN/ FIELD
   3 S AA040  * IF THE PGLN/ FIELD IS 99999, THE PROGRAM IS TERMINATED NORMALLY
   4   AA050  * A NON-ASCENDING SEQUENCE RESULTS IN AN UNUSUAL END OF PROGRAM.
   5   AA060  SEQROUTINE  SBR    EXITSEQRT&5                                 ¤     7  00000  G 00056       B   002
   6   AA070              C      PGLN/,@99999@  IS THIS THE LAST ENTRY       1    11  00007  C PGLN/ 00153       002
   7   AA080              BE     ENDOFJOB       YES                          ¤     7  00018  J 00058       S   002
   8   AA090              NOPWM                                              9     1  00025  N                    002
   9   AA100              B      CHECKSEQ                                    ¤     7  00026  J 00101            002
  10   AA110              SW     *-12           SET FIRST TIME NOP SWITCH TO BRANCH  @  6  00033  , 00026        002
  11   AA120              MLCWB  PGLN/,PGLNHOLD#5                            A L  12  00039  D PGLN/ 00158 P   003
  12   AA130  EXITSEQRT   B      0              EXIT - RETURN TO MAIN PROGRAM T     7  00051  J 00000            003
  13   AA135  *
  14   AA140  ENDOFJOB    IOCTL  TYPE,MESSAGE   NOTIFY OPERATOR OF END OF JOB
  15 G AA140  ENDOFJOB    EQU    *                                                        00058
  16 G 01510              BZN    *-11,/CTB/                                  C L  12  00058  V 00058 /CTB/ 2  003
  17 G 01520              BXPA   /CNC/                                       T     7  00070  Y /CNC/        X   003
  18 G 01530              DCW    MESSAGE                                     N     5  00081    00126            003
  19 G 01580              BZN    *-11,/CTB/                                  C L  12  00082  V 00082 /CTB/ 2  004
  20   AA145              B      /EOP/          NORMAL END OF PROGRAM        T     7  00094  J /EOP/            004
  21   AA148  *
  22   AA150  CHECKSEQ    C      PGLNHOLD, PGLN/                             1    11  00101  C PGLN/ 00158       004
  23   AA160              BH     EXITSEQRT-12   BRANCH IF PGLN/ IS IN SEQUENCE  ¤ L  7  00112  J 00039       U   004
  24   AA170              B      /UEP/          UNUSUAL END OF PROGRAM       T     7  00119  J /UEP/            004
  25   AA175  *


     64015                    SAMPLE SUBPROGRAM USING THE 1410/7010 AUTOCODER                PAGE    2              SAMPL
SEQNO  PGLIN  LABEL       OPCOD  OPERAND                                    REL   CT  ADDRS  INSTRUCTION      CARD  FLAG

  26   AA180  SEQR/       DEFIN  SEQROUTINE      SEQR/  LINKAGE SYMBOL FOR SUBPROGRAM                 00000        005
  27   AA185   MESSAGE    DCW    @END OF JOB@,G  CONSOL PRINTER NOTICE            11  00126                       006
  28   AA190              HALT   12345           EXAMPLE OF AN ERRONEOUS STATEMENT  A  12  00137  N 12345 .....  007  D
  29   AA200              END
  30                      @99999@                                               5  00153                       008
  31           PGLNHOLD   #0005                                                 5  00158                       008
                                          NUMBER OF FLAGGED STATEMENTS     1
  28

                                    1410/7010 AUTOCODER...SYSTEM /MID/ 0001
```

●Figure 2. A Page from an Assembly Listing

Assembly Listing

Each page of the assembly listing contains a page heading line and a column heading line.

The page heading line contains the following information, from left to right:

1. The date contained at location /DAT/ (the system symbol for the five-position date field in the Resident Monitor)
2. Information supplied via HEADR card
3. Page number in the listing
4. The identification supplied by HEADR or RESEQ cards

The column heading line is illustrated in Figure 2, which shows the assembly listing of a subprogram assembled by the 1410/7010 Autocoder processor. The subprogram contains a deliberate error contrived to exhibit Autocoder's diagnostic flagging system. Figure 2 illustrates the following items, going from left to right in the column heading line:

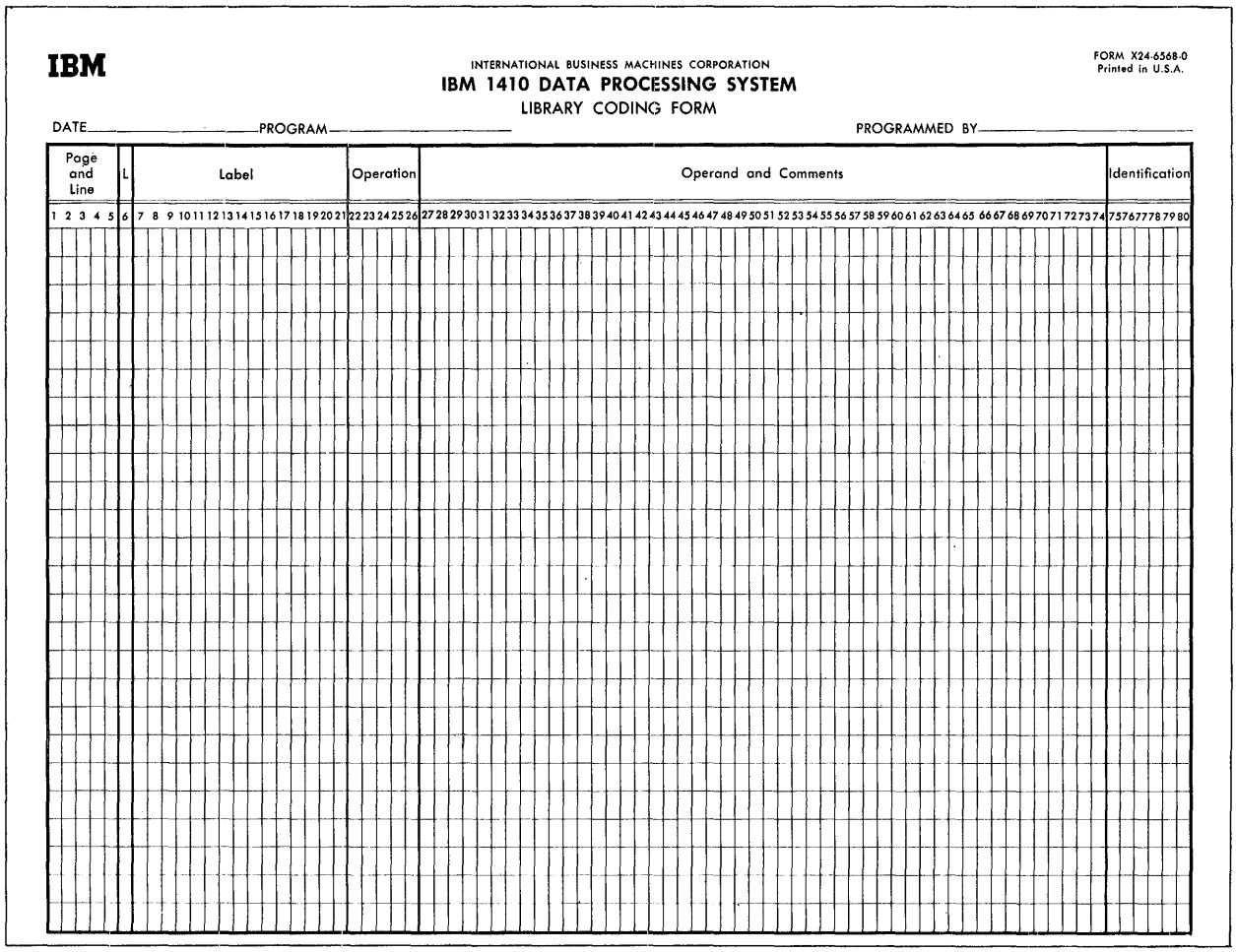


Figure 78. IBM 1410/7010 Library Coding Form

Library Entry

Page and Line	L	Label	Operation	Operand
			C	M001,M002
			BH	M00C
			BE	M00D
			BL	M00E

Macro-Instruction

Line	Label	Operation	OPE
0.1	ABCD	CHECK	PAR1,PAR2,PAR3,PAR4,PAR5
0.2			

Assembled Symbolic Program Entry

ABCD	C	PAR1,PAR2
	BH	PAR3
	BE	PAR4
	BL	PAR5

Figure 79. Macro Operations

Page and Line	L	Label	Operation	
		LABEL	BCE	A,B,C

Figure 80. Model Statement for a Complete Instruction

that a corresponding parameter from the macro-instruction operand field must be inserted in its place. This code is a □ followed by a number from 001 to 199, that indicates the position of the parameter in the macro-instruction. The macro-instruction in the source program will give the parameter entries to be inserted in the object routine. The model statement is illustrated in Figure 81.

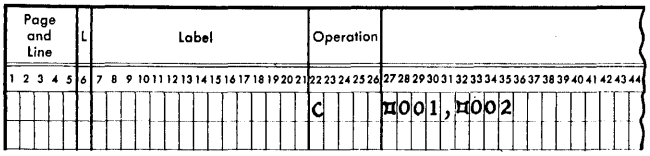


Figure 81. Model Statement for an Incomplete Instruction with Required Parameters

c. If the entry is incomplete, the programmer writes a □ followed by a number from 001 to 199 with AB bits over the units position (parameter 001 is □00A, parameter 2 is □00B, etc.). This indicates that the entry is to be included in the object routine only if the parameter is specified by the macro-instruction. For example, if parameter 003 does not appear in the macro-instruction, the instruction shown in Figure 82 will be deleted from the object routine.

NOTE: If parentheses are used, the programmer cannot use zoned switches in a MATH or BOOL statement.

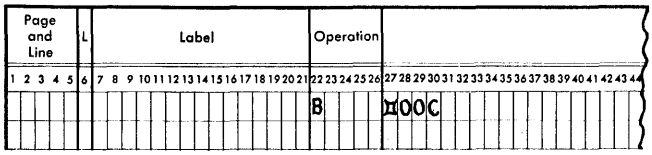


Figure 82. Model Statement for an Incomplete Instruction with Conditional Parameters

Labeling: If the model statement represents an instruction entry point for a branch instruction elsewhere in the program, it should have a label.

If additional external labels are required and specified as parameters in the macro-instruction they can be inserted in the label field of the symbolic program entry by using the □001-199 code.

The label of the macro-instruction causes the generation of an equate statement in the assembled object routine. The label is equated to an *, as shown in Figure 83.

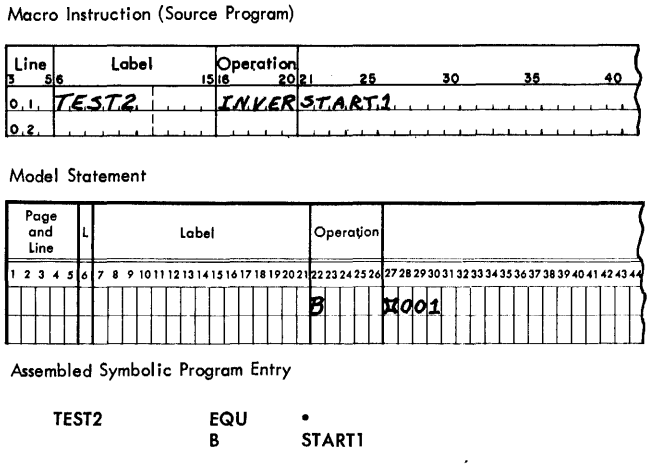


Figure 83. Labeling

Another example is shown in Figure 84.

Symbolic Addressing within the Library Routine: To allow a symbolic reference to other instructions in a library routine a □followed by a number from 001 to 199 with a B bit over the units position (□00J = symbolic address 1, □00K = symbolic address 2, etc.) can be used. For example, the processor generates the symbolic address if the code □00J is used as a label for one entry and as an operand of at least one other entry in the same library routine.

Internal labels within flexible routines are generated in the form □nnnmmm, where nnn is the code (00J-09R), and mmm is the number of the macro within the source program. This is done to avoid duplicate address assignments for labels.

Example: Use the generated symbolic address of □00J as an operand for entry 3 and as the label for entry 6. UPDAT is the 23d macro encountered in the source program (Figure 85).

Address Adjustment and Indexing: The parameters in a macro-instruction and the operands in partially complete instructions in a library routine can have address adjustment and indexing.

If address adjustment is used in both the parameter and the instruction, the assembled instruction will be adjusted to the algebraic sum of the two. For example, if the address adjustment on one is +7 and the other is −4, the assembled instruction will have address adjustment equal to +3.

Model statement operands can be indexed. This indexing takes precedence over any indexing of a parameter supplied by a macro-instruction. The model statement index is used.

Literals: Operands of instructions in library routines may use literals as required. However, these literals may not contain the @ symbol within an alphameric literal.

NOTE: Area defining literals and area defining constants cannot be used in a MACRO statement.

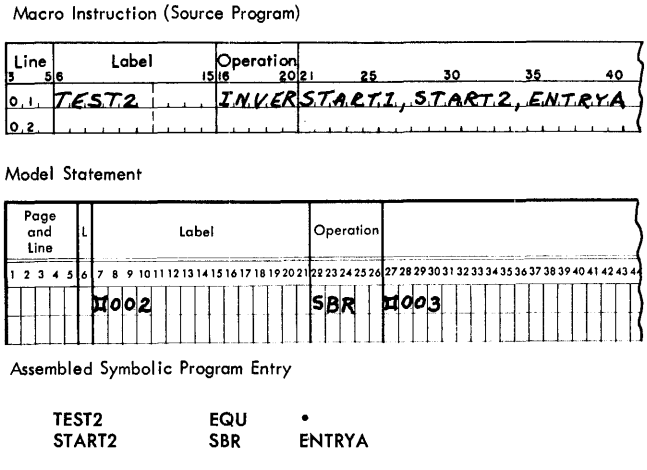
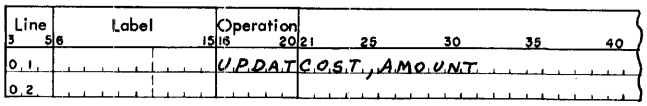
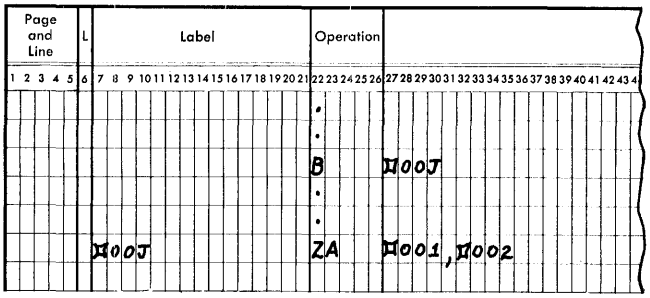


Figure 84. Additional External Labels

Macro Instruction (Source Program)



Model Statement



Assembled Symbolic Program Entry

```
                .
                .
                B       □00J023
                .
                .
□00J023         ZA      COST,AMOUNT
```

Figure 85. Internal Labels

NOTE 1: A model statement in the library routine for a macro-instruction may not be another macro-instruction.

NOTE 2: END statements cannot be used in library routines.

NOTE 3: A comments card can be included in the model statements. It must be written with an asterisk in column 7.

The Processor enters model statements in the library tape immediately following the header statement during System Generation.

Result: Any library routine can be extracted by writing the associated macro-instruction in the source program.

Figure 86 is a summary of the codes that can be used in the model statements of library routines.

CODE	POSITION	FUNCTION
□001 - □199	Statement	Substitute parameter (parameter must be present)
□00A - □191	Statement	Substitute parameter (if parameter is missing, delete statement)
□00J - □19R	Label Field and Operand Field	Assign internal label

Figure 86. Model Statement Codes

General Description: A macro-instruction is the entry in the source program that causes a series of instructions to be inserted in a program.

The Programmer:

1. Writes the name of the library routine in the operation field.

2. Writes the label that is to reference the first assembled model statement. A LABEL EQU * is generated to do this.

3. Writes the parameters that are required for the particular object routine desired. These parameters, used by the model statements, are written as follows:

   a. Parameters must be written in the sequence in which they are to be used by the codes in the model statements. For example, if cost is parameter 001, it must be written first so that it will be substituted wherever a □001 or □00A appears as a label, operation code, or operand of a model statement.

   b. As many parameters may be used as can be contained in the operand fields of five or fewer coding sheet lines. If more than one line is needed for a macro-instruction, the label and operation fields of the additional lines must be left blank. Parameters must be separated by a comma. They cannot contain blanks or commas unless they appear between @ symbols. The @ symbol itself cannot appear between @ symbols. Also, the @ symbol can be used only in pairs as a literal identifier. It cannot be used in any other way; e.g., a single @ symbol could not be used to represent the d modifier of a macro-instruction. If parameters for a single macro-instruction require more than one coding sheet line, the last parameter in each line must be followed immediately by a comma. The last parameter in a macro-instruction should not be followed by a comma.

   c. Parameters that are not required for the particular object routine desired can be omitted from the operand field of the macro-instruction. However, if a parameter is omitted, the comma that would have followed the parameter must be included, unless the omitted parameter is behind the last parameter which is included in the macro-instruction. These commas are necessary to count parameters up to the last included parameter. All parameters between the last included parameter and parameter 199 are assumed by the processor to be absent.

Figures 87, 88, 89 and 90 show how parameters can be omitted. The hypothetical macro-instruction called EXACT is used. EXACT can have as many as nine parameters.

The Processor extracts the library routine and selects the model statements required for the object routine as specified by the parameters in the macro-instructions, and by substitution and switches set by BOOL or COMP in the model statements.

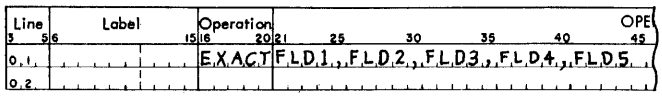


Figure 87. Parameter for EXACT. 006-199 Missing

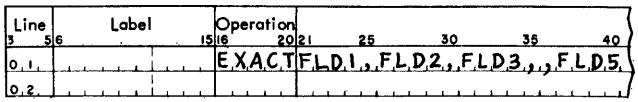


Figure 88. Parameters 004 and 006-199 Missing

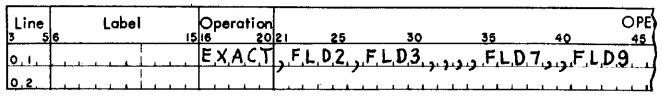


Figure 89. Parameters 001, 004-006, 008 and 010-199 Missing

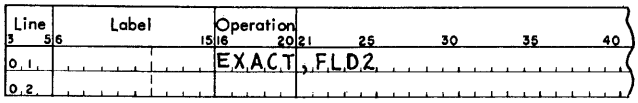


Figure 90. Parameters 001 and 003-199 Missing

Result: The resulting program entries are merged with the source program entries behind the macro-instruction.

Pseudo-Macro Instructions

These statements never appear in a user's source program or in the output listing of an assembled Autocoder program. However, they are used in library routines to signal the processor that certain conditions exist which can affect the assembly of an object routine. For example, the presence of a pseudo-macro-instruction in a library routine can cause a group of model statements to be deleted. Thus, pseudo-macros provide the writer of library routines with a coding flexibility which exceeds the limitations of the substitution and condition codes described previously.

Pseudo-macro-instructions may be written anywhere in a library routine. The five pseudo-macros incorporated in the Autocoder processor are MATH, BOOL, COMP, NOTE, and MEND.

Permanent and Temporary Switches

The MATH, BOOL, and COMP pseudo-macros use internal indicators (switches) to signal the processor of existing status conditions.

There are 099 permanent and 199 temporary switches available for recording status conditions. Each switch occupies one core-storage position during the macro generator phase of Autocoder. If a storage position contains the character A (AB 1 bits), the switch is ON; if it contains a ? (CAB 82 bits), the switch is OFF. At the beginning of assembly all switches are OFF.

Permanent Switches: Permanent switches retain status conditions during the entire macro generator phase unless changed by a pseudo-macro. They are addressed by using a # symbol followed by the three-digit number of the switch to be set or tested. For example, #001 addresses permanent switch 001; #002 addresses switch 002; and #099 addresses switch 099.

Temporary Switches: When the processor encounters a macro-instruction, the temporary switches are set to the condition (presence or absence) of the parameters in the operand of the macro field. If the parameter is present, the corresponding switch is set ON. If the parameter is missing, the switch is set OFF. For example, if parameter 001 is present, temporary switch 001 is turned ON. If parameter 002 is missing from the macro-instruction, temporary switch 002 is OFF. Temporary switches retain status throughout the processing of a macro-instruction unless changed by a pseudo-macro. After the macro-instruction has been completely processed, all temporary switches are set OFF. Temporary switches are addressed by using a □ symbol followed by the three-digit number of the switch to be set or tested. For example, □001 addresses temporary switch 001; □002 addresses switch 002; and □199 addresses switch 199.

If a macro with a maximum of nine parameters is encountered, the processor sets the first nine temporary switches to indicate the presence or absence of these nine parameters. Temporary switches 010-199, which are OFF, can be used by the pseudo-macros to communicate conditions to the processor while it is working on this particular macro-instruction. This use of temporary switches is recommended because it reserves the permanent switches for communicating information from one macro to another.

MATH — For Solving Algebraic Expressions

A MATH pseudo-macro contains as operands: sum boxes, arithmetic expressions, and sign switches.

Sum Boxes: A sum box is a group of five core-storage positions used to store the result of an arithmetic expression. Autocoder makes available 20 such sum boxes. A sum box is addressed by using a # symbol followed by the three-digit number (ending in zero or five) of the sum box to be referenced. For example, the address of the first sum box is #005; the address of the second sum box is #010; and the address of the twentieth sum box is #000.

At the beginning of the macro phase, a sum box contains 00000. Any number may be placed in a sum

is OFF, the statement is false. Therefore, set temporary switch 015 OFF and skip to statement labeled L.

The example shown in Figure 97 states:

1. If both temporary switches 001 and 002 or both temporary switches 003 and 004 are ON, the statement is true. Therefore, set temporary switch 015 ON.
2. However, if either temporary switch 001 or 002 and either temporary switch 003 or 004 is OFF, the statement is false. Therefore, set temporary switch 015 OFF and skip to the model statement whose label is L.

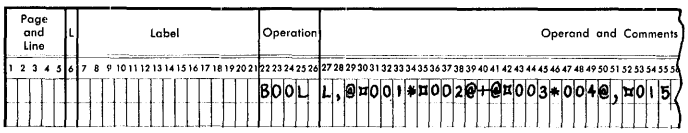


Figure 97. BOOL Pseudo-Macro

SWITCHES

CONDITIONS	001	*	002	+	003	*	004	=	LOGICAL VALUE
	OFF 0	*	OFF 0	+	OFF 0	*	OFF 0	=	0
	ON 1	*	OFF 0	+	OFF 0	*	OFF 0	=	0
	OFF 0	*	ON 1	+	OFF 0	*	OFF 0	=	0
	OFF 0	*	OFF 0	+	ON 1	*	OFF 0	=	0
	OFF 0	*	OFF 0	+	OFF 0	*	ON 1	=	0
	OFF 0	*	ON 1	+	OFF 0	*	ON 1	=	0
	ON 1	*	OFF 0	+	ON 1	*	OFF 0	=	0
	OFF 0	*	ON 1	+	ON 1	*	OFF 0	=	0
	ON 1	*	OFF 0	+	OFF 0	*	ON 1	=	0

FALSE

Figure 99. False Conditions

Figure 98 is a table showing all conditions that will cause the BOOL statement shown in Figure 97 to be true.

Figure 99 is a table showing all conditions that will cause the BOOL statement shown in Figure 97 to be false.

COMP — To Compare Two Fields

General Description: The COMP pseudo-macro compares an A-field to a B-field (maximum of 15 characters), and sets permanent or temporary switches to indicate the result of the comparison.

The Programmer:

1. Writes the name of the pseudo-macro (COMP) in the operation field.

SWITCHES

CONDITIONS	001	*	002	+	003	*	004	LOGICAL VALUE
	ON 1	*	ON 1	+	OFF 0	*	OFF 0	= 1
	OFF 0	*	OFF 0	+	ON 1	*	ON 1	= 1
	ON 1	*	ON 1	+	ON 1	*	ON 1	= 1
	ON 1	*	ON 1	+	ON 1	*	OFF 0	= 1
	OFF 0	*	ON 1	+	ON 1	*	ON 1	= 1
	ON 1	*	ON 1	+	OFF 0	*	ON 1	= 1
	ON 1	*	OFF 0	+	ON 1	*	ON 1	= 1

TRUE

Figure 98. True Conditions

2. Writes the operand field in the format shown in Figure 100. The first and second entries are the A- and B-fields. The A- and B-fields may be any of the parameters 001-199, sum boxes #005-#000, or literals. They cannot be switches.

NOTE 1: For the COMP pseudo-macro, alphameric literals are not enclosed by @ symbols. Entries 3, 4, and 5 are high, equal, and low switches.

NOTE 2: The codes for the two fields to be compared must be present in all COMP pseudo-macro-instructions. Codes for the switches may be omitted if they are not needed to store the result of the compare operation. However, if a switch is omitted, the comma that would have followed it must be included in the operand field.

NOTE 3: B-field controls compare. (High-order position of B-field ends compare.)

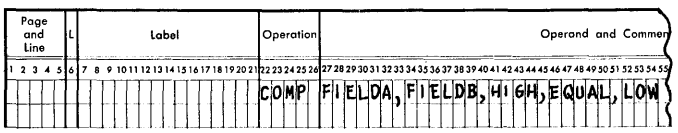


Figure 100. Format for COMP Pseudo-Macro

The Processor:

1. Compares the A-field to the B-field.
2. Sets one status switch ON and two switches OFF to reflect the result of the comparison.
   a. The first switch is set ON, if the value of the B-field is greater than that of the A-field.

b. The second switch is set ON, if the B-field is equal to the A-field.
c. The third switch is set ON, if the value of the B-field is less than that of the A-field.

Examples: Figure 101 shows a COMP pseudo-macro which states:

1. Compare parameter 002 of the macro statement to WORKAREA.
2. If parameter 002 is equal to WORKAREA, turn on temporary switch 25.
3. If WORKAREA is less than parameter 002, turn on temporary switch 26.

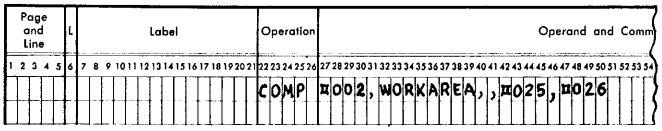


Figure 101. COMP Pseudo-Macro

Figure 102 shows a COMP pseudo-macro which states:

1. Compare the contents of sum box 005 to parameter 003 of the macro statement.
2. If the result is HIGH, set temporary switch 024 ON.
3. If the result is EQUAL, set temporary switch 025 ON.
4. If the result is LOW, set temporary switch 026 ON.

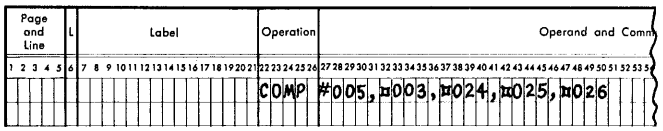


Figure 102. Comparing a Parameter to the Contents of a Sum Box

NOTE: Standard 1410/7010 collating sequence determines HIGH, EQUAL, or LOW conditions. Comparisons are controlled by the B-field. Thus, the statement shown in Figure 103 will cause temporary switch 025 to be set ON if the low-order position of parameter 002 is a 3.

NOTE—To Produce a Message

General Description: The NOTE pseudo-macro is used to write messages concerning conditions that can arise during the processing of a macro-instruction.

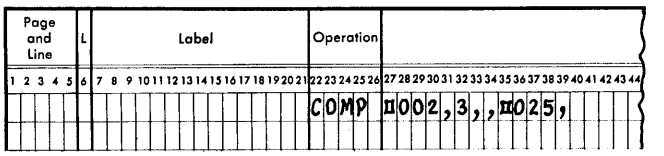


Figure 103. Checking for a Single Character

The message is printed in line on the output device (tape or on-line printer). In addition, an "N" will be automatically inserted in the flag field of the assembly listing.

The Programmer:

1. Writes the name of the pseudo-macro (NOTE) in the operation field.
2. Writes the message in the operand field.

NOTE: Two successive blanks terminate the operand of a NOTE statement.

The Processor: Prints the message on the Standard Print Unit (tape or on-line printer).

Examples: Figure 104 shows how the NOTE pseudo-macro can be used in combination with the BOOL pseudo-macro. The BOOL pseudo-macro tests to ensure that parameters 001 and 002 are present in the macro-instruction. If either parameter is missing, the processor skips to the NOTE pseudo-macro and prints:

PARAMETER ABSENT FROM MACRO.

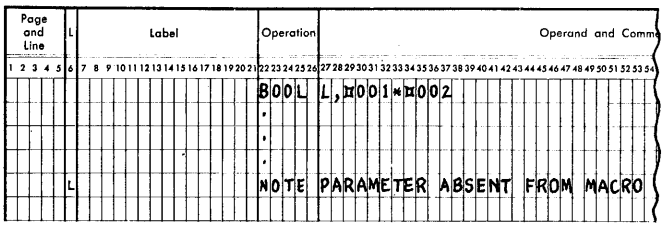


Figure 104. NOTE Pseudo-Macro

MEND — End of Routine

General Description: This pseudo-macro signals the end of generation for a macro-instruction. It may appear anywhere in a library routine.

The Programmer:

1. Writes the name of the pseudo-macro (MEND) in the operation field.
2. Leaves the operand field blank.

The Processor: Stops processing the macro-instruction when it encounters a MEND statement. Figure 105 shows a MEND pseudo-macro.

NOTE: A BOOL pseudo-macro can be used to skip over a MEND pseudo-macro which appears within the library routine if conditions indicate that more model statements must be processed.

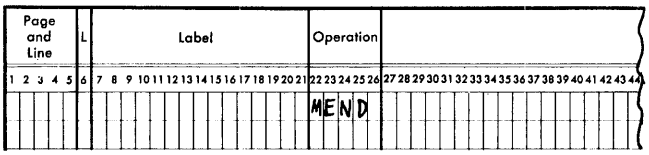


Figure 105. MEND Pseudo-Macro

Printed in U. S. A. C28-0326-2

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601