File Number GENL-22
Form C24-3258-2

IBM **Systems Reference Library**

Autocoder (on Disk) Language Specifications IBM 1401, 1440, and 1460

This reference publication describes the Disk Autocoder programming system for IBM 1401, 1440, and 1460. The first section contains the specifications of the symbolic language of Autocoder (mnemonics, labels, address types), a description of declarative, imperative, and assembler control operations, and the rules for writing the source program. The second section describes macro operations and macro instructions. Reference charts that list all valid Autocoder mnemonics also are included.

For a list of other publications and abstracts, see the IBM *Bibliography* for the associated data processing system.

Major Revision, April 1966

This publication, C24-3258-2, is a major revision of, and obsoletes C24-3258-1 and Technical Newsletter N21-0038.

Revisions to the text are indicated by a vertical line to the left of the changed text; revisions to the figures are indicated by a bullet (•) to the left of the figure caption.

Copies of this and other IBM publications can be obtained through IBM Branch Offices. Address comments concerning the content of this publication to:
IBM, Product Publications Department, Rochester, Minn. 55901.

Contents

The Disk Autocoder system is designed to simplify the programmer's task. Instead of coding program statements in machine language, he can write symbolic statements that comprise an Autocoder source program. The source program is input to an assembler program, which is supplied by IBM, that translates the source statements into machine language and produces an object program.

The Disk Autocoder language includes the following significant features:

- Mnemonic operation codes that are more easily remembered than the actual machine-language operation codes.
- Symbolic operands that eliminate actual core-storage address assignment and reference.
- Literal operands that eliminate prior definition of actual constants.
- Area-definition statements that allocate core storage for constants and work areas.
- Assembler-control statements that allow the programmer to exercise some control over the assembly process.
- A macro facility that eliminates repetitive coding of general routines. By writing a single instruction (macro instruction), the programmer can specify that a routine be extracted from the Autocoder library and incorporated in his program.

Machine Requirements

The Disk Autocoder system requires the following minimum machine configurations.

IBM 1401 System

- 4,000 positions of core storage
- High-Low-Equal Compare Feature
- One IBM 1311 Disk Storage Drive
- One IBM 1402 Card Read-Punch
- One IBM 1403 Printer

IBM 1440 System

- 4,000 positions of core storage
- One IBM 1301 Disk Storage or one IBM 1311 Disk Storage Drive
- One IBM 1442 Card Reader
- One IBM 1443 or 1403 Printer

IBM 1460 System

- 8,000 positions of core storage
- One IBM 1301 Disk Storage or one IBM 1311 Disk Storage Drive
- One IBM 1402 Card Read-Punch
- One IBM 1403 Printer

The Autocoder System can utilize the following devices and features if available:

- IBM 1444 Card Punch
- IBM 1404 Printer
- Console Printer
- 8,000, 12,000, or 16,000 positions of core storage
- Print Storage feature
- Direct Seek feature (for a library change only)

The system on which the object program is to be executed must have:

- A card reader or a disk unit to load the object program.
- Sufficient core storage to contain the object program. If the object program requires more than the available core storage, the program must be executed in sections (overlays) or the job must be divided into multiple runs.
- The devices and special features specified in the object program.
- The high-low-equal compare feature, if the MLTPY macro, the DIVID macro, or the clear option (, C) of the DA statement is in the program.

Related Information

One of the following SRL publications should be used in conjunction with the Autocoder language specifications:

System Operation Reference Manual for IBM 1401 and 1460, Form A24-3067.

System Operation Reference Manual for IBM 1440, Form A24-3116.

Programming with Autocoder

Source Program

The source program consists of statements written in symbolic language. Disk Autocoder symbolic language permits the programmer to define areas, write instructions, call in library routines, and exercise some control over assembler operations.

The Disk Autocoder language includes a standard set of mnemonic operation codes for declarative, imperative, and assembler control operations.

The mnemonics used in imperative statements are more easily remembered than the machine-language operation codes because they are usually abbreviations for the actual instruction. For example:

Instruction	*Mnemonic*	*Machine-Language Code*
Multiply	M	@
Clear word mark	CW	⌑

The mnemonics used in declarative and assembler control statements have no machine language equivalent.

Source-program statements are written using mnemonic operation codes and the names given to data, instructions, and constants. Literals (actual data to be operated on during processing) can also be written in the instruction statements that use them.

The information contained in Autocoder statements is divided into four categories:

1. *Area definition (declarative operations).* The area-definition entries are used to assign sections of storage for fixed data (constants) that will be needed during processing, to set aside work areas, and to assign symbolic names to data, devices, and areas used in the program.
2. *Instructions (imperative operations).* The instruction entries state, symbolically, the operations to be performed by the object machine. ADD, SUBTRACT, READ, and PUNCH are examples of imperative operations.
3. *Control Statements (assembler control operations).* The disk Autocoder system permits the programmer to exercise some control over the assembly process. For example, the programmer can specify the beginning address of the object program and the core-storage capacity of the object machine.
4. *Macro Instructions (macro operations).* Macro instructions are used to call out standard sets of instructions (routines) from the library that is stored on disk. During program assembly, the assembler can extract the routine associated with the macro instruction, tailor it to fit the program requirements, and insert it in the object program.

Assembler

The Autocoder Assembler Program operates under the direction of a System Control Program. The functions of this control program are to coordinate system functions and to handle input/output device assignments.

The Autocoder Assembler is a multiphase program designed to translate Autocoder statements into machine language. At assembly time, the source program is read into core storage from cards or disk. The System Control Program reads the Assembler Program into core storage from the disk unit that contains the Autocoder system.

The first step in the translation process is performed by the macro-generator phases of the Assembler Program. These phases examine source-program macro instructions, extract the associated library routines, and generate Autocoder statements.

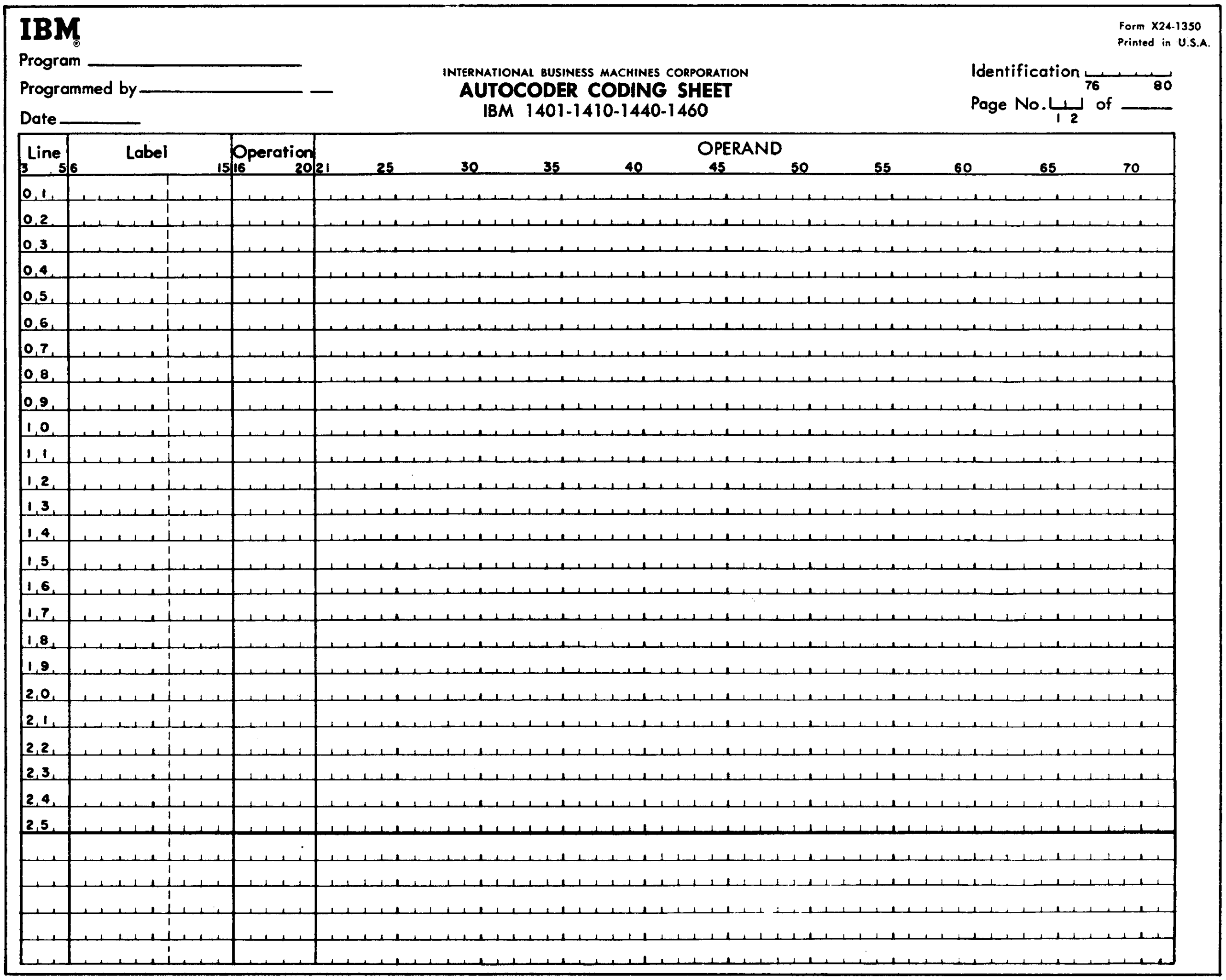
The Assembler then analyzes all Autocoder statements during a diagnostic phase. A diagnostic listing of all invalid statements is printed if the user specifies the option in his control card for assembly (CTL card). A programmed halt occurs after the diagnostics have been printed. The user can make corrections and restart the assembly, or he can continue processing.

After the macro instructions have been processed and the Autocoder statements have been analyzed, the Assembler translates the Autocoder statements into a machine-language object program. The object program is punched into cards or written in disk storage, depending on the specifications in the user's control cards.

Coding Sheet

Disk Autocoder statements are written on a coding sheet that is designed to organize them into the format required by the assembler. Figure 1 shows the Disk Autocoder free-form coding sheet.

Although the assembler can process statements coded in 1401 Symbolic Programming System (SPS) and 1440 Basic Autocoder lanuguages (see *ENT—Enter New Coding Mode*), this publication refers primarily to the coding of Disk Autocoder language.



IBM

Program

Programmed by

Date

INTERNATIONAL BUSINESS MACHINES CORPORATION

AUTOCODER CODING SHEET

IBM 1401-1410-1440-1460

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

Identification 76 80

Page No. 1 2 of

Line 3 5	Label 6 15	Operation 16 20	OPERAND 21 25 30 35 40 45 50 55 60 65 70
01			
02			
03			
04			
05			
06			
07			
08			
09			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Figure 1. Autocoder Coding Sheet

Write all source-program statements and comments on the coding sheet. Column numbers on the sheet indicate the card-punching format for all the cards in the source deck. Punch each line of the coding sheet into a separate card.

The function of each portion of the coding sheet is explained here.

Page Number (Columns 1 and 2)

Write the sequence number of the coding sheet in this field. These numbers should be arranged according to low-to-high collating sequence. Any character that is valid for IBM 1400 series systems can be used. Refer to the IBM character-code chart in the *System Operation Reference Manual.*

Note: Do not use an asterisk in column 1.

Line Number (Columns 3-5)

Use this field to indicate the sequence of entries on the coding sheet. The units position of this field may be left blank. It can be used later to indicate the sequence of inserts on a page. The five unnumbered lines at the bottom of a page can contain these inserts.

For example, number the first insert between lines 02 and 03 "025". Number later inserts at that point so that they can precede or follow the first insert, as required. Numbers on the coding sheet need not be consecutive; but, when the source deck is used as input to the assembler, the cards should be in low-to-high collating sequence.

Inserts can affect address adjustment. An insert may make it necessary to change the adjustment factor in other entries. See *Address Adjustment.*

Label (Columns 6-15)

A label can have as many as six alphameric (A-Z or 1-9) characters, but the first character must be alphabetic. Special characters and blanks must not be used within a label.

The label usually starts in column six. See *Define Constant with Word Mark* for exception. Any subsequent references to the labeled item must correspond to the name used in the label field of that particular item.

Columns 13-15 are not checked.

Operation (Columns 16-20)

Write the mnemonic or machine-language operation code in this field.

Operand (Columns 21-72)

Two operands and a d-character may be written in this field. An operand designates a core-storage address, an input/output unit, or a constant to be defined. A d-character modifies an operation code. It is a single alphabetic, numeric, or special character.

The operands and the d-character must be separated by commas because the Disk Autocoder coding sheet is free-form (the operand and d-character fields are not fixed fields).

Comments

The programmer can include a remark anywhere in the operand field if he leaves at least two non-significant blank spaces between it and the operand.

To include a whole line of information anywhere in the program, write a comments line that contains an asterisk in column 6 and the comment in columns 7-72. Columns 6-8 should not contain *1*. (*1* in columns 6-8 will cause a diagnostic to falsely appear during assembly.) A punched card containing a comments line is called a comments card. The information punched in the comments card appears in the symbolic-program listing produced by the assembler, but it does not affect the object program in any way.

Columns 73-75 are not checked.

Identification (Columns 76-80)

Write an identification name or number in this field to identify a program or program section (overlay). Punch the contents of this field into each card in the source deck. The identification appears on the symbolic-program listing but does not affect the object program in any way.

Writing Autocoder Statements

Three types of information can be specified in Autocoder statements: labels, operation codes, and operands.

Labels

Labels are descriptive terms selected to identify a specific area or instruction in a source-program statement. A label that suggests the meaning of the area or instruction makes coding easier. It also makes the program more easily understood by others. For example:

Type of Statement	*Meaning*	*Label*
Area Definition	Withholding Tax	WHTAX
Instruction	Update	UPDATE

When the assembler processes a source-program statement, it assigns an address and allocates storage for the instruction or defined area. If the statement has a label, the assembler equates the label to the assigned address. In this publication the assigned address is called the *equivalent address.*

The equivalent address of the label for an *instruction* is the leftmost (high-order) core-storage position of the area the assembler has allocated for it. For example, an instruction whose label is ENTRYC is located in core-storage locations 549-552. The equivalent address of ENTRYC is then 549.

The equivalent address of the label of an area-definition statement is usually the rightmost (low-order) core-storage position of the area the assembler has allocated for the constant or work area. (See *DCW — Define Constant with Word Mark* and *DC — Define Constant (No Word Mark)* for exceptions.) For example, in a DCW statement a constant whose label is RATE is located in core-storage positions 420-424. The equivalent address of RATE is then 424.

During processing the assembler maintains a table of labels and their equivalent addresses.

If a label appears in any Autocoder statement, it may be written as an operand in any other Autocoder statement. During processing, the assembler substitutes the equivalent address of the label whenever the label appears as an operand in a source-program statement. Thus, the programmer refers symbolically to the equivalent address of the constant, work area, or instruction.

Operation Codes

Most Autocoder statements have operation codes. (See *Subsequent DA Entries* for an exception.) In imperative instruction statements they are machine-operation codes such as A (ADD), S (SUBTRACT), SD (SEEK DISK), and P (PUNCH).

In area-definition statements they are commands to the assembler to allocate storage, such as DCW (Define a Constant with a Word Mark and DA (Define Area).

In assembler-control statements, they are signals to the assembler such as ORG (begin or originate the program) and END (end the program).

The appendix of this publication contains charts that list all valid mnemonic operation codes.

Operands

Use the operand portion of an Autocoder statement to specify:

1. *For instruction statements:* the address of the data to be operated on or the input/output units to be operated, and the d-character modifier to the operation code, if required.
   (A list of all valid operand sequences is included in the Appendix.)
2. *For area-definition statements:* the constant or area to be defined, or the address or input/output unit that is to be the equivalent of the label.
3. *For assembler-control operations:* the address to be used in a particular assembler operation.

Core-Storage Address Operands

There are five types of address operands used in Autocoder statements: symbolic, actual, asterisk, blank, and literal.

Symbolic

A symbolic operand refers to the equivalent address of an instruction or defined area. The symbolic operand must be the same as the label of the instruction or area-definition statement. Writing a symbolic operand in a statement that precedes the labeled statement is permitted.

In Figure 2, ENTRYA is used as a label for an ADD instruction and as a symbolic operand in a branch instruction. Assume that the equivalent addresses of ENTRYA, WHTAX, and DEDUCT are 568, 701, and 905 respectively. The assembled machine-language instructions would be A 701 905 and B 568. In a program using these instructions, WHTAX and DEDUCT would be used as labels elsewhere in the program.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
ENTRYA	A	WHTAX,DEDUCT
	.	
	.	
	.	
	B	ENTRYA

Figure 2. Symbolic Operand

Actual

The programmer may use an actual address as an operand in any Autocoder statement. This address is a one-to-five digit number within the range 0 to 15999, and represents an actual core storage position.

For example, to cause a word mark to be set in location 001 during execution of the object program, write in the source-program the instruction shown in Figure 3. Note that it is not necessary to write high-order zeros in an actual address written in Autocoder.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	SW	1

Figure 3. Actual Address Operand

Asterisk

Writing an asterisk in an Autocoder statement directs the assembler to assign an address equivalent to the right-most (low-order) position of the area that the instruction or data will occupy in the object machine.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
COMPR	C	A,B
	BU	TOTAL
	.	
	.	
	.	
TOTAL	A	*-6,COUNT

Figure 4. Asterisk Operand

Figure 4 shows a routine designed to compare field-A to field-B, and to add 1 to a field named COUNT if the result is unequal. Assume that the equivalent addresses of TOTAL and COUNT are 459 and 711 respectively. The asterisk then refers to 465, which is the address of the low-order position of the seven-character assembled instruction and *–6 refers to 459. The assembled instruction is A 459 711. When the instruction is executed, one is added to COUNT because 459 is the address of the operation code (A). In core storage, an A is composed of A- and B-zone bits and a 1-bit; these zone bits form the standard plus sign, and do not change the addition of the numeric 1. Figure 5 is a representation of the instruction in core storage during program execution.

Character	A	4	5	9	7	1	1
Core Storage Location	459	460	461	462	463	464	465

Figure 5. Instruction in Core Storage

Blank

Blanks are valid in statements where no operand is needed, or when useful addresses are supplied by the chaining method. Chaining is explained in the *System Operation Reference Manual.*

Literals

A literal operand is the actual data to be used when the instruction in which it appears is executed. The assembler stores the actual data (constant) with a word mark over the high-order position when it encounters a LTORG, EX, or END assembler-control statement. The equivalent address of the stored constant is substituted for the literal operand when the instruction is assembled. The programmer can address-adjust and/or index a literal. See *Operands: Address-Adjustment and Indexing.*

Duplicate literals are assigned core-storage space only once per program or program section. When a literal is referred to, a program section means those source-program entries that precede a LTORG, EX, or END assembler-control statement.

Figure 6 shows literal operands and the constants produced for them.

Type of Literal	Literal Operand	Stored Constant
Numeric	+10	1?
Alphameric	@JANUARY 28, 1962@	JANUARY 28, 1962
Area-Defining	WORKAR#6	bbbbbb
Address Constant	+CASH	xxx (Equivalent Address of CASH)

Figure 6. Literals

Numeric Literals. A numeric literal must be made up of integers only (0-9) and must be preceded by a plus or minus sign. The sign is necessary because the assember uses it to distinguish numeric literals from actual addresses. The literal may be any length, provided that it is contained in the operand portion of one program card. The sign is stored in the same core-storage position as the units position of the numeric literal.

Figure 7 shows how a numeric literal can be coded in an Autocoder imperative instruction. Assume that the literal (+10) is assigned storage locations 584 and 585, and INDEX is assigned an equivalent address of 682. The symbolic instruction causes the assembler to produce a machine-language instruction (A 585 682) that adds +10 to the contents of INDEX when the instruction is executed in the object program.

Label	Operation	OPERAND
	A	+10,INDEX

Figure 7. Numeric Literal

Alphameric Literals. An alphameric literal is one or more alphameric characters written between two @ symbols. Alphameric characters include numeric, alphabetic, and special characters (including blanks). Any combination of alphameric characters can be used within the two @symbols, with the following restrictions:

1. If the object program is to be punched into cards in the *condensed-loader* format, a word-separator character (0-5-8 punch) should not be the first character following the first @ symbol.
2. If the object program is to be written on disk (*coreload* format), a group mark should not be the first character following the first @ symbol.

(Object-program formats are described in the publication: *Autocoder (on Disk) Program Specifiications and Operating Procedures, IBM 1401, 1440,* and *1460,* form C24-3259.)

Only one alphameric literal is permitted in a coding-sheet line. One or more @ symbols can be included within an alphameric literal (between the two @ symbols enclosing the literal), but an @ symbol must not appear anywhere else in a line containing an alphameric literal. The assembler scans the contents of the card from the left for the first @ symbol and from the right for the second @ symbol. All characters between the two @ symbols are assumed by the assembler to be part of the literal.

Figure 8 shows how to use an alphameric literal in an imperative instruction. Assume that during assembly the literal JANUARY 28, 1964 is assigned a storage area whose equivalent address is 906, and DATE is assigned 230. For this statement, the assembler produces a machine-language instruction (M 906 230) that moves the literal JANUARY 28, 1964 to DATE.

Label	Operation	OPERAND
	MLC	@JANUARY 28, 1964@,DATE

Figure 8. Alphameric Literal

Address-Constant Literals. An address-constant literal is the label of an instruction, defined area, or constant preceded by a plus or a minus sign. A plus sign preceding the label indicates that the constant represented by the literal is the machine-address the assembler assigns to the label. A minus sign preceding the label indicates that the constant represented by the literal is the 16,000's complement of the machine-address assigned to the label. The address assigned to any label, *except* labels associated with *area-defining literals,* can be represented as constants by address-constant literals. (Area-defining literals are described in the following section.)

When the assembler encounters an address-constant literal, it:

1. Allocates a three-position area that will contain the equivalent address or its 16,000's complement at object-program execution time. (The equivalent address is a three-character machine-language address.)
2. Assigns an address to the allocated area and equates the address-constant literal to the assigned address.

Figure 9 shows two address-constant literals (+ CASH and + CHECKS) used in a source program. It also shows the entries the assembler makes in the object program, and the results when the instructions are executed by the object program. The pro-

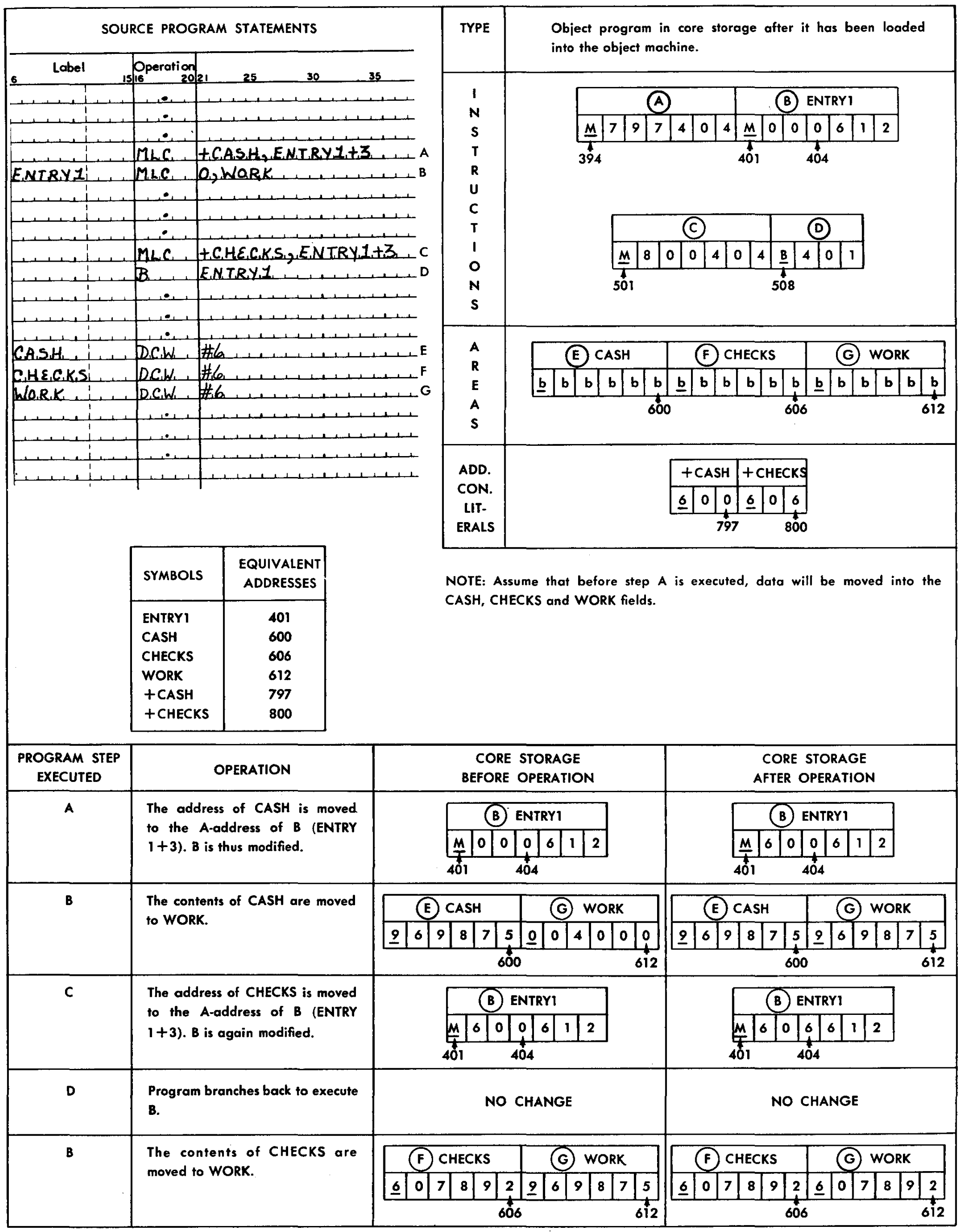


Figure 9. Address-Constant Literals

grammer did not know what addresses would be assigned to CASH and CHECKS when he wrote the source-program statements. He did, however, write two instructions (A and C) that move these addresses into instruction B (ENTRY 1). The address-constant literals (+ CASH and + CHECKS) caused the assembler to allocate storage in the object machine for equivalent addresses of CASH and CHECKS and to substitute the addresses of the address-constant literals in instructions A and C.

Autocoder permits the programmer to adjust an equivalent address. To use the adjusted equivalent address, code the address-constant literal as follows:

1. Plus or minus sign.
2. Period.
3. Label whose equivalent address is to be adjusted.
4. Adjustment factor (plus or minus any integer that will produce a number greater than zero, but less than the number of core-storage positions available in the object machine) and/or an index-register symbol.
5. Period.

Figure 10 shows an equivalent address that is modified by an adjustment factor. Assume that the equivalent address of TOTAL is 565. When the instruction is executed 561 will be moved to the area whose label is SUM.

Label	Operation	OPERAND
	MLC	+.TOTAL-4.,SUM

Figure 10. Adjusted Address-Constant Literal

Figure 11 shows an equivalent address that is modified by an adjustment factor and the contents of an index register. Assume that the equivalent address of TOTAL is 565. The constant that will be adjusted is 565. The adjustment factor is —4. The 16,000's complement of 561 is used because the address-constant literal contains a minus sign.

Label	Operation	OPERAND
	MLC	-.TOTAL-4+X3.,SUM

Figure 11. Adjusted and Indexed Address-Constant Literal

When the instruction is executed, DCI will be moved to the area whose label is SUM. D3I is the machine language equivalent of 15,439 [16,000 — (565 — 4)]. The 3 becomes a C because A- and B-bits represent X3. See *Indexing* for a discussion of index registers.

Area-Defining Literals. This literal is used to define an area of blanks equal to the number following the # symbol. The area may be referred to by using the label that precedes the # symbol.

At object-program load time, the defined number of blanks will be loaded into storage with a word mark in the high-order position.

For example, in the statement shown in Figure 12 the area-defining literal is #5, which can be referred to as WORKAR. Assume that the equivalent address of OUTAR is 800. If the assembler assigns locations 896-900 to the label WORKAR, then the assembled instruction will be: M 900 800. This instruction will move the contents of WORKAR to locations 796-800 when it is executed in the object program.

Label	Operation	OPERAND
	MLC	WORKAR#5,OUTAR

Figure 12. Area-Defining Literal

Note: If a source program consists of two or more sections, the label that precedes the # symbol can be used only in the program section that contains the area-defining literal.

Address Adjustment

It is not necessary to devise so many labels for a source program, if adjustment for addresses is specified in the operand fields of Autocoder statements. To do this, write an integer preceded by a plus or minus sign immediately following the operand. The assembler then develops an equivalent address, plus or minus the adjustment factor, and inserts it into the assembled object-program statement in place of the address-adjusted operand. The adjustment factor can be any positive or negative integer that will produce an address greater than zero but less than the number of core-storage positions available in the object machine.

Figure 13 shows a symbolic operand with address adjustment. Assume that the statement whose label is LAST is assigned storage locations 404 through 407. The equivalent address of the label LAST is then 404, which is the position that the B operation code of the branch instruction will occupy in core storage when the object program is loaded.

Label	Operation	OPERAND
	SBR	LAST+3
	.	
	.	
	.	
LAST	B	0

Figure 13. Address Adjustment

The assembler substitutes the address of LAST + 3 (407) in place of the symbolic address-adjusted operand (LAST + 3) when the object program is assembled: H 407 . . . . B 000.

When the object program is executed, the contents of the B-address register are transferred to positions 405-407, so that the I-address of the branch instruction contains whatever was in the B-address register before the SBR instruction was encountered (B xxx).

Figure 14 shows an address-adjusted literal operand. The first statement is an instruction that adds a literal (+100) to SUM. The assembler allocates a three-position area in core storage to store this literal. Assume that the equivalent address of this literal is 698, and SUM has an equivalent address of 805. The assembled instruction is A 698 805. Later in the source program the same literal appears with address-adjustment. Because the literal has been previously assigned with an area whose address is 698, the address-adjusted literal +100—2 refers to 698—2 or 696. Thus, the assembled instruction, A 696 805, will add 1 into SUM when it is executed in the object program, because storage-location 696 contains the 1-portion of the literal +100.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	A	+100,SUM
	.	
	.	
	.	
	A	+100-2,SUM

Figure 14. Address-Adjusted Literal

Figure 15 shows an address-constant literal operand with address adjustment. Assume that the equivalent addresses of the literal (+ACCUM) and TOTAL are 697 and 734, respectively, and that the address-constant literal is 419 (equivalent address of ACCUM). The assembled instruction is A 697 734. Later in the source program the same address-constant literal appears with address-adjustment. Because the literal has been previously assigned to an area whose address is 697, the address-adjusted literal +ACCUM—1 refers to 697 —1 or 696. Thus, the assembled instruction, A 696 734, will add 41 into TOTAL when it is executed in the object program, because 696 is the address of the area that contains 41. The instruction does not affect the address-constant literal (419).

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	A	+ACCUM,TOTAL
	.	
	.	
	.	
	.	
	A	+ACCUM-1,TOTAL

Figure 15. Address-Adjusted Literal

Indexing

If an object machine has the advanced-programming special feature (1401) or the indexing-and-store-address-register feature (1440 and 1460), the source programmer can use the three 3-position index locations (registers) provided by the feature. The assigned core-storage addresses and index-register numbers are shown in Figure 16.

Index Location	Core-Storage Locations	3-character Machine Address	Tag bits in tens position of 3-character machine address
1	087-089	089	A-bit, No B-bit
2	092-094	094	B-Bit, No A-bit
3	097-099	099	A-bit, B-bit

Figure 16. Index Locations and Associated Tag Bits

The primary use of index locations is to modify addresses automatically by adding the contents of an index location to an address. The core-storage address of the A- and/or B-operand can be modified by the contents of any index location:

1. Set a word mark in the high-order position of the index-register location before inserting the index factor.
2. Use an add or move instruction to insert or change the index factor. The programmer can use a label, X1, X2, X3, or the actual machine address (89, 94, or 99) as the B-operand. If he uses a label he must first write an EQU statement to assign a label to the index location. (See *EQU—Equate.*)

   Note: If an index factor is to be used for address modification the user should be sure that no zone bits appear in the tens position of the factor, nor in the units position if the system has 4000 or fewer positions of core storage.

3. Write +X1, +X2, or +X3 after the operand that is to be indexed. X1, X2, and X3 represent index registers 1, 2, and 3, respectively.

When the assembler encounters an indexed operand, it puts tag bits over the tens position of the 3-character machine address assigned to the operand to specify which index register is to be used. The bit combinations and the registers they specify are shown in Figure 16.

The modfication of the A- and/or B-address occurs in their respective address registers. For instance, if the A-address is indexed, the indexing occurs in the A-address register. This means that the original instruction in storage is in no way changed or modified.

The three index registers can be used as normal storage positions when not being used as index-register locations.

Figure 17 shows an indexed imperative instruction that causes the contents of the location labeled TOTAL to be placed in an area labeled ACCUM as modified by the contents of index-location 2. TOTAL is the label for locations 3101 and ACCUM is the label for location 140. The assembled machine-language instruction for this entry is: M A01 1MO. The M in the tens position of the B-address is a 4-bit and a B-bit. The B-bit is the tag for index-locaton 2.

Label	Operation	OPERAND
	MLC	TOTAL,ACCUM+X2

Figure 17. Symbolic Operand with Indexing

Symbolic Indexing

Symbolic indexing is permitted in any statement that can have actual indexing, except in an EQU statement or in a DA statement. The name used can be as many as six letters or digits, but the first character must be a letter.

The assembler first reserves the index location(s) referred to by actual addresses (X1, X2, and/or X3) in the source program. Later, unreserved index locations are assigned to the symbolic references in the order of occurrence in the source program. For example, if the statement shown in Figure 18 appears in a source program, INDEXA will be assigned to an unused index location.

Label	Operation	OPERAND
ENTRYC	A	FIELDA+INDEXA,FIELDB

Figure 18. Symbolic Indexing

After all three index locations have been reserved, the assembler will not process any *new* symbolic reference. Instead, an error indication will print on the assembly listing. Because the assembler must control the assignment of index locations, a symbolic reference to an index location cannot be equated by the use of an EQU statement to an actual address of an index location.

To insert or change the index factor, write an add or move instruction with the name of the index location as the B-operand. The name must not be used as a label elsewhere in the program.

Address-Adjustment and Indexing

Figure 19 shows an imperative instruction with address adjustment and indexing on a symbolic address. The assembler will subtract 12 from the address that was assigned the label TOTAL. The effective address of the A-operand is the sum of TOTAL −12 plus the contents of index-location 1 at program-execution time. The assembled instruction (M ?Y9 140) will cause the contents of the effective address of TOTAL −12 +X1 to be placed in the location labeled ACCUM (assuming again that TOTAL is the label for location 3101 and ACCUM is the label for location 140). The Y in the tens position of the A-address is an 8-bit and an A-bit. The A-bit is a tag for index location 1.

Label	Operation	OPERAND
	MLC	TOTAL-12+X1,ACCUM

Figure 19. Symbolic Operand with Address-Adjustment and Indexing

Figure 20 shows examples of address-constant-literal adjustment and of address-constant-literal address adjustment. Assume that the equivalent addresses of the address-constant literal (+TAX or −TAX) and ADDR are 503 and 700, respectively, and that the address constant of TAX is 123. (See *Address-Constant Literals* and *Address Constants Defined by a DCW Statement.)*

Constant Operands

Constant operands are defined by area-definition statements. See *DC and DCW Statements.* The assembler assigns an area in core storage in which the constant is stored at object-program load time.

Input/Output Operands

For operations involving disk storage, write the mnemonic operation code in the operation field and the symbolic disk-address control field in the operand field. For example, the statement shown in Figure 21 will be assembled M %F1 598 W if 598 is the equivalent address of OUTPUT.

Label	Operation	OPERAND
	WD	OUTPUT

Figure 21. Write Disk

For operations involving magnetic tape, write the mnemonic operation code in the operations field and the number of the tape unit in the operand field. The programmer can specify the number of the tape unit in one of three ways:

1. Write the actual address of the tape unit (%Ux) as

Type of Adjustment	Source Program Instruction	Assembled Instruction	Constant Moved to ADDR (700)	Constant Stored in 501 - 503
Adjusting the address constant literal	Operation: MLC; Operand: +.TAX-1..,ADDR	M 503 700	122	122
Adjusting and indexing the address constant literal	Operation: MLC; Operand: +.TAX-1+X2..,ADDR	M 503 700	1K2	1K2
Adjusting the 16,000's complement of the address constant literal	Operation: MLC; Operand: -.TAX-1..,ADDR	M 503 700	H7H (15,878)	H7H
Adjusting the address of the address constant literal	Operation: MLC; Operand: +TAX-2,ADDR	M 501 700	1	123
Adjusting the address constant literal and the address of the address constant literal	Operation: MLC; Operand: +.TAX-1.-1,ADDR	M 502 700	12	122
Indexing the address constant literal and adjusting the address of the address constant literal	Operation: MLC; Operand: +.TAX+X2.-1,ADDR	M 502 700	1K	1K3

Figure 20. Address-Constant Literals with Adjustment and Indexing

the A-operand. The statement shown in Figure 22 will be assembled M %U4 615 W if 615 is the equivalent address of OUTPUT.

Label	Operation	OPERAND
	WT	%U4,OUTPUT

Figure 22. Tape Instruction with Actual Address

2. Assign a label to the actual address of the tape unit, and use it as the A-operand of the tape instruction. (See *EQU—Equate.)*
3. Write the number of the tape unit in column 21 of the tape instruction. The assembled instruction for the statement shown in Figure 23 will cause a record to be written on tape-unit 4 using the data beginning in a storage area labeled OUTPUT.

Label	Operation	OPERAND
	WT	4,OUTPUT

Figure 23. Write Tape

For operations involving the 1443 printer, write W or WS in the operation field and the symbolic address of the high-order position of the print-line in the operand field. For example, the statement shown in Figure 24 will be assembled: M %Y1 801 W if 801 is the equivalent address of PRINT1.

Label	Operation	OPERAND
	W	PRINT1

Figure 24. Printer Operand

For operands involving the 1442 card read-punch, write the mnemonic operation code in the operation field. Then write the number of the unit (1 or 2), followed by a comma and the symbolic address of the high-order position of the I/O area. For example, M %G1 110 R will be the instruction assembled from the statement shown in Figure 25, if 110 is the equivalent address of INPUT.

Label	Operation	OPERAND
	R	1,INPUT

Figure 25. Reader Operand

Refer to the Appendix for a list of the mnemonics and operands that can be used to specify input/output operations.

Statement Descriptions

All Autocoder statements must be presented to the assembler program according to a special format. There are also rules and restrictions for writing the information in these statements. These requirements are necessary because the assembler needs and can handle only certain kinds of information from each type of Autocoder statement, and it must know where in the statement that information can be found.

In this publication the Autocoder statement descriptions are presented in a format that:

1. Describes the operation which the statement specifies.
2. Shows how the statement is written by the programmer.
3. States the actions of the assembler during processing of the statement.
4. Describes the effect of the statement on the object program.
5. Shows an example of the statement.

Declarative Statements

Declarative statements are used to assign sections of storage for fixed data (constants) that will be needed during processing, to set aside work areas, and to assign symbolic names to data and devices used in the program.

The six declarative operations are:

Op Code	*Purpose*
DCW	Define Constant with Word Mark
DC	Define Constant (No Word Mark)
DS	Define Symbol
DSA	Define Symbol Address
DA	Define Area
EQU	Equate

DCW — Define Constant with Word Mark

General Description. Use a DCW statement to enter a numeric, alphameric, blank, or address constant into core storage at object-program load time.

The programmer:

1. Writes DCW in the operation field. If more than one DCW statement is to be written in succession, the programmer needs to write the DCW operation code for the first DCW statement. The DCW operation code for the remaining statements of the group can be omitted, if desired.
2. May write a label, but not an actual address, in the label field. He can refer to the constant by using the label as an operand elsewhere in the program. If the label starts in column 6, its equivalent address is the address of the low-order position of the constant in the object machine. If the label starts in column 7, its equivalent address is equal to the high-order position of the constant in the object machine.
3. Writes the constant in the operand field beginning in column 21. A comma and a G immediately following the constant inserts a group-mark with a word-mark after the constant.

The assembler:

1. Allocates a field in core storage that will be used at object-program load time to store the actual constant.
2. Inserts the equivalent address of the label in the object program wherever the label is used as a symbolic operand in a source-program statement.

Result: The constant with a high-order word mark is loaded with the object program.

Numeric Constants

A plus or minus sign may be written preceding an integer. A plus sign causes the assembler to store the constant with A- and B-bits over the units position; a minus sign stores a B-bit there. If the integer is unsigned, it will be stored as an unsigned field.

The first non-numeric column in the operand field indicates that the preceding position contains the last digit in the constant.

A constant may be as large as 51 digits with a sign, or 52 digits with no sign.

Examples. Figures 26, 27, and 28 show the three types of numeric constants that can be defined in DCW statements. The labels TEN1, TEN2, and TEN3 identify the constants. Thus, they can be used as operands to cause the equivalent addresses of +10, −10, and 10 to be inserted in the object program whenever TEN1, TEN2, and TEN3 appear in operand fields of other entries in the source program.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TEN1	DCW	+10

Figure 26. Numeric Constant with a Plus Value

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TEN2	DCW	-10

Figure 27. Numeric Constant with a Minus Value

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TEN3	DCW	10

Figure 28. Unsigned Numeric Constant

Form C24-3258-1
Page Revised 6-1-65
By TNL N21-0038-0

Alphameric Constants

Place an @ symbol before and after the constant. As with alphameric literals, blanks and the @ symbol may appear between these @ symbols, but the @ symbol must not appear in a comment in the same line as the constant.

Up to 50 valid characters can be written in an alphameric constant. Any combination of alphameric characters can be used, with the following restrictions:

1. If the object program is to be punched into cards in the condensed-loader format, a word-separator character (0-5-8 punch) should not be the first character following the first of the two @ symbols enclosing the constant.
2. If the object program is to be written on disk (coreload format), a group mark should not be the first character following the first of the two @ symbols enclosing the constant.

A comma and a G following the alphameric constant cause the assembler to insert a group-mark with a word-mark after the constant.

Example: Figure 29 shows how to define the alphameric constant, JANUARY 28, 1964 in a DCW statement. The assembler will insert the equivalent address of the constant in the object-program instruction wherever DATE appears in the operand of another source-program entry.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
DATE	DCW	@JANUARY 28, 1964@

Figure 29. Alphameric Constant

Blank Constants

Blank constants used in DCW statements are equivalent to area-defining literals in instructions.

Write the # symbol and an integer in the operand field to indicate how many blank storage positions are needed in the area. The defined area can contain any number of blank positions.

Example: Figure 30 shows how to define an 11-position blank field using a DCW statement. The equivalent address of the 11-position field is inserted in the object program wherever BLANK appears as an operand in another source program statement.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
BLANK	DCW	#11

Figure 30. Blank Constant

Address Constants

A DCW statement can define the equivalent address of an instruction, defined area, or constant.

In the operand field, write the label of the instruction, area-definition, or constant, and precede the label with a plus or minus sign. If a minus sign is used, the constant defined is the 16,000's complement of the equivalent address of the label.

Example. Figure 31 shows how an address constant (the equivalent address of MANNO) can be defined by a DCW statement. The address of the equivalent address of MANNO will be inserted into an object-program instruction wherever SERIAL appears as the operand of another source-program entry. Thus, +MANNO is the symbolic address of the field that contains the equivalent address of MANNO.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
SERIAL	DCW	+MANNO

Figure 31. Address Constant Defined by a DCW Statement

Address constants can be adjusted and indexed. The adjustment and indexing refer to the address constant itself rather than to the address of the location of the address constant. If CASH is the symbolic address of a field, the equivalent address of CASH is indexed or address-adjusted rather than the equivalent address of +CASH.

Example. In Figure 32 the address constant (the equivalent address of CASH) is 600. Whenever TOTAL appears as the operand of another source-program entry, it will represent the equivalent address of a location that contains 604 (the adjusted address constant of CASH). (See Figure 20.)

Note: −CASH+4 would refer to position 15,404 (16,000−600+4).

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TOTAL	DCW	+CASH+4

Figure 32. Adjusted Address-Constant Defined by a DCW Statement

DC — Define Constant (No Word Mark)

General Description: To load a constant without a high-order word mark, write a DC statement. The format of a DC statement is the same as that of a DCW statement. The DC operation code is used in the operation field. If more than one DC operation code is to be written in succession, the programmer needs to write the DC operation code for the first DC statement. The DC operation code for the remaining statements of the group can be omitted, if desired.

Example: Figure 33 shows TEN1 defined as a constant without a word mark.

Label	Operation	OPERAND
TEN1	DC	+10

Figure 33. Constant Defined by a DC Statement

DS — Define Symbol

General Description. Use a DS statement to label and skip over an area of core storage. With a DS statement, the bypassed area is undisturbed during the loading process. Thus, any information that was in storage before loading begins will still be there after the object program has been loaded.

The programmer:

1. Writes DS in the operation field.
2. May write a label, but not an actual address, in the the label field.
3. Writes a number in the operand field that tells the assembler how many positions of storage to bypass.

The assembler:

1. Assigns an equivalent address to the label. This equivalent address refers to the low-order position of the bypassed area.
2. Inserts this address wherever the label appears as an operand in another source-program entry.

Result. The positions included in the bypassed area remain undisturbed during object-program loading.

Example. Figure 34 shows how to direct the assembler to bypass a 10-position core-storage area. Assume that the last core-storage position the assembler allocated before it encountered the DS statement was 940. The equivalent address of ACCUM is 950, the address of the low-order position of the core-storage area bypassed by the DS statement. Wherever ACCUM is used as an operand, 950 will be inserted in the object program.

Label	Operation	OPERAND
ACCUM	DS	10

Figure 34. DS Statement

DSA — Define Symbol Address

General Description. The ability to code address constants in Autocoder language eliminates the need for the DSA statement except when the three-character machine address of an actual address in the source program is desired. The address constants previously discussed were created from labels.

The programmer:

1. Writes DSA in the operation field.
2. May write as the label, the name that will be used to make reference to the address constant.
3. Writes the actual or symbolic address to be defined in the operand field. This address may be address-adjusted and indexed. DSA with a symbolic operand is equivalent to a DCW address constant.

The assembler:

1. Produces a constant containing the equivalent address of the storage address written in the operand field.
2. Assigns to this address constant an equivalent address in core storage and labels it using the name that appears in the label field.

Result. At program-load time the address constant will be loaded into its assigned locations with a word mark in the high-order position.

Example. Figure 35 shows how to develop and store an address constant for an actual address.

Label	Operation	OPERAND
MINSIX	DSA	694

Figure 35. Defining the Address Constant of an Actual Address

DA — Define Area

General Description. A DA entry reserves and defines portions of core storage. Use a DA entry to:

1. Define one area, such as an input, output, or work area.
2. Define several areas that have the same format.
3. Define fields within the defined area.

The complete DA entry has two parts: the DA header, which gives the assembler specific information on how to set up the area, and the subsequent DA statements, which define the fields within the area.

DA Header

The programmer:

1. Writes DA in the operation field.
2. Writes a label. The equivalent address of the label represents the high-order position of the entire area defined by the DA header statement.
3. Writes the first operand in the form B × L. B is the number of identical areas to be defined and L is the length of each area.
4. May write a comma and the number of an index location (X1, X2, or X3) after the B × L entry. The indexing specified in the DA header statement refers

to subsequent DA entries. Tag bits will be over the tens position of the equivalent addresses wherever the labels of subsequent DA entries appear as operands in source-program instructions, *unless the operand is indexed.* The indexing in the operand overrides the indexing specified by the DA header.

Note: Symbolic indexing is not permitted in a DA header statement.

5. May write ,+ after the B × L entry to cause the assembler to insert a record mark without a word mark immediately after each area defined by the B × L entry.
6. May write ,G after the B × L entry to cause the assembler to insert a group-mark with a word-mark immediately after the last area defined by the B × L entry.
7. May write ,C to cause the assembler to clear the defined area(s) at object time before any word marks are set.

Note: The ,+,G,C and, index-code entries may be written in any order after the B × L entry.

Subsequent DA Entries

The programmer:

1. Leaves the operation field blank.
2. May write a label. The equivalent address of the label represents the low-order position of the field or subfield with which it is associated. A subfield is a field within a defined area or field.
3. Specifies in the operand field the relative location of a field or subfield. The first position of each area defined by the DA header statement is considered location 1.
   a. To define a field, write the high-order and low-order position of the field (beginning in column 21). Separate the two numbers by a comma. To define a one-position field, write the relative location number twice. Word marks are set in the high-order positions of all defined fields.
   b. To specify the location of a subfield, write the number (beginning in column 21) that represents the relative location of the low-order position of the subfield. The location is relative to the first position of the area defined by the DA header statement. No word marks are set in the low-order positions of subfields.

      A subfield can be located anywhere within the area defined in the DA header statement. It does not have to be within a field defined by a subsequent DA entry.
4. May list fields and subfields in any order after the DA header statement. All the fields within the area need not be defined.

The assembler:

1. Allocates an area which is equal in length to the total of B × L plus positions for record marks and a group-mark with a word-mark, if they have been specified in the DA header.
2. Assigns equivalent addresses to the DA header label and to the labels of all defined fields and subfields.
3. Inserts the equivalent address of the high-order position of the entire defined area wherever the label of the DA header appears as an operand in the program.
4. Inserts the equivalent addresses of the low-order positions of fields and subfields defined in the other DA entries wherever their labels appear as operands in the program.

Result. When the object program is loaded:

1. The entire defined area is cleared if the DA header statement contained a comma C.
2. Word marks are set in the high-order position of all fields defined by subsequent DA entries. A word mark is set in the high-order position of each area defined by the DA header if a subsequent DA entry is 1,n.
3. A group-mark with word-mark, and record marks are set if they have been specified by the DA header.

Example. Figure 36 shows a DA header statement that defines four 100-position areas. If only one area is to be defined, write 1 × 100 as the first entry in the operand field.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TAPEAR	DA	4X100

Figure 36. DA Header

Example. INAREA is defined by the DA header shown in Figure 37. The second statement in Figure 37 defines a field within INAREA. Thus, the equivalent address of ACCUM has a tag bit (A-bit) over the tens position to indicate that it is to be indexed by the contents of index-location 1.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
INAREA	DA	3X60,X1
ACCUM		35,40

Figure 37. Indexing a DA Entry

However, an imperative statement elsewhere in the source program indicates that ACCUM is to be modified by the contents of index-location 2. Because the statement shown in Figure 38 contains indexing, the assembler will tag the equivalent address of ACCUM with a B-bit when it assembles the instruction for that statement. The indexing in the statement that uses ACCUM as an operand overrides the indexing prescribed by the DA header statement.

Label	Operation	OPERAND
	ZA	GROSS,ACCUM+X2

Figure 38. Overriding Previously Prescribed Indexing

To negate the effect of indexing on a field or subfield, put an X0 in the operand field of each instruction in which indexing is not wanted (Figure 39).

Label	Operation	OPERAND
	ZA	GROSS,ACCUM+X0

Figure 39. Negating the Effect of Indexing

Example. Figure 40 shows a DA header statement that directs the assembler to insert a record mark after each of the areas defined and a group-mark with word-mark immediately after the last-defined area. The 2 ×100 entry causes 200 positions to be reserved by the assembler. The ,‡ and ,G entries cause 3 additional positions to be reserved as shown in Figure 41.

Label	Operation	OPERAND
OUTA	DA	2X100,‡,G

Figure 40. DA Header

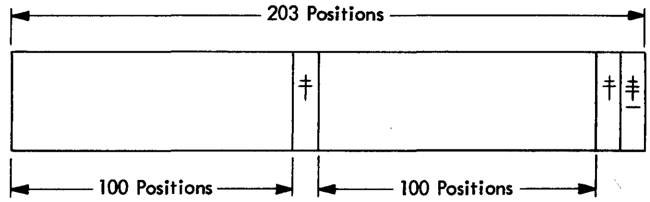


Figure 41. Record Marks and a Group-Mark with a Word-Mark

Example. A payroll record (Figure 42) is to be moved into an area of core storage.

The card record is 80 positions, one for each column of the card. The significant fields to be defined in the record area are:

Positions	*Label*	*Description*
4-8	MANNO	Man Number
11-26	NAME	Employee Name
32-37	DATE	Date
45-64	GROSS	Gross Wages
66-71	WHTAX	Withholding Tax
74-79	FICA	FICA Deduction

The remaining card columns contain data not needed for the operation. Positions 34 and 35, which indicate the month within the date, will be defined as a subfield. A group-mark with a word-mark is needed in the storage position immediately following the area.

The DA header shown in Figure 43 defines the area into which the record is to be moved. The 1 × 80 entry causes the assembler to reserve 80 core-storage

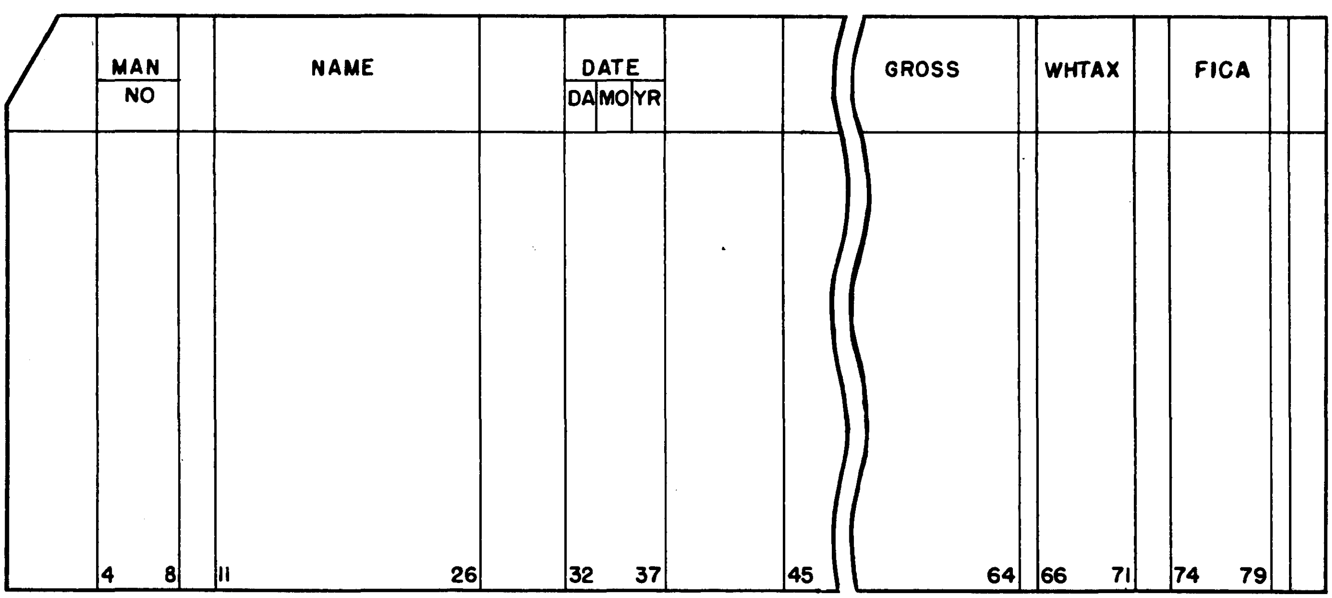


Figure 42. Input Card

positions for the area. The ,G causes the assembler to set a group-mark with a word-mark in the 81st position of the area at program-load time. This area can then be referred to by using the name RDAREA in the operand of another source-program entry. The equivalent address of RDAREA will be the 3-character machine address of the high-order position of the entire area allocated by the assembler. The other DA entries shown in Figure 43 define the fields and subfields within the record.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
RDAREA	DA	1X80,G
DATE		32,37
MONTH		35
NAME		11,26
MANNO		4,8
GROSS		45,64
WHTAX		66,71
FICA		74,79

Figure 43. DA Entry

In the source program, an instruction to move the record into a storage area labeled RDAREA will cause the data in the record to be stored in the appropriate fields. Source-program statements may then be written to manipulate this data, using the labels as operands. The word marks set at program-load time will stop the transfer of data when individual fields are moved, added, etc.

EQU — Equate

General Description: Use an EQU statement to assign a label to an actual, asterisk, or symbolic address, or to a control field or an index location. More than one label can be assigned to represent the same storage location.

The programmer:

1. Writes EQU in the operation field.
2. Writes a label.
3. Writes an actual, asterisk, or symbolic address in the operand field.

Note: X1, X2, and X3 should not be used as labels of EQU statements, because the assembler assumes that they are equated to 089, 094, and 099, respectively. Further, a label must not be equated to a literal, because the assembler considers such a label as being undefined.

The assembler:

1. Assigns to the label of the EQU statement the same equivalent address that was assigned to the name in the operand field (with appropriate alteration if indexing and address adjustments are indicated).
2. Inserts this equivalent address wherever the label of the EQU statement appears as an operand.

Result: Either the label or the operand of the EQU statement can be used to refer to the same core-storage location.

Examples: Figure 44 shows how to assign another label (INDIV) to a location which was previously labeled MANNO. The EQU statement causes the assembler to assign the same equivalent address (1976) to INDIV that it previously assigned to MANNO. Now, whenever either MANNO or INDIV appears as an operand, the assembler will replace the operand with 1976.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
INDIV	EQU	MANNO

Figure 44. Equating Two Symbolic Addresses

Figure 45 shows a statement equating the equivalent address of FICA—10 to WHTAX. If the assembler assigns FICA an equivalent address of 890, the WHTAX will be assigned an equivalent address of 880, which is also equal to FICA—10. WHTAX now refers to a field whose units position is 880.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
WHTAX	EQU	FICA-10

Figure 45. Equating a Symbolic Address to an Address-Adjusted Symbolic Address

Figure 46 shows how to equate a label to an actual address. Assume that a certain field will be in a storage location whose units position is known to be at actual-address 319. The programmer wishes to refer to this field as ADDA, but it has not been labeled elsewhere in the program. To equate the label ADDA to 319, write the statement shown in Figure 46. Thus, 319 becomes the equivalent address of ADDA.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
ADDA	EQU	319

Figure 46. Equating a Label to an Actual Address

Figure 47 shows how to index an operand in an EQU statement. With indexing, the label of the EQU statement is indexed by the same index location that is specified in the operand field of that EQU statement. However, if the label appears in the operand field of another source-program entry with another index code, the new code overrides the index code in the EQU statement.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
CUSTNO	EQU	JOB+X3

Figure 47. Indexing an EQU Statement

For example, in the statement shown in Figure 47 the equivalent address of JOB wth the tag bits of index-location 3 is assigned to the label CUSTNO. Thus, if JOB+X3 is equal to 5H5, CUSTNO also has 5H5 as its equivalent address. However, if CUSTNO+X1, CUSTNO+X2, or CUSTNO+X0 appears as the operand of another source-program entry, the address inserted in its place will be 5Y5, 5Q5, or 585, which specifies index-location 1, 2, or none, respectively.

Figure 48 shows how to assign a label to an asterisk in an EQU statement. The * refers to the last storage location the assembler assigned before it encountered the EQU statement. Assume that this address is 698. FIELDA has an equivalent address of 698.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
FIELDA	EQU	*

Figure 48. EQU Statement with an Asterisk Operand

Figure 49 shows how to assign a label to an index location. Because the actual core-storage address of index-location 1 is 089, the EQU statement assigns the label INDEX1 to that location.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
INDEX1	EQU	89
	(OR)	
INDEX1	EQU	X1

Figure 49. Equating a Label to an Index Location

Figure 50 shows how to assign a label to the card-reader number 1 whose actual address is %G1. It is now possible to refer to this device as INPUT1.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
INPUT1	EQU	%G1

Figure 50. Equating a Symbolic Address to an I/O Device

Imperative Statements

General Description. These are the symbolic instructions for the commands to be executed in the object computer. A source program will probably contain more of these imperative instructions than any other type of Autocoder statement.

The programmer:

1. Writes the mnemonic operation code for the instruction in the operation field.
2. Writes the operand(s) in the operand field. The first operand is the A- or I-operand; the second is the B-operand. A- and B-operands are literals or addresses of data fields. An A-operand can also be an input/output operand. An I-operand is the address of an instruction. If a d-character is required, it must be written at the immediate right of the operands.

   All items in the operand field must be separated by commas.

   Note: Several mnemonic operation codes have been developed which cause the d-character to be supplied automatically by the assembler. However, some operation codes (for example, BIN) have so many valid d-characters that it is impractical to provide a separate mnemonic for each. For these operation codes, the programmer must supply the d-character. In the listing of mnemonic operation codes for imperative instructions *(Appendix)*, all mnemonics that require a d-character in the operand field are indicated by two asterisks.
3. If the instruction is to be referenced, the programmer can label such an instruction. The label will have an equivalent address that is the storage location that will hold the operation code of the associated instruction when the object program is loaded. Thus, the label can be used as the I-operand of a branch instruction elsewhere in the program. (See Figure 51).

The assembler:

1. Substitutes the actual machine-language operation code for the mnemonic in the operation field.
2. Substitutes the 3-character equivalent machine address of the operands to indicate the A/I or B-address of the instructions.

   If address-adjustment or indexing codes are written with these operands, the appropriate alteration will be made for these addresses. Tag bits will be inserted in the tens position of indexed operands. Address-adjusted operands will be modified by adding or subtracting the adjustment factor. The assembler will supply the d-character for unique mnemonics, or place in the instruction the d-character from the operand field of the Autocoder statement if the programmer has supplied it.
3. Assigns to the actual-machine-language instruction an area in object core storage. The address of this area is the storage location the operation code will occupy when it is loaded into the object machine at program-load time. This address is the equivalent address of the label if one appears in the label field

of the source-program statement that contains the instruction.

Result. The instruction is loaded with a word mark in the high-order position.

Examples. Figure 51 shows an imperative instruction with an I-operand. When the instruction is executed in the object program, a branch to the instruction whose label is START will occur. Assume that START has an equivalent address of 360. The instruction will be assembled B 360.

Label	Operation	OPERAND
	B	START

Figure 51. Unconditional Branch with a Symbolic I-Operand

Figure 52 shows an imperative instruction with A- and B-operands. This instruction, when executed, causes the contents of ACCUM to be added to the contents of TOTAL. Assume that the equivalent addresses of ACCUM and TOTAL are 495 and 520, respectively. The assembled machine-language instruction is A 495 520.

Label	Operation	OPERAND
	A	ACCUM,TOTAL

Figure 52. ADD Instruction

Figure 53 shows an imperative instruction with I- and B-operands and a mnemonic (BCE), which requires that the programmer supply the d-character (5) in the operand. When this instruction is executed in the object program, a branch to the instruction whose label is READ will occur if the location labeled TEST contains a 5. Assume that the equivalent address of READ is 596 and TEST is 782. The assembled instruction is B 596 782 5.

Label	Operation	OPERAND
	BCE	READ,TEST,5

Figure 53. BRANCH IF CHARACTER EQUAL

Figure 54 shows an imperative instruction with a unique mnemonic (BAV). The assembler supplies the d-character (Z) for this instruction when it is assembled. Assume that OVFLO is assigned an equivalent address of 896. If, when the program is executed, an arithmetic overflow occurs, the first instruction causes a branch to OVFLO. The assembled instruction is B 896 Z.

Label	Operation	OPERAND
	BAV	OVFLO
	.	
	.	
	.	
OVFLO	ZA	FIELDA,FIELDB

Figure 54. BRANCH IF ARITHMETIC OVERFLOW

CU, LU, and MU Mnemonics

These mnemonics permit the programmer to code instructions for systems equipped with special features and devices that are not otherwise provided for in this Autocoder.

CU — Control Unit

The programmer:

1. Writes CU in the operation field.
2. Writes the address of the unit in the operand field in the format %Xn, d. A symbolic operand may be used to represent the address of the unit, if that symbolic operand has been defined by an EQU statement elsewhere in the source program.

The assembler: Provides a five-character instruction with the operation code U.

LU — Load Unit

The programmer:

1. Writes LU in the operation field.
2. Writes the address of the unit in the operand field in the format %Xn, BBB, d. A symbolic operand may be used to represent the address of the unit, if that symbolic operand has been defined by an EQU statement elsewhere in the source program.

The assembler: Provides an eight-character instruction with the operation code L.

MU — Move Unit

The programmer:

1. Writes MU in the operation field.
2. Writes the address of the unit in the operand field in the format %Xn, BBB, d. A symbolic operand may be used to represent the address of the unit, if that symbolic operand has been defined by an EQU statement elsewhere in the source program.

The assembler: Provides an eight-character instruction with the operation code M.

Machine Language Coding

Autocoder permits the programmer to use actual machine-language operation codes and d-characters.

The programmer:

1. Writes in column 19 the actual machine-language operation code for the instruction. Columns 16, 17, and 18 must be left blank.
2. Writes in column 20 the d-character in machine language. If no d-character is required, column 20 must be left blank.
3. May write a label in the label field.
4. Writes an actual, symbolic, blank, or asterisk address in the operand field. The operand field must not contain the d-character.

   The actual address of an input/output unit must be used unless a label has been assigned to the unit in an EQU statement.

Example. Figure 55 shows machine-language coding for an operation involving the IBM 1012 Tape Punch. Figure 55 also shows the same instruction coded in Autocoder. Either statement will cause the assembler to produce the instruction: M %P1 754 W if the equivalent address of LABEL is 754.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	MW	%P1,LABEL
		OR
	MU	%P1,LABEL,W

Figure 55. IBM 1012 Tape Punch Instructions

Assembler Control Statements

These are the Autocoder statements that permit the programmer to exercise some control over the assembly process:

Operation Code	*Purpose*
JOB	Job Card
CTL	Control Card
ORG	Origin Assembly
LTORG	Literal Origin
EX	Execute
XFR	Transfer
END	End Assembly
SFX	Suffix
ULST	Stop Listing
LIST	Start Listing
SPCE	Space n Lines
SKIPN	Skip to next page

JOB — Job Card

General Description. This card tells the assembler how to identify the program in the output listing from the assembly process. It also identifies the object program.

The programmer:

1. Writes JOB in the operation field.
2. Writes in the operand field the indicative information to be printed in the heading line of the output listing. Any combination of valid characters may be written in this statement (within columns 21-72).
3. Writes in the identification field (columns 76-80) the identification name or number that refers to the program.

The assembler:

1. Prints the information contained in the operand field of the JOB card, the identification number, and a page number in the heading line of each page of the output listing. If the source deck does not contain a JOB card, the assembler prints only the page number.
2. Punches the identification number in columns 76-80 of all condensed cards it produces for the object program.
3. If several JOB cards appear in the source deck, the assembler changes the information in the heading line and in the object program to reflect the new JOB identification. A new JOB card also causes the printer carriage to restore so that the new job or program starts on a new page of the output listing.

Result: Different programs or program overlays are easily identified in the output listing.

Example: Figure 56 shows a JOB card prepared for a program identified as EMPLOYEE PAYROLL REGISTER. It is identified in the object program as PRLRG.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	JOB	EMPLOYEE PAYROLL REGISTER

Figure 56. JOB Card

CTL — Control Card for Assembly

General Description: The CTL card describes the configuration of the object machine and specifies whether or not the cross reference listing, label table, and diagnostic messages are to be printed. The cross reference listing shows each label, its core storage address, and the sequence numbers of each line on the program listing that refers to it. The label table lists all labels and their core storage addresses; the diagnostic messages list the invalid source statements and the reasons for their invalidity.

The format of the CTL card is shown in Figure 57. The CTL card may be partially punched or omitted.

Columns	Indicates	Punch (Meaning)	Assumptions If the Columns Are Left Blank
16-18	Mnemonic operation code	CTL	
21	Object machine size	1 (4K) 2 (8K) 3 (12K) 4 (16K)	4K
22	Modify address feature available	1 (yes)	No, if the object machine has 4K; Yes, if the object machine has 8, 12, or 16K.
23	Advanced programming or indexing and store address register feature available	1 (yes)	No
24	Multiply-divide feature available	1 (yes)	No
25	Object machine	0 (1401) 4 (1440) 6 (1460)	Processor machine
26	Punch device	S (1442, 1444) P (1402)	S if the object machine is a 1440; P if the object machine is a 1401 or 1460
27	Read device	S (1442) P (1402)	S if the object machine is a 1440; P if the object machine is a 1401 or 1460
28	* Print device	S (1443) P (1403)	S if the object machine is a 1440; P if the object machine is a 1401 or 1460
29	Disk device	1 (1311, 1301) 2 (1405)	1311 or 1301
30	Source Statement Diagnostics	1 (yes) N (no)	Yes
31	Label Table or Cross Reference Listing	L (Label Table) N (neither)	Cross Reference Listing
32-36	** Object deck in the self-loading format, or Read-in area for a 1440 object deck in the condensed-loader format.	Sbbbb 5 digit starting address	Object deck in the condensed-loader format with the read-in area starting at 00001.
37-41	Loader location (These columns are not checked if column 32 contains an S.)	5 digit starting address. If column 42 contains a D, punch: 03701 (4K) 07701 (8K) 11701 (12K) 15701 (16K)	00075 if the object machine is a 1440; 00081 if the object machine is a 1401 or 1460
42	Disk Loader (for object programs in the coreload format)	D (yes)	No

* Consider a 1403 printer attached to a 1440 system as being the same as a 1443 printer.

** Object-program formats are described in Autocoder (on Disk) Program Specifications and Operating Procedures for IBM 1401, 1440, and 1460, Form C24-3259.

Figure 57. CTL Card Format

The figure shows the assumptions made by the assembler when columns are left blank. These assumptions are also made if the CTL card is omitted. If the CTL card is used, it must contain CTL in columns 16-18.

Notes:

1. The modify-address feature is standard on all IBM systems equipped with more than 4,000 positions of core storage.
2. Column 42 should contain a D if the object program is to be placed on disk in the coreload format. This will result from any of the following processor jobs:
   a. AUTOCODER RUN THRU EXECUTION (load and go)
   b. AUTOCODER RUN THRU OUTPUT (conventional assembly) with a CORELOAD OPTN
   c. OUTPUT RUN THRU EXECUTION (partial processing)
   d. OUTPUT RUN (partial processing) with a CORELOAD OPTN
3. If an object program in the condensed loader format is desired in addition to or in place of one in the coreload format, the card loader begins at the position specified in columns 37-41.
4. The only statements that may be placed between RUN and CTL cards are a JOB card and comments cards.

ORG — Origin

General Description: Use an origin card to tell the assembler the address at which to begin allocating storage for the program or for a particular part of the program (program overlay). An ORG statement may be included anywhere in the source program (except within a DA entry). If no ORG statement precedes the first entry in the source program, the assembler automatically begins allocating storage locations, starting at address 334 for 1401 and 1460 systems, and at address 210 for 1440 systems.

The programmer:

1. Writes ORG in the operation field.
2. Writes a symbolic, actual, blank, or asterisk address in the operand field. This address indicates the next storage location to be assigned by the assembler. Symbolic, actual, or asterisk addresses can have address adjustment. An operand in an ORG statement cannot be indexed and must be greater than zero.
3. If a symbolic address is used in the operand field of an ORG statement, its corresponding label must be defined ahead of it in the symbolic program.

The assembler: assigns addresses to instructions, constants, and work areas beginning at the address specified in the operand field of the ORG statement.

If the assembler encounters an ORG statement anywhere in the source program, it begins allocating storage for subsequent entries beginning at the address specified in the operand field of the new ORG statement.

Result: The programmer can choose the area(s) of core storage where the object program will be located.

Examples: Figure 58 shows an ORG statement with an actual address.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	ORG	500

Figure 58. ORG Statement with an Actual Address

The assembler will assign storage to the first source-program entry following this ORG statement with storage-location 500 as a reference point. This means that if the first entry following the ORG statement is an instruction, the Op-code position of that instruction will be 500. If the first entry is a 5-character DCW, it will be assigned address 504.

The ORG statement in Figure 59 shows how to instruct the assembler to save the address of the last storage location allocated. This ORG statement causes the assembler to equate the label to the address, plus 1, of the last storage location assigned before the ORG statement. The assembler continues assigning addresses beginning at the equivalent address of START.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
ADDR	ORG	START

Figure 59. Saving the Address of the Last Storage Allocation

Another ORG statement may be used later in the source program to direct the processor to begin assigning storage locations at ADDR (Figure 60).

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	ORG	ADDR

Figure 60. ORG Statement with a Symbolic Address

Figure 60 shows an ORG statement that directs the assembler to start assigning addresses with the actual address assigned to ADDR.

When the assembler encounters the statement shown in Figure 61, it will begin assigning addresses to subsequent entries in the source program at the next available storage location whose address is a multiple of 100. For example, if the last address assigned was 525, the next instruction (if the next entry is an instruction) will have an address of 600. It is possible to use additional address-adjustment factors with X00. For example, ORG *+X00−9 will give an address of 591.

Label	Operation	OPERAND
	ORG	*+X00

Figure 61. Adjustment to Next Available Century Block

If the object machine has an IBM 1443 Printer and does not have the print storage special feature, the print area must begin in a *hundreds +1* core storage position (101, 201, 301, etc.). Such a print area can be defined by an ORG statement with an operand of *+X00+1, followed by a DA statement (Figure 62).

Label	Operation	OPERAND
	ORG	*+X00+1
PRINT1	DA	1X120,6

Figure 62. Defining a Print Area

Note: +X00 is permitted as an adjustment factor only when it is used with an asterisk, and it may be used only in an ORG or LTORG statement.

Figure 63 shows an ORG statement with an asterisk and an address-adjustment factor in the operand field. The asterisk represents the address, plus one, of the last storage position assigned by the assembler. If the last address assigned was 525, the assembler will start assigning addresses at 591 (526 + 65).

Label	Operation	OPERAND
	ORG	*+65

Figure 63. ORG Statement with an Address-Adjusted Asterisk Operand

An ORG statement with a blank operand field may be used. It will cause the assembler to start assigning addresses beginning with the first address (beyond 333) after the highest address already assigned to other entries.

LTORG — Literal Origin

General Description: The programmer codes LTORG statements in the same way as ORG statements. A LTORG statement directs the assembler to begin assigning storage locations to literals, address constants, and closed library routines (see *Macro System*), which have been written ahead of the LTORG statement in the source program. The address of the storage location, which is the first to be allocated for a literal or closed library routine, is written in the operand field of a LTORG statement. A LTORG statement may be included anywhere in the source program.

If the assembler does not find a LTORG statement in the source program, it begins literal origin after finding an EX or END statement.

Example: Figure 64 shows how to direct the assembler to begin assigning storage locations to literals and closed library routines.

Label	Operation	OPERAND
	ORG	500
WKAREA	DCW	#8
CALC	EQU	1500
	ZA	+10,WKAREA
	INCLD	SUB01
	B	SUB01
ADDR	LTORG	CALC
	ORG	ADDR
FIELDA	DCW	#6
FIELDB	DCW	#5
	ZA	FIELDA,FIELDB

Figure 64. Using a LTORG Statement

The programmer has instructed the assembler to begin storage allocation at 500. All instructions, constants, and work areas (ending with B SUB01) will be assigned storage. However, the literal (+10) in the statement ZA +10 WKAREA, and the library routine (SUB01) extracted by the INCLD macro (see *INCLD*), will not be assigned storage until the LTORG statement is encountered. The first instruction in the library routine (SUB01) will be assigned address 1500 (V00) because CALC has been equated to 1500.

After all instructions in SUB01 have been assigned storage locations, the literal +10 will be assigned an address. The assembler will begin assigning the rest of the instructions, constants, and work areas with the storage location immediately following the area occupied by the instruction B SUB01. Thus, if a B SUB01 (B V00) is assigned locations 591-594, FIELDA will be assigned storage locations 595-600.

EX — Execute

General Description: An EX statement makes it possible to interrupt the object-program-loading process temporarily so that the part of the program that has already been loaded can be executed.

The programmer:

1. Writes EX in the operation field.
2. Writes a symbolic operand. This must be identical to the label used for the first instruction to be executed after the loading process has been halted.

The assembler:

1. Assembles an unconditional-branch instruction for 1440 systems and, if the self-loading format is specified, for 1401 and 1460 systems. A clear-and-branch instruction is assembled for 1401 and 1460 systems if the condensed-loader format is specified. The I-address of the instruction assembled is the equivalent address of the first instruction to be executed after the loading process has been halted. This instruction does not become part of the object program. However, it is used by the loading routine to transfer control to the object program.
2. Causes literals and closed library routines that have previously been encountered to be included at this point in the object program.

   Note: To continue the loading process after the program overlay has been executed, the programmer must provide re-entry to the load routine by writing the appropriate instruction(s) before the EX statement. The instructions are:

   A. 1401-1460 condensed-loader format. Branch to the starting address of the loader. If the loader has not been relocated (CTL card), the starting address is 081.

   B. 1440 condensed-loader format. Branch to the starting address of the loader, plus eight. If the loader has not been relocated (CTL card), the starting address is 075.

   C. 1401-1460 self-loading format. If the read-in area has not been disturbed during execution of the program overlay, read a card and branch to 040 (1040).

   If the read-in area has been disturbed:

   1. Clear the read-in area.
   2. Set word marks in 001, 040, 047, 054, 061, and 068.
   3. Read a card and branch to 040.

   D. 1440 self-loading format. If the read-in area has not been disturbed during the program overlay, branch to 073. If the read-in area has been disturbed:

   1. Clear the read-in area.
   2. Set word marks in 001, 040, 047, 054, 061, 068, 072, 073, 081, and 085.
   3. Positions 72-84 must contain ⧧M%G1001RB040.
   4. Branch to 073.

   E. 1401-1440-1460 coreload format. Use the LDRCL macro. See *Linkage Macros.*

All object program formats are described in *Autocoder (on Disk) Program Specifications and Operating Procedures for IBM 1401, 1440, and 1460,* Form C24-3259.

Example. Figure 65 shows how an EX statement can be coded. When the loader encounters the branch instruction produced by the assembler, the loading process stops and a branch to the instruction whose label is ENTRYA occurs.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	EX	ENTRYA

Figure 65. EX Statement

SFX — Suffix

General Description: This statement directs the assembler to put a suffix code in the sixth position of all labels and symbolic operands that have five or fewer characters, until another SFX statement is encountered. In this way, the programmer can use the same label in different sections of the complete program.

In using the INCLD macro (see *INCLD Macro*), the same routine can be extracted more than once because it is used in different program sections (there is a LTORG or EX statement between the two INCLD macros). In these cases use the SFX statement to ensure that the label does not appear exactly the same in two different sections of the program. Thus, the suffix code makes the labels in each section unique.

The suffixing can be discontinued by an SFX statement with a blank operand. To prevent a particular label from being suffixed within a portion of the program in which the other symbols are being suffixed, make the label six-characters long.

The programmer:

1. Writes SFX in the operation field.
2. Writes the character, which can be any valid character, to be used for the suffix code in the operand field.

The assembler:

1. Inserts the suffix code in the sixth position of all labels in the source program that have fewer than six characters.
2. Changes the suffix code when a new SFX card is encountered.

Result: Each program section has unique labels.

Example: Figure 66 is an example of coding for a suffixing operation.

Label	Operation	OPERAND
	SFX	A
ENTRY	ZA	FELDA,FELDB

Figure 66. Specifying a Suffix Operation

XFR — Transfer

General Description: This entry is like EX except that it does not signal the assembler to include literals and closed library routines in the object program.

END — End

General Description: The END statement signals the assembler that all of the source-program entries have been read. This card, which is always the last card in the source-program deck, provides the assembler with the information necessary to produce a branch instruction. The branch instruction in turn causes a transfer to the first instruction to be executed after the object program has been loaded.

The programmer:

1. Writes END in the operation field.
2. Writes an actual or symbolic address in the operand field. This must be the same symbol as the label of the first instruction to be executed after the loading processor has been completed.

The assembler:

1. Assembles an unconditional-branch instruction for 1440 systems, and a clear-storage-and-branch instruction for 1401 and 1460 systems. The I-address of this instruction is the equivalent address of the first instruction to be executed after the loading process has been completed. This instruction does not become part of the object program. However, it is used by the loading routine to transfer machine instruction execution to the object program.
2. Causes literals and closed library routines that have previously been encountered to be included at this point in the object program.

Result: Object-program execution begins automatically after loading.

Example: Figure 67 shows an END statement.

Label	Operation	OPERAND
	END	START

Figure 67. END Statement

ULST — Stop Listing

General Description: This operation stops the output listing of specified portions of the program. All other output options are not affected.

The programmer:

1. Writes ULST in the operation field.
2. Inserts the ULST card at the beginning of the section that is not to be listed.

The assembler:

1. Stops printing the output listing.
2. Indicates that this portion of the listing is being skipped.

Example: Figure 68 shows an ULST statement.

Label	Operation	OPERAND
	ULST	

Figure 68. ULST Statement

LIST — Start Listing

General Description: To resume listing after an ULST operation has been in effect, the LIST operation is specified.

The programmer:

1. Writes LIST in the operation field.
2. Inserts the card at the end of the section which has not been listing.

The assembler: resumes printing the output listing.

Example: Figure 69 shows a sample LIST statement.

Label	Operation	OPERAND
	LIST	

Figure 69. LIST Statement

SPCE — Space n Lines

General Description: This operation causes the assembler to insert extra spaces in the output listing.

The programmer:

1. Writes SPCE in the operation field.
2. Writes the numeric character 1, 2, or 3 in column 21. Use 1 for no space before printing, that is, single space printing, 2 for one space before printing, that is, double space printing, and 3 for two spaces before printing, that is, triple space printing.

3. Inserts the SPCE card following the card after which the spacing is to start.

The assembler: Leaves the specified number of spaces after each line printed until another SPCE card is encountered. If no SPCE card is included in the source deck, the assembler will not leave any spaces between printed lines.

Example: Figure 70 shows a space statement that causes the assembler to leave one space between lines in the output listing.

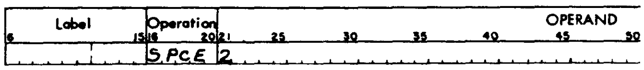


Figure 70. SPCE Statement

SKIPN—SKIP TO NEXT PAGE

General Description: This operation causes the assembler to skip to the next page of the printed output listing. Thus, the programmer can force the start of a new listing page without having to use a JOB card.

The programmer: Writes SKIPN in the operation field.

The assembler: Skips to the next page in the output listing.

Example: Figure 71 shows a SKIPN statement.

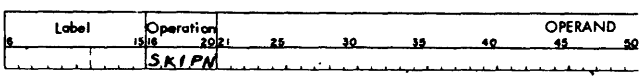


Figure 71. SKIPN Statement

ENT — Enter New Coding Mode

General Description: An ENT statement is used by the programmer to inform the assembler that a change in coding form follows. The END card must be processed in the full Autocoder mode.

The Autocoder assembler accepts source programs coded in any of these three formats:

1. The standard free-form Autocoder format described in the *Coding Sheet* section of this publication.
2. The fixed-form SPS language format described in *IBM 1401 Symbolic Programming Systems: SPS-1 and SPS-2,* Form C24-1480.
3. The free-form Basic Autocoder format described in *Basic Autocoder for IBM 1440: Specifications,* Form C24-3023.

The programmer:

1. To enter Basic Autocoder from full Autocoder, writes ENT in columns 16-18 and writes BASIC in columns 21-25.
2. To enter full Autocoder from Basic Autocoder, writes ENT in columns 36-38, and writes AUTOCODER in columns 41-49.
3. To enter SPS: from full Autocoder, write ENT in columns 16-18, and write SPS in columns 21-23. (SPS statements are assembled into 1401-1460 machine-language coding.)
4. To enter full Autocoder from SPS, writes ENT in columns 14-16 and writes AUTOCODER in columns 17-25.

The assembler: Interprets the source-program coding as identified by the ENT statements.

Result: Programs prepared partially in SPS or Basic Autocoder format can be reassembled by the Autocoder assembler.

Examples. Figures 72, 73, 74, and 75 are ENT statements to be used with Autocoder.

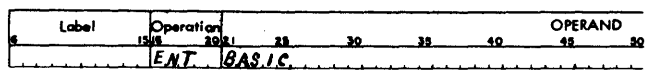


Figure 72. Enter Basic Autocoder from Autocoder

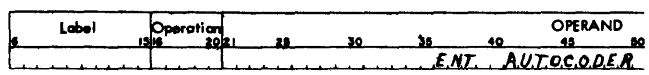


Figure 73. Enter Autocoder from Basic Autocoder

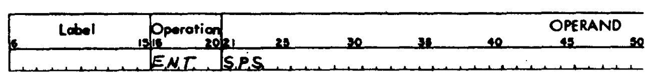


Figure 74. Enter SPS from Autocoder

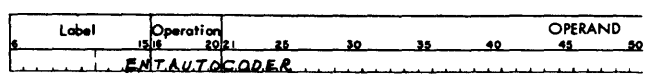


Figure 75. Enter Autocoder from SPS

Many program routines are quite general. These routines (consisting of a series of instructions originally developed to handle one phase of one specific program) can, with little or no alteration, be incorporated in other programs. For example, a routine for checking the accuracy of a write-disk operation can be used, with modification of addresses, in many programs.

The Autocoder system includes a macro facility that eliminates the repetitive coding of general routines. Before the macro facility can be used, the user must create a *library* by storing the routines in disk storage. The user can then write a single symbolic instruction (a macro instruction) that causes the assembler to extract the routine associated with the instruction, tailor it to fit the program requirements, and insert it in the object program.

IBM provides several macro instructions and library routines. Others can be developed by the user, then stored in the library and incorporated into programs as needed.

Library Routines

A library routine is a complete set of instructions designed to perform a specific operation. The name of a library routine is referred to as a *macro name.* This name is used as a header label in the disk-storage record that contains the routine. It is also used to specify the routine in a macro instruction. Each library routine must have a unique macro name. A source program cannot contain more than 99 macro names. This is the maximum number of routines that can be in the Autocoder library.

Library routines are written on a coding form designed to organize them into the format required by the assembler. Figure 76 shows the library coding form. (See *Developing the Library Routine.)*

During the *librarian* phase of Autocoder, the routines are transferred to the disk-storage library. (See *INSER – Insert* and *DELET – Delete.)* At program assembly time the required routines are extracted, tailored to fit program requirements if necessary, and inserted in the symbolic source program. The source program, including the symbolic library entries, is then processed by the assembler to produce the machine-language object program.

Flexible Library Routines

A routine that can be tailored to fit program requirements is a *flexible* library routine. These routines consist of *model statements* that are general outlines for symbolic-program statements. During program assembly, the *macro-generator* phase of the assembler program replaces the codes in model statements with the *parameters* (symbolic addresses, control fields, or other information) specified in the source-program macro instructions. Model statements can be deleted if they are not needed in the program.

Flexible library routines may contain *pseudo macro instructions.* These are commands to the macro generator that control the production of the symbolic routine. Pseudo macros are never used by the source programmer. They are used by the library programmer when he develops the library routine.

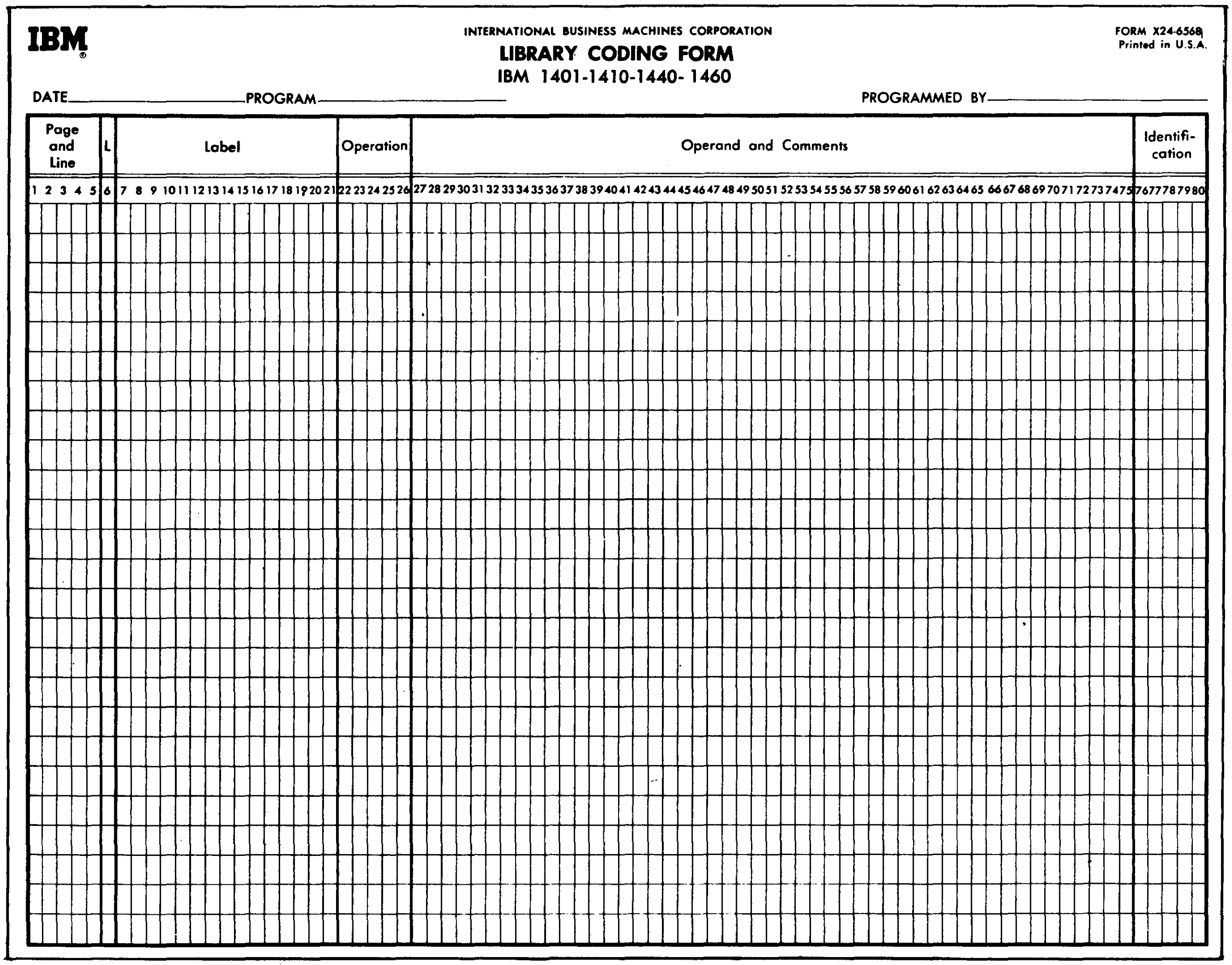
Inflexible Library Routines

A library routine that requires no alteration is an inflexible library routine. All the instructions (model statements) are incorporated in the symbolic program. No parameters may be inserted. The data needed by the routine must be in the locations indicated by the symbolic addresses in the operand fields of its instructions. An inflexible library routine is called an INCLD routine because the INCLD macro instruction causes the assembler to insert it in the symbolic source program.

Macro Instructions

General Description. A macro instruction is the entry in the source program that specifies the routine to be extracted from the library and inserted in the program. It also gives the assembler the information necessary to tailor a flexible library routine.

An INCLD or CALL macro instruction must be used to insert an inflexible library routine in the program. A regular macro instruction (contains the name of the library routine in the operation field) is used to tailor and incorporate a flexible library routine. The following discussion applies to regular macro instructions. (See *INCLD Macro* and *CALL Macro.)*


IBM
INTERNATIONAL BUSINESS MACHINES CORPORATION
LIBRARY CODING FORM
IBM 1401-1410-1440-1460
FORM X24-6568
Printed in U.S.A.
DATE ____ PROGRAM ____ PROGRAMMED BY ____

Page and Line	L	Label	Operation	Operand and Comments	Identification
1 2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75	76 77 78 79 80

Figure 76. IBM Autocoder Library Coding Form

The source programmer:

1. Writes the macro name (the name of the library routine) in the operation field of the Autocoder coding sheet.
2. Writes in the label field the name that is to be used as the label of the first statement in the generated symbolic routine.
3. Writes in the operand field the parameters that are to be used by the model statements required for the particular object routine as follows:
   a. Parameters must be written in the sequence in which they are used by the codes in the model statements. For example, if COST is parameter 1, it must be written so that it will be substituted wherever a ⌑01, or ⌑0A appears in a label, operation code, or operand field of a model statement.
   b. A macro instruction may have as many parameters as can be written in the operand fields of five or fewer coding-sheet lines. If more than one coding-sheet 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. A parameter may not contain blanks or commas unless they are enclosed by @ symbols (as in an alphameric literal). The @ symbol itself must not appear between @ symbols in a parameter.

   If more than one line is needed to list the parameters for a given macro instruction, a comma must be written after the last parameter of each line. A comma is not needed after the last parameter listed for the macro.
   c. A parameter, or parameters, may be omitted if not required for the object routine. To omit a parameter, include the comma that would have followed the parameter, unless the parameter to be omitted follows the last parameter used in the macro instruction. The assembler uses these commas to count parameters up to and including

the last included parameter. All parameters between the last one included and parameter 99 are assumed by the assembler to be absent.

The assembler:

1. Extracts the library routine and selects the model statements required for the routine as specified by the parameters in the macro instructions, by the substitution and condition codes in the model statements, and by the pseudo macros in the library routine.
2. Substitutes parameters when they are indicated in the model statements, producing the symbolic routine.

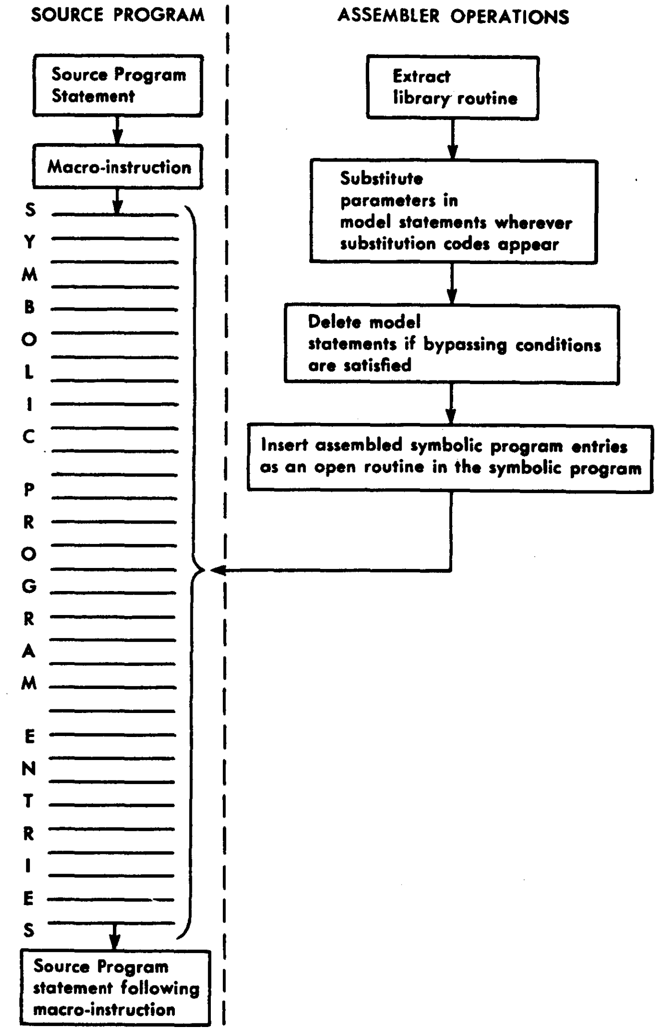
Result. The symbolic routine is merged into the symbolic program following the macro instruction. This routine is called an *open* or *in-line* routine because it is inserted directly into a larger routine without linkage or calling sequence.

Figure 77 shows the effect of a regular macro instruction.

Example. To illustrate the basic operation of the macro system, a hypothetical macro named CHECK, with a simple flexible library routine, is shown here.

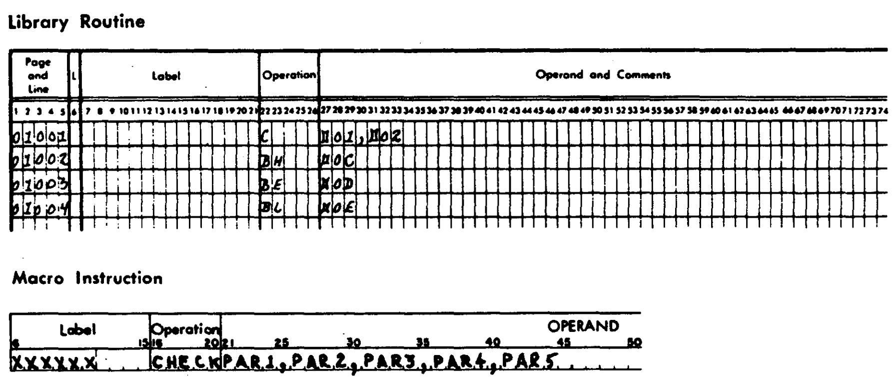
This routine is designed to compare an input field to another field, and to test the compare indicators for a high, low, or equal condition (or any combination of the three) as prescribed by the macro instruction in the source program. For example, the source programmer may use the object routine to test only for an equal condition in one program; in another, high or equal.

Figure 78 shows the library routine and a sample macro instruction specifying that a routine using all



When a regular macro instruction is encountered in the source program, the assembler extracts the specified library routine, tailors it, and inserts it in-line in the user's source program.

Figure 77. Macro Processing



Generated Symbolic Routine

```
XXXXXX C     PAR1,PAR2
       BH    PAR3
       BE    PAR4
       BL    PAR5
```

Figure 78. Macro Operations

Label	Operation	OPERAND
	EXACT	FLD1,FLD2,FLD3,FLD4,FLD5,FLD6,FLD7,FLD8,FLD9

Figure 79. Parameters for EXACT included; Parameters 10-99 Missing

Label	Operation	OPERAND
	EXACT	FLD1,FLD2,FLD3,,FLD5,FLD6,FLD7,,FLD9

Figure 80. Parameters 04, 08, and 10-99 Missing

the model statements is needed in the object program. The symbolic routine generated by the assembler is also shown. The symbolic routine is inserted in the symbolic program following the macro instruction. During assembly of the object program, the symbolic program will be translated to actual machine-language instructions. The actual addresses of the symbols supplied as parameters in the macro instruction will be inserted in the label, operation, and operand fields.

Examples. Figures 79, 80, 81, and 82 show how parameters can be omitted. A hypothetical macro instruction called EXACT is used. EXACT can have as many as nine parameters.

Label	Operation	OPERAND
	EXACT	,FLD2,FLD3,,,,FLD7,,FLD9

Figure 81. Parameters 01, 04, 05, 06, 08, 10-99 Missing

Label	Operation	OPERAND
	EXACT	,FLD2

Figure 82. Parameters 01 and 03-99 Missing

INCLD Macro

General Description. This macro extracts an inflexible library routine from the disk-storage library. The programmer establishes his own linkage to the closed routine.

The source programmer:

1. Writes INCLD in the operation field.
2. Writes the name of the library routine in the operand field.

The assembler:

1. Extracts the library routine at Literal Origin time. If no LTORG statement appears in the user's source program, the library routine is included in the program when the assembler encounters an EX or END statement.
2. Incorporates the library routine only once per program or overlay, regardless of how many INCLD statements name the same routine.

Note: The programmer must insert a branch instruction at the place in the main routine at which the exit to the library routine is needed. Several INCLD statements can be written in a group in the source program to cause the associated library routines to be incorporated by the assembler at LTORG, END, or EX time. Thus, one exit from the main routine can be used to cause several library routines to be executed at object-program execution time.

Note: There can be no more than 30 INCLD statements within any one program overlay.

Result. An inflexible library routine is included in the symbolic source program. This routine is called a *closed* or *out-of-line* routine because it is entered by a basic linkage (a branch instruction) from the main routine.

Figure 83 shows the effect of an INCLD macro instruction.

Example. Figure 84 shows an INCLD statement used to extract an inflexible library routine named SUBRT1.

Label	Operation	OPERAND
	INCLD	SUBRT1

Figure 84. INCLD Macro

CALL Macro

General Description. The CALL macro provides linkage to inflexible (closed) library routines and generates the INCLD statement needed to incorporate the routine in the source program.

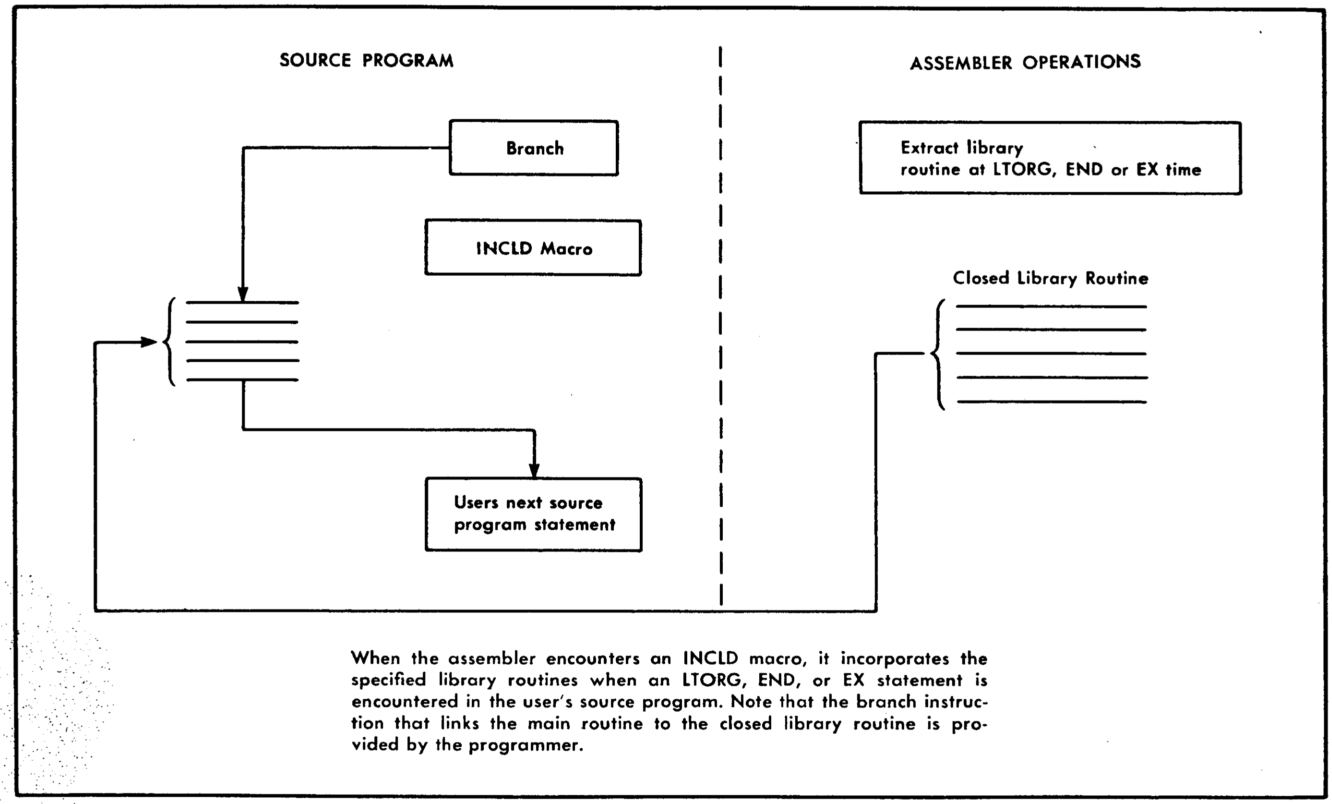


Figure 83. INCLD Processing

The source programmer:

1. Writes CALL in the operation field.
2. Writes the name of the routine in the operand field. The routine name must also be the label of the first instruction to be executed.
3. May write a maximum of ten operands immediately after the routine name. The assembler generates a DCW for each of these operands so that they can be used as labels or data in the routine.

The assembler:

1. Advanced programming or indexing-and-store-address-register feature not available:

   a. Generates a label and a move instruction. When the program is executed, the equivalent address of the label is moved to a three-character field immediately ahead of the first instruction to be executed. (In the routine, the statement that precedes the first instruction to be executed must define the three-character field.)

      If any DCW's are generated, the equivalent address of the label is the address of the units position of the first DCW.

      If no DCW's are generated, the equivalent address of the label is the address of the instruction that follows the CALL statement in the source program.

      Because an address is stored in the three-character field, the library programmer can plan the use of the generated DCW's and prepare linkage back to the main routine.

   b. Generates a branch instruction to the first instruction to be executed.

   c. Generates an INCLD statement.

   d. Generates DCW's for the operands that follow the routine name. The DCW's immediately follow the branch instruction in the object program.

Advanced programming or indexing-and-store-address-register feature available:

a. Generates a branch instruction to the first instruction to be executed. The first statement to be executed must be an SBR instruction.

   If any DCW's are generated, the address stored is the address of the high-order position of the first DCW.

   If no DCW's are generated, the address stored is the address of the instruction that follows the CALL statement in the source program.

The SBR instruction enables the library programmer to plan the use of the generated DCW's, and to prepare linkage back to the main routine.

b. Generates an INCLD statement.

c. Generates DCW's for the operand that follows the routine name. The DCW's immediately follow the branch instruction in the object program.

2. Extracts the library routine at Literal Origin time. If no LTORG statement appears in the user's source program, the routine is included in the program when the assembler encounters an EX or END statement.
3. Incorporates the routine only once per program or program overlay, regardless of how many INCLD or CALL statements name the same routine.

Example. Figure 85 shows a CALL statement with three operands and the statements generated by the assembler. Assume that the object machine does not have the advanced programming or indexing-and-store-address-register feature. The figure also shows the portion of the library routine that utilizes the three-character field immediately ahead of the first instruction in the routine.

Library Routine

Page and Line (1-5)	L (6)	Label (7-21)	Operation (22-26)	Operand and Comments (27-59)
01001		SUBADR	DCW	@000@
01002		SUBRT	MLC	SUBADR,SUBRT1+3
01003		SUBRT1	MLC	0,TOTAL
			.	
			.	
			.	
01019			MA	@004@,SUBADR
01020			MLC	SUBADR,EXIT+3
01021		EXIT	B	0

Macro Instruction

Label (6-15)	Operation (16-20)	OPERAND (21-50)
	CALL	SUBRT,+NAME,@430@

Generated Symbolic Routine

```
        MLC    +¤0J001,SUBRT-1
        B      SUBRT
¤0J001  DCW    +NAME
        DCW    @430@
         .       .
         .       .
         .       .
        INCLD  SUBRT
  (generated closed routine)
```

Figure 85. CALL Macro

Label	Operation	OPERAND
	MLC	A,B
	CHAIN	5

Generated Symbolic Program Entries

```
MLC   A,B
MLC
MLC
MLC
MLC
MLC
```

Figure 86. CHAIN Macro

CHAIN Macro

General Description. The CHAIN macro makes it easier for the programmer to code chained instructions.

The source programmer:

1. Writes the instruction to be chained.
2. Writes the macro instruction, using CHAIN as the mnemonic operation code, and writes a number from 1 to 99 in the operand field. This number represents the number of chained instructions desired.

The assembler: Repeats the operation code as many times as specified by the CHAIN macro.

Example. Figure 86 shows how an MLC statement can be chained five times.

MA Macro — Modify Address

General Description. The source programmer may use the MA macro instruction to modify a one- or two-address instruction, if the modify-address feature is not available in the object machine. The modify-address feature is an additional operation code (MA—MODIFY ADDRESS) that is standard on all IBM 1460 systems and on IBM 1401 and 1440 systems with more than 4000 positions of core storage.

The MA macro is supplied by IBM as part of the Autocoder library. The assembler determines if the machine instruction can be issued (if the modify-address feature is available), or if the symbolic routine (generated from the library routine) is to be inserted in the source program.

The source programmer:

1. Writes MA in the operation field.
2. May write a label in the label field.
3. Writes the macro instruction with one or two operands. An alphameric literal used as an operand must be three characters.

The assembler:

1. Selects the model statements necessary to modify the correct address(es). The presence or absence of parameters in the source-program macro instruction determines which model statements are to be used.
2. Puts the label (if any) in the first instruction used for the address modification routine.

Result. Tailored symbolic-program statements are inserted as an open routine in the program.

Examples. Figure 87 shows a routine designed to move all the items from a card to their proper places in the area named TABLE, using a single move instruction (named SAVE) to perform all the necessary moves. Two MA macro instructions (READY and ADJUST) are used to modify the operands of the move instruction. In analyzing the routine, assume that every fifteenth column in each input card contains the last character of an item of information and that word marks have been previously set in the read area to identify the beginning of each item. Also, for the benefit of 1440 users, the read-and-branch instruction (R ADJUST) used in the routine is a 1401-1460 instruction that causes a card to be read and then a branch to be made to the address specified in the operand.

Figure 88 shows the MA macro instruction with a parameter for the A-address only. The symbolic routine generated by the assembler processor is also shown.

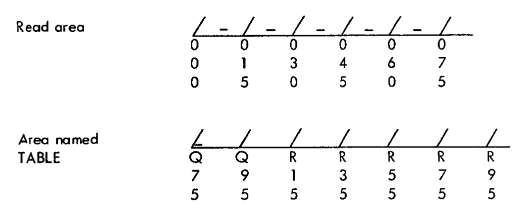
Source Program Statements

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TABLE	DCW	#120
	.	
	.	
	.	
	R	ADJUST
READY	MA	@015@, SAVE+3
ADJUST	MA	@020@, SAVE+6
SAVE	MLC	DATA, TABLE+14
	A	*-6, COUNT
	BCE	PRINT, COUNT, 5
	B	READY

Generated Symbolic Routines

READY	SW	SAVE+1
	A	@015@, SAVE+3
	CW	SAVE+1
ADJUST	SW	SAVE+4
	A	@020@, SAVE+6
	CW	SAVE+4

Name	Equivalent Address
DATA	015
READY	647
ADJUST	662
SAVE	677
COUNT	724
TABLE	Q75 (2875) The high-order position of the area is the equivalent address of TABLE because the label of the area-defining DCW begins in column 7.



	SAVE			SAVE+3			SAVE+6
Core-storage locations	677	678	679	680	681	682	683
Original SAVE instruction	M	0	1	5	Q	7	5
First SAVE instruction executed	M	0	1	5	Q	9	5
Second SAVE instruction executed	M	0	3	0	R	1	5
Third SAVE instruction executed	M	0	4	5	R	3	5
Fourth SAVE instruction executed	M	0	6	0	R	5	5
Fifth SAVE instruction executed	M	0	7	5	R	7	5

Figure 87. MA Macro with Two Parameters

Macro Instruction

Label (6–15)	Operation (16–20)	OPERAND (21–50)
ALTERB	MA	FIELDA

Generated Symbolic Program Entries

ALTERB	SW	FIELDA–2
	A	FIELDA
	CW	FIELDA–2

Figure 88. MA Macro with One Parameter

LOOP Macro

General Description. This macro generates instructions to execute a loop a specified number of times. This may be any number within the range 1-999. The LOOP macro is the last instruction in the loop.

The source programmer:

1. Writes LOOP in the operation field.
2. May write a label in the label field.
3. Writes the parameters in the operand field in this order:

   Parameter 1. The symbolic address of the first instruction in the loop.

Parameter 2. The symbolic address of a one-, two-, or three-position field that contains the number that indicates how many times the loop is to be executed. After looping is completed, the loop counter is reinitialized to the original number.

Parameter 3. The number that indicates how many times the loop is to be executed. After looping is completed, the loop counter is automatically reinitialized to the number specified.

Note: Use either parameter 2 or parameter 3, but not both. No reinitialization takes place on the loop counter if an exit is taken within the loop.

Example. The macro instruction shown in Figure 89 causes the program to branch to TEST3 eight times to execute the loop nine times.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	LOOP	TEST3,,9

Figure 89. LOOP Macro

COMPR Macro

General Description. This macro generates instructions to compare and test indicators for low, equal, or high results. Rules for word-mark control or low, equal, or high indication are the same as for the machine COMPARE instruction.

The source programmer:

1. Writes COMPR in the operation field.
2. May write a label in the label field.
3. Writes the parameters in the operand field in this order:

   Parameter 1. The symbol of the A-field to be compared.

   Parameter 2. The symbol of the B-field to be compared.

   Parameter 3. The symbolic address of the next instruction, if a branch occurs as a result of a low condition.

   Parameter 4. The symbolic address of the next instruction, if a branch occurs as a result of an equal condition.

   Parameter 5. The symbolic address of the next instruction, if a branch occurs as a result of a high condition.

   Note: Any or all of the parameters 3, 4, and 5 may be included for the COMPR macro.

Example. (Figure 90) Compare stock on hand (STOCK) to projected usage (USAGE). If the stock on hand is lower than the projected usage, branch to the reorder routing (REORDR).

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	COMPR	USAGE,STOCK,REORDR

Figure 90. COMPR Macro

Linkage Macros

Autocoder (on Disk) provides two linkage macros, LDRCL and SYSCL. The LDRCL macro facilitates the execution of object programs (or program overlays) that are to be loaded from disk (coreload format). The SYSCL macro enables the user to stack jobs (such as program assemblies, program executions, and librarian operations) under control of Autocoder (on Disk).

Note: Object-program formats and Autocoder jobs are described in *Autocoder (on Disk) Program Specifications and Operating Procedures for IBM 1401, 1440, and 1460,* Form C24-3259.

LDRCL Macro

General Description. The LDRCL macro enables the programmer to resume loading an object program from disk after a portion of the program has been executed. The machine size specified in the CTL card determines the location of the disk loader. The locations are 3701 for 4K, 7701 for 8K, 11701 for 12K, and 15701 for 16K. The LDRCL macro generates the appropriate branch instruction.

The programmer can also use the LDRCL macro to begin loading another independent object program that is in the coreload format.

The source programmer:

1. Writes LDRCL in the operation field of the macro instruction.
2. If another independent object program is to be loaded, from disk, the programmer must precede the LDRCL macro instruction with an instruction that will move the starting address of the next program to the core storage locations that contain the address of the next section to be read (3831 for 4K, 7831 for 8K, 11831 for 12K, and 15831 for 16K).

SYSCL Macro

General Description. The SYSCL macro causes the assembler to generate a branch instruction to the *bootback* routine, which transfers program control to the System Control Program after the object program (card format or coreload format) has been executed. The System Control Program reads the control card for the next job and initiates the processing required to perform the job.

The machine size specified in the CTL card determines the location of the *bootback* routine. The locations are 3928 for 4K, 7928 for 8K, 11928 for 12K, 15928 for 16K.

When used, the SYSCL macro should be the last instruction executed in the source program.

The source programmer: writes SYSCL in the operation field of the macro instruction.

Arithmetic Macros

These macros are incorporated in Autocoder to make it easier to program addition, subtraction, multiplication, and division.

The following information applies to all arithmetic macros:

1. Permanent switches set from information in the CTL card govern the uses of the indexing-and-store-address-register, modify-address, and multiply/divide features.
2. Any positive set of decimal-place configurations is considered valid. (This includes zeros.) They must be expressed as unsigned integers.
3. A literal may be used as a parameter wherever the the name of a field is required.
4. The fields from which values are obtained are not modified in any way. The symbols for these fields are used as parameters 1 and 4.
5. Rounding is performed by computing the result to one extra position of accuracy, and then adding five to the extra position.
6. Whenever rounding or editing is required, a temporary result field is used.
7. The result field need not be set to zeros before the macro routine is entered.
8. Actual decimal points appear only in edited results.
9. The absence of the sign-control parameter (12) causes shorter (and slightly faster) macro routines.
10. The result field must be large enough to contain the complete edit-control word.

ADD Macro

General Description. This macro produces a routine that adds two fields, and stores the result in a third field.

The source programmer:

1. Writes ADD in the operation field.
2. May write a label in the label field.
3. Writes parameters in the operand field in this order:

Parameter 1. The name of the first field to be added. This must be the field with the lesser number of decimal places unless both fields have the same number of decimal places.

Parameter 2. The length of the field specified in parameter 1 (number).

Parameter 3. The number of decimal places in the field specified by parameter 1. If there are no decimal places, use a zero.

Parameter 4. The name for the second field to be added. This must be the field with the greater number of decimal places unless both fields have the same number of decimal places.

Parameter 5. The length of the field specified in parameter 4 (number).

Parameter 6. The number of decimal places in the field specified in parameter 4. If there are no decimal places, use a zero.

Parameter 7. The name of the result (sum) field.

Parameter 8. If editing is not used, this number is the length of the result field. If editing is used, this number must correspond to the number of blanks and zeros in the edit-control word.

Parameter 9. The number of decimal places desired in the result.

Parameter 10. Truncate parameter (T). The T indicates that the result is not to be rounded. If parameter 10 is absent, the result will be rounded, provided the number of decimal places specified for the result is less than the number of decimal places specified for either of the two fields to be added.

Parameter 11. This may be either the name of an edit-control word for the result, or an edit-control word expressed as an alphameric literal.

Parameter 12. S indicates sign-control for negative and positive numbers. If parameter 12 is absent, numbers will be handled as positive, and must not have negative zones.

Note: Parameters 10, 11, and 12 are optional. All others must be present.

Example. (Figure 91) Add the contents of a field called CASH to the contents of a field called RECPTS, and store the result in a field called TOTALS.

CASH	XXXX.00
RECPTS	XXX.00
TOTALS	XXXXX.00

Label	Operation	OPERAND
	ADD	CASH,6,2,RECPTS,5,2,TOTALS,7,2

Figure 91. ADD Macro

SUB Macro

General Description. The subtract macro subtracts one field from another and stores the result in a third field.

The source programmer:

1. Writes SUB in the operation field.
2. May write a label in the label field.
3. Writes parameters in the operand field in this order.

Parameter 1. The name for the minuend field (quantity from which another field is subtracted).

Parameter 2. The length of the minuend (number).

Parameter 3. The number of decimal places in the minuend. Specify zero if there are no decimal places in this field.

Parameter 4. The name for the subtrahend (quantity to be subtracted from another field).

Parameter 5. The length of the subtrahend (number).

Parameter 6. The number of decimal places in the subtrahend. Specify zero if there are no decimal places in the field.

Parameter 7. The name for the result (difference) field.

Parameter 8. If editing is not used, this number is the length of the result field. If editing is used, this number must correspond to the number of blanks and zeros in the edit-control word.

Parameter 9. The number of decimal places in the result. Specify zero if there are no decimal places in this field.

Parameter 10. Truncate parameter (T). The T indicates that the result is not to be rounded. If parameter 10 is absent, the result will be rounded, provided that the number of decimal places specified for the result is less than the number of decimal places specified for either the minuend or the subtrahend.

Parameter 11. The name of an edit-control word for the result, or an edit-control word expressed as an alphameric literal.

Parameter 12. S indicates sign-control for negative and positive numbers. If parameter 12 is absent, the minuend and subtrahend will be handled as positive fields and therefore must not have negative zones. If a negative result is possible, sign-control should be used.

Note: Parameters 10, 11, and 12 are optional. All other parameters must be included.

Example. (Figure 92) Subtract a field called ISSUES from a field called INSTCK and store the result in a field called BALAN.

ISSUES	XXXX
INSTCK	XXXXXX
BALAN	XXXXXX

Multiply and Divide Macros

If the multiply/divide feature is included in the machine used to execute the object program, the multiply and divide macros will use it (if the feature has been specified in the CTL card). However, if this feature is not present in the object machine, the multiply and divide macros provide instructions to perform these operations.

MLTPY Macro

General Description. The multiply macro multiplies one field by another and stores the result in a third field.

The source programmer:

1. Writes MLTPY in the operation field.
2. May write a label in the label field.
3. Writes the parameters in the operand field in this order:

Parameter 1. Multiplier field (name). For maximum efficiency this should be the shorter field involved in the multiplication.

Label	Operation	OPERAND
	SUB	INSTCK,6,0,ISSUES,4,0,BALAN,6,0

Figure 92. SUB Macro

Parameter 2. Length of the multiplier field (number).

Parameter 3. Number of decimal places in the multiplier field (number).

Parameter 4. Multiplicand field (name).

Parameter 5. Length of the multiplicand field (number).

Parameter 6. Number of decimal places in the multiplicand field (number).

Parameter 7. Product field (name).

Parameter 8. If editing is not used, this number is the length of the result field. If editing is used, this number must correspond to the number of blanks and zeros in the edit-control word.

Parameter 9. Number of decimal places in the desired product field (number).

Parameter 10. Truncate parameter (T). The T indicates that the answer (product) is not to be rounded. The answer will be rounded if parameter 10 is missing, and if the number of decimal places in the product field desired is less than the sum of the number of decimal places in the multiplier and multiplicand fields.

Parameter 11. This parameter can be either the name of an edit-control word for the answer, or a control word expressed as an alphameric literal.

Parameter 12. This parameter is an S that indicates sign-control for positive and negative numbers. If parameter 12 is missing, numbers will be treated as positive and in this case, must not have negative zones.

Note: Parameters 3, 6, 9, 10, 11, and 12 are optional. However, parameters 3, 6, and 9 must all be included if any decimal number is used.

Example: (Figure 93) multiply a field called HOURS by a field called RATE, and store the result in a field called GROSS. EDTWD1 is used to edit the result field.

HOURS	XX.00
RATE	XX.00
GROSS	XXXX.00000

DIVID Macro

General Description. The divide macro divides one field into another and stores the result in a third field. The macro does not provide for division by zero. The user should test the divisor field before using the divide macro.

The source programmer:

1. Writes DIVID in the operation field.
2. May write a label in the label field.
3. Writes the parameters in the operand field in this order:

Parameter 1. Divisor field (name).

Parameter 2. Length of the divisor field (number).

Parameter 3. Number of decimal places in the divisor field (number).

Parameter 4. Dividend field (name).

Parameter 5. Length of the dividend field (number). If extra quotient digits are to be developed, the divide macro will insert low order zeros and shift the sign.

Parameter 6. Number of decimal places in the dividend field (number).

Parameter 7. Quotient field (name).

Parameter 8. If editing is not used, this number is the length of the result field. If editing is used, this number must correspond to the number of blanks and zeros in the edit-control word.

Parameter 9. Number of decimal places desired in the quotient field (number).

Parameter 10. Truncate parameter (T). The T indicates that the answer (quotient) is not to be rounded. The answer will be rounded if parameter 10 is missing, and if parameters 3, 6, and either 9 or 13 are present.

Parameter 11. This parameter is either the name of an edit-control word for the answer, or a control word expressed as an alphameric literal.

Parameter 12. This parameter is an S that indicates sign-control for positive and negative numbers. If parameter 12 is missing, numbers will be treated as positive and must not have negative zones.

Parameter 13. Remainder field (name). This parameter may be used with parameter 7 if both the quotient field and the remainder are desired. Parameter 7 may be omitted if only the remainder is desired. However, at least one of the parameters (7 or 13) must be included for the DIVID macro.

When the multiply-divide feature is specified, the sign of the remainder will be the sign of the

Label (6–15)	Operation (16–20)	OPERAND (21–72)
	MLTPY	RATE,4,2,HOURS,4,2,GROSS,9,5,,EDTWD1

Figure 93. MLTPY Macro

dividend. If the feature is not specified, the sign of the remainder will always be positive.

Note: Parameters 3, 6, 7, 8, 9, 10, 11, 12, and 13 are optional. If any decimal number is used, parameters 3, 6, and either 9 or 13 all must be included.

Example. (Figure 94) Divide a field called SUMS by a field called FACTOR, and store the result in a field called AVERAG.

SUMS	XXXX.00
FACTOR	XX.
AVERAG	XXX.000

Label	Operation	OPERAND
	DIVID	FACTOR,2,0,SUMS,6,2,AVERAG,6,3

Figure 94. DIVID Macro

Developing Library Routines

General Description. The library routine is a general routine designed to perform many specific functions (depending on the parameters supplied by the source programmer in his macro instruction) when it is executed in the object program.

The library routines needed for a given installation are prepared by the library programmer. In many cases the library programmer and the source programmer are the same person, but the two functions are separate and are thus treated here.

The librarian phases of Autocoder maintain the library by inserting, deleting, and/or modifying library routines. At assembly time, the macro-generator phases extract the routines named in macro instructions.

The library programmer:

1. Designs the general routine.
2. Writes the model statements needed in the routine.

The librarian: enters the model statements in disk storage immediately following the heading information contained in the associated INSER statement during the librarian phase of Autocoder.

Result. The source programmer can write a macro instruction in his source program that will cause the macro generator to extract and tailor the routine and insert it as an inline routine in the symbolic program.

Model Statements

Library routines consist of model statements that establish the conditions for inserting parameters in the symbolic routine, and define the basic structure of the symbolic program entries produced by the macro generator.

Model statements can be divided into two categories:

1. *Complete (no parameters needed).* The format of a complete model statement is the same as that of a source-program statement. A complete model statement is included in the generated symbolic routine unless a bypass condition exists. (See *BOOL.)*

   All model statements in an inflexible library routine must be complete.

   Figure 95 shows a complete model statement designed to compare FIELDA to FIELDB.

Label	Operation	Operand and Comments
	C	FIELDA,FIELDB

Figure 95. Model Statement for a Complete Instruction

2. *Incomplete.* The substitution codes used by the library programmer determine if parameters are required or optional.

   a. *Parameters required.* A substitution code in the form □01-□99 indicates that a parameter must be supplied. The number that follows the □ indicates the position of the parameter in the macro instruction. The statement, with the proper parameters inserted, appears in the generated symbolic routine unless a bypass condition exists. Figure 96 shows a model statement that requires parameters, and a macro instruction that supplies the required parameters.

Macro-Instruction

Label	Operation	OPERAND
	CHECK	PAR1,PAR2

Model Statement

Label	Operation	Operand and Comments
	C	□01,□02

Generated Symbolic Program Entry

C PAR1,PAR2

Figure 96. Incomplete Instruction with Required Parameters

b. *Parameters optional.* A substitution code in the form □0A-□9I indicates that a parameter is optional. (□01-□99 with A- and B-bits over the units position.) The statement is included in the symbolic routine only if the parameter is supplied by the macro instruction. This kind of model statement can also be bypassed by a BOOL statement.

Figure 97 shows a model statement with a conditional substitution code. The □0C represents the third parameter of the macro instruction that extracts the routine. If the third parameter is supplied, the statement is included in the generated symbolic routine. If it is omitted, the statement is not inserted.

L	Label	Operation	Operand and Comments
6	7–21	22–26	27–55
		B	□0C

Figure 97. Conditional Parameter

A model statement in a flexible library routine can contain any combination of valid codes. The following descriptions state the kinds of codes that can be used in the label, operation, and operand fields of model statements. Figure 98 summarizes the uses of model-statement codes.

CODE	POSITION	FUNCTION
□01-□99	Statement	Substitute parameter (parameter must be present)
□0A-□9I	Statement	Substitute parameter (if parameter is missing, delete statement)
□0J-□9R	Label Field and Operand Field	Assign internal label

Figure 98. Model-Statement Codes

Labels

The two kinds of labels used in model statements are:

1. *External.* These labels are used as operands in the source program. For example, if the model statement outlines an instruction that is an entry point for a branch instruction, the label of the statement must be the I-operand of the branch instruction.

   The label of the source-program macro instruction causes the macro generator to produce an EQU statement, in the form LABEL EQU *+1, as the first statement in the symbolic routine. The library programmer can allow for additional external labels by writing a □ followed by a number (01-99) in the label fields of model statements that require labels.

Macro Instruction

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TESTZ	INVER	START1,START2,ENTRYA

Model Statement

L	Label	Operation	Operand and Comments
6	7–21	22–26	27–58
	□02	SBR	□03

Generated Symbolic Program Entry

START2 SBR ENTRYA

Figure 99. Additional External Labels

   The source programmer must supply the label by writing the corresponding parameter in the macro instruction.

   Figure 99 shows a macro instruction and a model statement that produce an external label.

2. *Internal.* These labels are used as operands in other model statements within the same library routine. To refer symbolically to instructions in flexible library routines, write the code □0J-□9R (01-99 with a B-bit over the units position) in the label field of the instruction, and use the label as the operand in another model statement.

   The macro generator produces an internal label in the form □nn mmm, where nn is the code (0J-9R), and mmm is the number of the macro within the source program. These special symbolic addresses are developed to ensure that duplicate core-storage addresses are not assigned to internal labels.

   A label used within an inflexible library routine must be written according to the rules of Autocoder. It *can* be alphameric, *must* begin with a letter, *must not* contain blanks or special characters, and *must not* exceed six characters.

   Figure 100 shows a macro instruction and model statements that produce an internal label. Assume that UPDAT is the 23rd macro in the source program.

Operation Codes

Any valid Autocoder mnemonic can be used in the operation field of a model statement. In flexible library routines, the library programmer can write a substitution code in the form □01-□99 or □0A-□9I instead of a mnemonic.

A model statement in the library routine for a macro instruction may not be another macro instruction except the INCLD macro. An INCLD model statement must have a $ symbol (11-3-8 punch) in column 6.

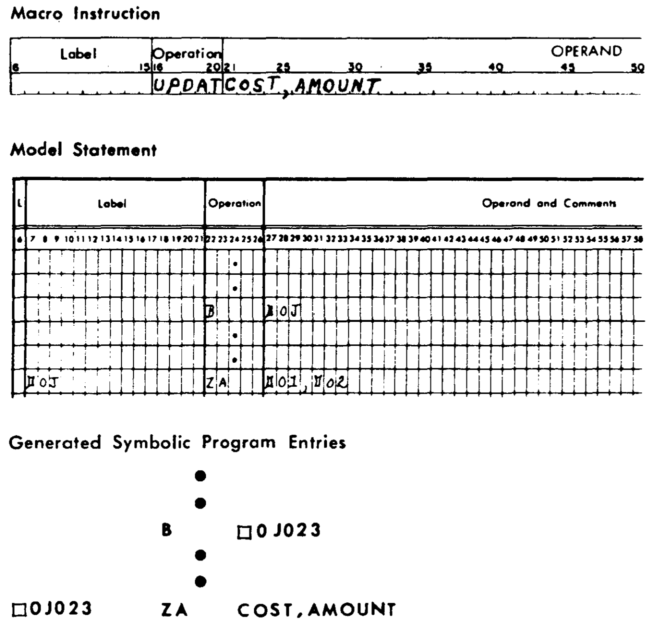


Figure 100. Internal Labels

LTORG and EX statements may be used in library routines. If LTORG or EX is used in a library routine, closed library routines will *not* be included in the program at this point.

Operands

The library programmer can use any valid operand in a model statement. If a symbolic operand is used, it must appear as a label within the same library routine or in a source-program statement.

Any of the substitution codes can be used as model-statement operands in flexible library routines. If the code ◻01-◻99 or ◻0A-◻9I is used, the corresponding parameter must appear as a label in the source program. If the code ◻0J-◻9R is used, it must appear as the label of another model statement within the same flexible library routine.

Literals

Literals are valid in all model statements. In flexible routines, substitution codes (◻01-◻99 or ◻0A-◻9I) can represent a literal or any part of a literal.

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 in the model statement, the generated symbolic instruction will be adjusted to the algebraic sum of the two. For example, if the address-adjustment of one is +7 and the other is −4, the generated instruction will have an address-adjustment factor of +3.

Operands may be indexed in the library routine. However, if a parameter supplied by the macro instruction is also indexed, the parameter will be indexed by the index code in the model statement in the library routine.

Special Requirements for INCLD Library Routines

The inflexible library routines that the library programmer develops for use with the INCLD (or CALL) macro have several requirements that must be considered.

1. Every entry point in the routine should have a label. If a CALL macro is to be used to generate the routine, the first five characters of every entry point label must be the same as the name of the routine. This is required because a CALL uses the first five characters of the entry beginning in column 21 of the CALL statement to generate the routine, and the first six characters of the entry to generate a branch to the routine. This same labeling procedure may be used if the routine is generated by an INCLD. As with the CALL, only the first five characters, beginning in column 21 of the INCLD statement, are used to generate the routine; however, the source programmer must still code a branch to the routine. Note that if this labeling procedure is used for an INCLD routine with more than one entry point, suffixing (*see SFX-Suffix*) cannot prevent the occurrence of multiple-defined labels if the routine is generated two or more times within a program.
2. For routines called by INCLD's, the first instruction at each entry point must store the contents of the B-address register (SBR) in an index location or in the last instruction executed in the library routine. This provides for re-entry in the proper place in the main routine after the INCLD routine has been executed.

   Note: If the object machine does not have the advanced programming feature or the indexing-and-store-address-register feature, the programmer must provide other linkage back to the main routine. An example of such linkage is shown in Figure 101. (For linkage to routines brought out by CALL macros, see *CALL Macro*.)
3. All macro instructions except INCLD are invalid in inflexible library routines. All other statements acceptable to Autocoder, except END, may be used.
4. INCLD statements may appear in either flexible or inflexible library routines. An INCLD model statement should have a $ symbol (11-3-8 punch) in column 6.

Main Program

Label	Operation	OPERAND
	MLC	*-3,SUBRO0
	B	SUBRO1
	INCLD	SUBRO

Library Routine

Label	Operation	Operand and Comments
SUBRO0	DCW	@000@
SUBRO1	MA	@008@,SUBRO1
	MLC	SUBRO0,SUBROX+3
	.	
	.	
SUBROX	B	0

Figure 101. Sample Linkage between the Main Program and an INCLD Routine

Pseudo Macro Instructions

These are instructions that can be used by the library programmer to control the generation of symbolic routines. They are never used by the source programmer, nor do they ever appear in the output listing of an assembled Autocoder program.

They are written within library routines to signal the macro generator that certain conditions exist that affect the generation of the symbolic routines. For example, the presence of a pseudo-macro instruction in a library routine can cause the macro generator to delete one or more model statements when it develops the symbolic routine. Thus, pseudo macros provide the library programmer with a coding flexibility that exceeds the limitations of the substitution and condition codes.

Pseudo-macro instructions may be written anywhere in a library routine. The three pseudo macros implemented by Autocoder are MATH, BOOL, and MEND.

Permanent and Temporary Switches

The MATH and BOOL pseudo macros use internal indicators (switches) to signal the macro generator of existing status conditions. (Model statements do not interrogate switches.)

There are 99 permanent and 99 temporary switches that are used for recording status conditions during processing. Of these, permanent switches 06-50 and all 99 temporary switches are available to the user. Each switch occupies one core-storage position during the macro-generation phase of Autocoder. At the beginning of macro generation, all switches are OFF. During macro generation, if one of these storage positions contains the character 1 (1-bit), the switch is ON. If it contains a 0 (8- and 2-bits), the switch is OFF.

Permanent Switches

Permanent switches retain status conditions throughout the macro-generation phase unless they are changed by a pseudo macro. Address them by using a # symbol followed by the two-digit number of the switch to be set or tested. For example, #06 addresses permanent switch 06, #07 addresses switch 07, and #49 addresses switch 49.

Note: The Autocoder processor uses permanent switches #01, #02, #03, #04, and #05 to store information from the control card. Permanent switches 51-99 are reserved for the Autocoder assembler:

1. The presence of the modify-address, advanced-programming, indexing-and-store-address-register feature, and multiply/divide features in the object machine will set permanent switches #01, #02 and #03, respectively.
2. Permanent switches #04 and #05 are set according to the storage capacity of the object machine as shown here.

Storage Capacity	*#04*	*#05*
4,000	OFF	OFF
8,000	OFF	ON
12,000	ON	OFF
16,000	ON	ON

Temporary Switches

The 99 temporary switches are set at the time the macro generator encounters a macro instruction in the source program. Each of the 99 parameters that can be written in a macro instruction has a corresponding temporary switch that reflects the presence or absence of the parameter in the particular macro instruction being processed. 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 01 is present, temporary-switch 01 is turned on. If parameter 02 is missing from the macro instruction, temporary-switch 02 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 two-digit number of the switch to be set or tested. For example, □01 addresses temporary switch 01; □02 addresses switch 02, and □99 addresses switch 99.

For another example, if a macro with a maximum of nine parameters is encountered, the macro generator sets the first nine temporary switches to indicate the presence or absence of these nine parameters. Temporary switches 10-99, which are off, can be used by the pseudo macros to communicate conditions to the macro generator 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

General Description. 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 two-digit number (ending in zero or five) of the sum box to be referenced. For example, the address of the first sum box is #00; the address of the second sum box is #05; and the address of the twentieth sum box is #95.

Note: Sum box 95 should not be reset, as it is used by the assembler. If the object program is to be in either condensed loader or coreload format, sum box 95 contains the address that branches back to the program loader after loading has been interrupted for execution of a part of the object program. For 1440 systems, note that this branch-to address is the address of the loader for coreload format. The branch-to address for the condensed-loader format is the address of the loader + eight. Column 42 of the CTL card determines which of these two values is placed in sum box 95. If the object program is to be in the self-loading format, sum box 95 contains 0008L.

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 contains the sign of the result if it is negative. 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 102, assume that sum box #05 contains 02345 and sum box #10 contains 0001N (negative 00015).

In a DC or DCW model statement, a blank constant may only define an area up to nine positions (#1 through #9). This requirement must be met for model statements so that the assembler will not confuse a blank constant with a sum box.

Arithmetic Expressions

Arithmetic expressions within the MATH pseudo macro use add (+), subtract (—), multiply (*), and divide (/). Arithmetic operations are executed in the following order: multiplication and division, and addition and subtraction. If parentheses are needed to define the expression the @ symbol represents both the left and right parentheses. For example:

(001+12—5) 20 is written @ 001+12—5 @ *20.

Each term of an arithmetic expression is expanded to five characters before the MATH pseudo macro is placed on the library; any part of the expanded macro exceeding column 75 will not be placed on the library. An arithmetic expression should not begin with a signed number.

Arithmetic operations are executed in the operand field of the MATH pseudo macro from left to right. The quotient resulting from a divide operation is *not* half-adjusted, and the remainder is lost. At the end of a multiplication operation the five low-order positions are used for the result. (The high-order digits are lost.) An overflow is ignored. The five low-order positions of intermediate results are used, but the high-order positions are lost.

The result of the arithmetic expression is produced and inserted with its sign in the designated sum box if a sum box is specified.

Sign Switches

Permanent and temporary switches may 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 (greater than zero), the second represents a zero result, and the third represents a negative (less than zero) result. Consequently, one switch is ON and the other two are OFF after the arithmetic expression has been processed.

It is not necessary to specify all three switches in the pseudo-macro operand. However, if a switch code is omitted, the comma that would have followed the

Macro Instruction

Label	Operation	OPERAND
	REORG	FLD1,,FLD2

Model Statement

Label	Operation	Operand and Comments
	ORG	#05
	ZA	#01+#10,#02

Generated Symbolic Program Entries

```
ORG  02345
ZA   FLD1+0001N,FLD2
```

Figure 102. Sum Boxes

switch code must be included unless it is the last-specified switch. This is the same rule that applies to missing parameters in a source-program macro instruction. The same rule applies to omitted sum boxes. Any switch may be used to represent a sign switch.

The library programmer:

1. Writes 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(es) in which the sign(s) of the result are to be stored.

   Note: Commas must separate the sum-box code, the arithmetic expression, and the individual-sign switch codes. Figure 103 shows the format for the MATH pseudo macro.

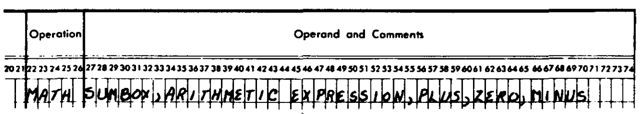


Figure 103. Format for MATH Pseudo Macro

The macro generator:

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

Example. The MATH pseudo macro shown in Figure 104 multiplies parameter 07 by 401 and adds 12 to the result. The answer is stored in SUMBOX#30. If the result is positive, permanent switch 04 is set ON; if the result is zero, permanent switch 06 is set ON; if the result is negative, temporary switch 09 is set ON.

Note: Sum boxes may be used within the arithmetic expression in a MATH pseudo macro.

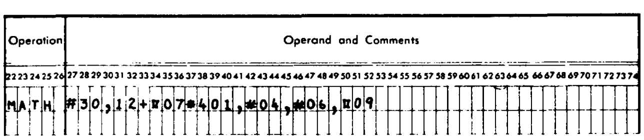


Figure 104. MATH Pseudo Macro

BOOL — For Solving Logical Expressions

General Description. The BOOL pseudo macro can be used:

1. To set a permanent or temporary switch as the result of a logical expression.
2. To cause the macro generator to skip over certain model statements if the logical expression is false. If the logical expression is true, the macro generator goes to the next sequential model statement.

The library programmer:

1. Writes BOOL in the operation field.
2. May write a one-character label, the logical expression, and the switch code in the operand field in the format shown in Figure 105.

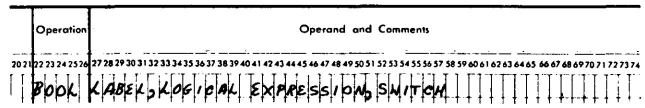


Figure 105. Format for the BOOL Pseudo Macro

Labeling

A special one-character label permits skipping forward over model statements in the library routine as the symbolic routine is being developed. Write this one-character label 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. The macro generator will skip over the intervening model statements only if the logical expression is false.

Omit the label to direct the macro generator not to skip over any model statements, but include the comma that would have followed the label to indicate that it is missing. Use any alphabetic or numeric character as the label, but do not use a special character.

Logical Expression

The library programmer may use any combination of the three logical operations: *and* (*), *or* (+), and *not* (—). Logic operations are executed in the following order: (—), (*), and (+). If parentheses are needed to define the expression, the @ symbol represents both the right and left parentheses. The operators are defined in Figure 106. The combination of these operators and the switches to be tested for ON or OFF status make up the logical expression (Figure 107).

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

Figure 106. Table of Operators

Page and Line	L	Label	Operation	Operand and Comment
01001			BOOL	L,□01*□02,□15
01002			A	FIELDA,FIELDB
01003			B	EOJ
01004	L		C	AREA1,AREA2

Figure 107. Using the BOOL Pseudo Macro

Switches

The programmer may use either a permanent or temporary switch to store the result of a logical expression. If the macro generator determines that the expression is true, the specified switch will be set ON. If it finds that the expression is false, the specified switch will be set OFF.

The macro generator:

1. Examines the status switches to determine whether the conditions specified in the logical expressions are satisfied. If the conditions are met, the expression is true; if they are not, 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 macro generator skips forward to the command or model statement whose label position contains the same label character.

   To determine if a logical expression is true or false:

   a. Call all ON switch conditions true and all OFF switch conditions false.
   b. Let 1 = true and 0 = false.
   c. Calculate the logical value of the expression using the table of operators shown in Figure 106.

   If the logical value of the expression is 0, the expression is false. If the logical value is 1, the expression is true. For example, if switches 01, 02, 03, and 04 are ON, the expression:

   □01* □02 + □03* □04 is true because:

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

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

   1 + 1 = 1

Examples. Figure 107 shows how the BOOL pseudo macro can be used. The BOOL entry states:

1. If temporary switches 01 and 02 are ON, the statement is true. Therefore, set temporary switch 15 ON.
2. However, if either temporary switch 01 or 02 is OFF, the statement is false. Therefore, set temporary switch 15 OFF and skip to statement 004.

The examples shown in Figure 108 state:

1. If both temporary switches 01 and 02 or both temporary switches 03 and 04 are ON, the statement is true. Therefore, set temporary switch 15 ON.
2. However, if either temporary switch 01 or 02 and either temporary switch 03 and 04 are off, the statement is false. Therefore, set temporary switch 15 OFF and skip to the statement whose label is L.

Operation	Operand and Comments
BOOL	L,□01*□02+□03*□04,□15

Figure 108. BOOL Pseudo Macro

Figure 109 is a table showing all conditions that will cause the BOOL statement shown in Figure 108 to be true.

SWITCHES

01	*	02	+	03	*	04		LOGICAL VALUE
ON		ON		OFF		OFF		
1	*	1	+	0	*	0	=	1
OFF		OFF		ON		ON		
0	*	0	+	1	*	1	=	1
ON		ON		ON		ON		
1	*	1	+	1	*	1	=	1
ON		ON		ON		OFF		
1	*	1	+	1	*	0	=	1
OFF		ON		ON		ON		
0	*	1	+	1	*	1	=	1
ON		ON		OFF		ON		
1	*	1	+	0	*	1	=	1
ON		OFF		ON		ON		
1	*	0	+	1	*	1	=	1

CONDITIONS — TRUE

Figure 109. True Conditions

Figure 110 is a table showing all conditions that will cause the BOOL statement shown in Figure 108 to be false.

MEND — End of Routine

General Description. Use this pseudo macro to signal the processor that no more model statements in the library routine are to be processed.

The library programmer:

1. Writes MEND in the operation field.
2. Leaves the operand field blank.

Note: The library programmer may use a BOOL pseudo macro to direct the assembler to skip over a

SWITCHES

01	*	02	+	03	*	04		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

CONDITIONS — FALSE

Figure 110. False Conditions

MEND pseudo macro that appears within the library routine, if conditions indicate that more library statements must be processed.

The macro generator: Stops processing the source-program macro instruction when it encounters a MEND statement.

Example. Figure 111 shows a MEND statement.

Operation	Operand and Comments
MEND	

Figure 111. MEND Pseudo Macro

Librarian Control Operations

The INSER and DELET statements are used during the librarian phase of Autocoder.

INSER — Insert

General Description. An INSER statement identifies the library routine. This identification precedes the library routine in disk storage. The programmer can use this statement to insert whole library routines or part of a library routine.

The library programmer:

1. Writes INSER in the operation field of the standard Autocoder coding sheet.
2. Writes the name of the library routine in the label field.

   The following may not be used as names for library routines: DIOCS, DTF, FILE, GET, MERGE, PUT, and SORT.
3. To insert an entire library routine, leave the operand field blank.

   To insert model statements, write the sequence number of the statement after which the insertion is to be made.

   To substitute model statements, write the sequence numbers, separated by a comma, of the first and last model statements to be deleted.

   Note: The sequence numbers of model statements are given in the ALTER column of the library listing.

The librarian:

1. Inserts the new model statements, or
2. Inserts the new library routine.

Result. The library contains the new or modified library routine.

During the macro-generator phases of Autocoder, the header label is matched with the macro name in a source-program macro instruction. The model statements following the header label in the library are used to assemble the symbolic routine that will be incorporated in the object program.

Examples. Figure 112 is an INSER statement that will cause a library routine named CHECK to be inserted into the disk-storage library.

Label	Operation	OPERAND
CHECK	INSER	

Figure 112. Inserting an Entire Library Routine

Figure 113 is an INSER statement that causes the first model statement that is in the library routine to be deleted, and the model statement shown to be inserted into its place.

Autocoder Statement

Label	Operation	OPERAND
CHECK	INSER	1,1

Model Statement

L	Label	Operation	Operand and Comments
	NOJ	B	PAR4

Figure 113. Substituting One Model Statement for Another

Autocoder Statement

Label (6–15)	Operation (16–20)	OPERAND (21–50)
CHECK	INSER	1,2

Model Statement

L	Label (7–21)	Operation (22–26)	Operand and Comments (27–58)
	NOJ	B	PAR4
		C	PAR1,PAR2
		BE	PAR3

Figure 114. Substituting Multiple Model Statements

Figure 114 is an INSER statement that causes model statements 1 and 2 to be deleted, and the model statements shown to be inserted into their places.

DELET — Delete

General Description. The programmer may use this statement to delete a library routine, or parts of a library routine, from the disk-storage library.

The library programmer:

1. Writes DELET in the operation field of the standard Autocoder coding sheet.
2. Writes the name of the library routine in the label field.
3. To delete an entire library routine, leave the operand field blank.

   To delete one model statement, write the sequence number of the statement in the operand field.

   To delete more than one model statement, write the sequence numbers, separated by a comma, of the first and last statements to be deleted.

   Note: The sequence numbers of model statements are given in the ALTER column of the library listing.

The librarian:

1. Deletes the specified model statements, or
2. Deletes the entire routine, if the operand field is blank.

Result. The library is modified according to the user's specifications.

Examples. Figure 115 is a DELET statement that causes the entire CHECK routine to be removed from the library.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
CHECK	DELET	

Figure 115. Deleting an Entire Library Routine

Figure 116 is a DELET statement that causes the first model statement to be deleted from the CHECK routine.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
CHECK	DELET	1

Figure 116. Deleting a Single Model Statement

Figure 117 is a DELET statement that causes model statements 2, 3, 4, and 5 to be deleted from the CHECK routine.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
CHECK	DELET	2,5

Figure 117. Deleting Multiple Model Statements

Declarative and Assembler-Control Statements

Figure 118 lists all the declarative and assembler-control mnemonic operation codes that are valid for the Disk Autocoder language.

Imperative Statements

Figure 119 is an imperative-statement reference chart that lists all the valid-mnemonic imperative-operation codes. The information given for each mnemonic listed is:

1. The description of the mnemonic.
2. The machine-language operation code.
3. The operand sequence. This entry represents the valid set of operands to be used with the mnemonic. Deviations from the specified operand sequences will be diagnosed.

   The following symbols are used to describe the operand sequence.

Symbols	*Meaning*
RD	Declared field — an actual, symbolic, or asterisk address, or an area-defining literal. Address-adjustment and indexing are permitted.
D	Constant or declared field — an actual, symbolic, or asterisk address, or a literal. Address-adjustment and/or indexing are permitted.

DECLARATIVE OPERATIONS	
Mnemonic	Description
DA	Define Area
DC	Define Constant (No Word Mark)
DCW	Define Constant With Word Mark
DS	Define Symbol
DSA	Define Symbol Address
EQU	Equate

ASSEMBLER CONTROL OPERATIONS			
Mnemonic	Description	Mnemonic	Description
CTL	Control	ULST	Stop Listing
END	End	ORG	Origin
ENT	Enter New Coding Mode	XFR	Transfer
		SFX	Suffix
EX	Execute	JOB	Job
LTORG	Literal Origin	INSER	Insert
LIST	Resume Listing	DELET	Delete
SPCE	Space n Lines		

Figure 118. Declarative and Assembler Control Operations

Symbols	*Meaning*
XC	X-control field — address of a unit, such as %U1 used to address tape-unit 1. Address-adjustment and/or indexing are not permitted.
n	Single numeric character. Address-adjustment and/or indexing are not permitted.
S	Symbolic address. Address-adjustment and/or indexing are not permitted.
d	d-character — used to modify an operation code.
,	Operand separator.
/	Optional operand separator. For example, n/XC/S means that either a single numeric character or an X-control field, or a symbolic address, may be used for the operand.

4. The code that indicates whether deletion of one or both operands is permitted.

Code	*Meaning*
2	Both operands deleted
1 or 2	Either the last or both operands deleted
None	No operands deleted

Autocoder diagnostic phases detect an invalid number of operands. For example, if a BWZ instruction contained one operand and a d-character, the diagnostic message # OPERANDS would be printed.

Note: The programmer should know the effects of his instructions on the status of the A- and B-address registers in order to determine whether deletion of operands is practical in specific cases.

Most single-address instructions (Op code and an A-address) cause the A-address to be inserted in both the A- and B-address registers. However, MOVE, LOAD, and STORE B-ADDRESS REGISTER (Op codes M, L, and H) do not disturb the B-address register, and therefore permit the programmer to use the previous contents of that register as part of the instruction.

All no-address instructions (Op code only) use the previous contents of the A- and B-address registers.

The contents of the B-address register after a branch instruction depend on the type of branch, the success of the branch, and the presence or absence of the indexing feature.

5. The X-control field, if required.
6. The d-character, if required. Figures 120 and 121 list the d-characters for Control Carriage (CC) and Select Stacker (SS) mnemonics.
7. The object systems or devices on which the instruction can be executed.

Mnemonic	Description	Op Code	Operand Sequence	Operand Deletion	X-Control Field	d-Character	Object System or Device
ARITHMETIC OPERATIONS							
A	Add	A	D,RD	1 or 2			all systems
S	Subtract	S	D,RD	1 or 2			all systems
ZA	Zero and Add	?	D,RD	1 or 2			all systems
ZS	Zero and Subtract	!	D,RD	1 or 2			all systems
D	Divide	%	D,RD	None			all systems*
M	Multiply	@	D,RD	None			all systems*
DATA CONTROL OPERATIONS							
MBC	Move and Binary Code	M	D,RD	None		B	1401*, 1460*
MBD	Move and Binary Decode	M	D,RD	None		A	1401*, 1460*
MCE	Move Characters and Edit	E	D,RD	1 or 2			all systems
MCS	Move Characters and Suppress Zeros	Z	D,RD	1 or 2			all systems
MIZ	Move and Insert Zeros	X	D,RD	None			1401*, 1460*, 1440*
MLC	Move Characters to Word Mark	M	D,RD	1 or 2			all systems
MCW	Move Characters to Word Mark	M	D,RD	1 or 2			1401, 1460
MLCWA	Move Characters and Word Marks	L	D,RD	1 or 2			all systems
LCA	to Word Mark in A-field	L	D,RD	1 or 2			1401, 1460
MLNS	Move Numeric portion of Single Character	D	D,RD	1 or 2			all systems
MN		D	D,RD	1 or 2			1401, 1460
MLZS	Move Single Zone	Y	D,RD	1 or 2			all systems
MZ		Y	D,RD	1 or 2			1401, 1460
MRCM	Move Characters to Record Mark	P	D,RD	1 or 2			1401*, 1460, 1440
MCM	or Group Mark-Word Mark	P	D,RD	1 or 2			1401*, 1460
MRCWG	Move Characters and Word Marks to Group Mark-Word Mark in A-field	P	D,RD	1 or 2		>	1440*, 1460 Mod 3*
LOGIC OPERATIONS							
B	Branch Unconditional	B	RD	None			all systems
BAV	Branch on Arithmetic Overflow	B	RD	None		Z	all systems
BBE	Branch if Bit Equal	W	RD,D,d	2		d**	all systems*
BCE	Branch if Character Equal	B	RD,D,d	2		d**	all systems
BCV	Branch on Carriage Overflow (12)	B	RD	None		@	all systems
BC9	Branch on Carriage Channel 9	B	RD	None		9	all systems
BE	Branch on Equal Compare (B = A)	B	RD	None		S	1401*, 1460, 1440
BEF	Branch on End of File or End of Reel	B	RD	None		K	1401*, 1460*, 1440
BER	Branch on Tape Transmission Error	B	RD	None		L	1401*, 1460*, 1440
BH	Branch on High Compare (B > A)	B	RD	None		U	1401*, 1460, 1440
BIN	Branch if any Disk Drive Error Condition	B	RD,d	None		Y**	1401*, 1460*, 1440
BIN	Branch if Access Inoperable	B	RD,d	None		N**	1401*, 1460*, 1440
BIN	Branch if Disk Error	B	RD,d	None		V**	1401*, 1460*, 1440
BIN	Branch if Wrong-Length Record (Disk)	B	RD,d	None		W**	1401*, 1460*, 1440
BIN	Branch if Unequal Address Compare (Disk)	B	RD,d	None		X**	1401*, 1460*, 1440
BIN	Branch if Reader Error I/O Check Stop Switch Off	B	RD,d	None		?**	all systems
BIN	Branch if Punch Error I/O Check Stop Switch Off	B	RD,d	None		!**	all systems
BIN	Branch if Printer Error I/O Check Stop Switch Off	B	RD,d	None		‡**	all systems
BIN	Branch if Check Stop Switch Off	B	RD,d	None		%**	all systems
BIN	Branch if Access Busy	B	RD,d	None		**	all systems
BIN	Inquiry Clear	B	RD,d	None		* **	all systems*
BIN	Inquiry Request	B	RD,d	None		Q**	all systems
BIN	Reader Busy	B	RD,d	None		H**	1401, 1460
BIN	Punch Busy	B	RD,d	None		I**	1401, 1460
BIN	Tape or Input-Output Busy	B	RD,d	None		J**	1401*, 1460*
BL	Branch on Low Compare (B < A)	B	RD	None		T	1401*, 1460, 1440
BLC	Branch on Last Card (Sense Switch A)	B	RD	None		A	all systems

* Special Feature
** d-Character must be placed in operand when coding in Autocoder.
† (See Figure 120)
†† (See Figure 121)

Figure 119. Imperative Operations (Part 1 of 4)

Mnemonic	Description	Op Code	Operand Sequence	Operand Deletion	X-Control Field	d-Character	Object System or Device
			LOGIC OPERATIONS (CONT.)				
BLC2	Branch on Last Card (Reader Unit 2)	B	RD	None		&	1440
BM	Branch on Minus (11-Zone)	V	RD,D	2		K	all systems
BPCB	Branch Printer Carriage Busy	B	RD	None		R	1401*, 1460*, 1440
BPB	Branch Printer Busy	B	RD	None		P	1401*, 1460*, 1440
BSS	Branch on Sense Switch (B-G)	B	RD,d	None		(B-G)**	all systems*
BSS	Branch on Sense Switch A	B	RD,d	None		A**	all systems
BU	Branch on Unequal Compare (B ≠ A)	B	RD	None		/	all systems
BW	Branch on Word Mark	V	RD,D	2		1	all systems
BWZ	Branch on No Zone (No A- or B-Bit)	V	RD,D,d	2		2**	all systems
BWZ	Branch on 12-Zone (AB-bits)	V	RD,D,d	2		B**	all systems
BWZ	Branch on 11-Zone (B-bit, no A-bit)	V	RD,D,d	2		K**	all systems
BWZ	Branch on 0-Zone (A-bit, no B-bit)	V	RD,D,d	2		S**	all systems
BWZ	Branch on either a Word Mark or No Zone	V	RD,D,d	2		3**	all systems
BWZ	Branch on either a Word Mark or 12-Zone	V	RD,D,d	2		C**	all systems
BWZ	Branch on either a Word Mark or 11-Zone	V	RD,D,d	2		L**	all systems
BWZ	Branch on either a Word Mark or 0-Zone	V	RD,D,d	2		T**	all systems
C	Compare	C	D,D	1 or 2			all systems
			MISCELLANEOUS OPERATIONS				
CC	Carriage Control	F	d	None		d**†	all systems
CCB	Carriage Control and Branch	F	RD,d	None		d**†	1401, 1460
CS	Clear Storage	/	RD	1 or 2			all systems
CS	Clear Storage and Branch	/	RD,RD	1 or 2			all systems
CW	Clear Word Mark	⌑	RD,RD	1 or 2			all systems
H	Halt	.	D,D	1 or 2			all systems
MA	Modify Address	#	D,RD	1 or 2			all systems*
NOP	No Operation	N	XC/D,D,d	1 or 2		**	all systems
SAR	Store A-Address Register	Q	RD,D	1 or 2			all systems*
SBR	Store B-Address Register	H	RD,D	1 or 2			all systems*
SS	Select Stacker	K	d	None		d**††	all systems
SSB	Select Stacker and Branch	K	RD,d	None		d**††	1401, 1460
SS	Overlap On	K	d	None		$**	1401*, 1460*
SSB	Overlap On and Branch	K	RD,d	None		$**	1401*, 1460*
SS	Overlap Off	K	d	None		.**	1401*, 1460*
SSB	Overlap Off and Branch	K	RD,d	None		.**	1401*, 1460*
SW	Set Word Mark	,	RD,RD	1 or 2			all systems
TR	Translate	T	D,RD	None			1440*, 1460*
TRW	Translate with Word Marks	T	D,RD	None		>	1440*, 1460*
			MAGNETIC TAPE OPERATIONS				
BSP	Backspace Tape	U	n/XC/S	None	%Un	B	all systems
RT	Read Tape	M	n/XC/S,RD	None	%Un	R	all systems
RTB	Read Tape Binary	M	n/XC/S,RD	None	%Bn	R	all systems
RTW	Read Tape with Word Marks	L	n/XC/S,RD	None	%Un	R	all systems
RWD	Rewind Tape	U	n/XC/S	None	%Un	R	all systems
RWU	Rewind and Unload Tape	U	n/XC/S	None	%Un	U	all systems
SKP	Skip and Blank Tape	U	n/XC/S	None	%Un	E	all systems
WT	Write Tape	M	n/XC/S,RD	None	%Un	W	all systems
WTB	Write Tape Binary	M	n/XC/S,RD	None	%Bn	W	all systems
WTM	Write Tape Mark	U	n/XC/S	None	%Un	M	all systems
WTW	Write Tape with Word Marks	L	n/XC/S,RD	None	%Un	W	all systems

Note. For tape operations in the overlap mode (1401*, 1460*), the operand sequence is XC/S,RD.
The X-control field must begin with an @ symbol instead of a % symbol.

* Special Feature
** d-Character must be placed in operand when coding in Autocoder.
† (See Figure 120)
†† (See Figure 121)

Figure 119. Imperative Operations (Part 2 of 4)

Mnemonic	Description	Op Code	Operand Sequence	Operand Deletion	X-Control Field	d-Character	Object System or Device
I/O DEVICE OPERATIONS							
R	Read a Card	1	RD	None			1402
R	Read a Card	M	n/XC/S,RD	None	%Gn	R	1442
RCB	Read Column Binary (Card Image)	1	RD	None		C	1402*
RCB	Read Column Binary (Card Image)	M	n/XC/S,RD	None	%Gn	R	1442*
P	Punch a Card	4	RD	None			1402
P	Punch a Card and Feed	M	n/XC/S,RD	None	%Gn	G	1442
PCB	Punch Column Binary (Card Image)	4	RD	None		C	1402*
PCB	Punch Column Binary and Feed (Card Image)	M	n/XC/S,RD	None	%Gn	G	1442*
PS	Punch a Card and Stop	M	n/XC/S,RD	None	%Gn	P	1442
W	Write a Line	2	RD	None			1403, 1404
W	Write a Line	M	RD	None	%Y1	W	1443, 1445***
WM	Write Word Marks	2	RD	None		□	1403
WS	Write and Suppress Space	M	RD	None	%Y1	S	1443, 1445***
WR	Write and Read	3	RD	None			1402
RP	Read and Punch	5	RD	None			1402
RF	Read Punch Feed	4	RD	None		R	1402*
WP	Write and Punch	6	RD	None			1402
WRF	Write and Read Punch Feed	6	RD	None		R	1402*
WRP	Write, Read, and Punch	7	RD	None			1402
SRF	Start Read Feed	8	No operands				1402*
SPF	Start Punch Feed	9	No operands				1402*
WCP	Write Console Printer	M	RD	None	%T0	W	1407, 1447
RCP	Read Console Printer	M	RD	None	%T0	R	1407, 1447
WCPW	Write Console Printer with Word Marks	L	RD	None	%T0	W	1407, 1447
RCPW	Read Console Printer with Word Marks	L	RD	None	%T0	R	1407, 1447
PSK	Punch Skip	M	n/XC/S,RD	None	%Gn	C	1442
LU	Load Unit	L	XC/S,RD,d	None		d**	all devices
MU	Move Unit	M	XC/S,RD,d	None		d**	all devices
CU	Control Unit	U	XC/S,d	None		d**	all devices
Note. If MU and LU are used for overlap operations (1401*, 1460*) with magnetic tape, paper tape, or character reader, the X-control field must begin with an @ symbol instead of a % symbol.							
DISK OPERATIONS							
RD	Read Disk Sector(s)	M	RD	None	%F1	R	1405, 1311, 1301
RDCO	Read Disk with Sector Count Overlay	M	RD	None	%F5	R	1311, 1301
RDCOW	Read Disk with Sector Count Overlay with Word Marks	L	RD	None	%F5	R	1311, 1301
RDT	Read Disk Track Sectors with Addresses	M	RD	None	%F6	R	1311, 1301
RDT	Read Disk Full Track	M	RD	None	%F2	R	1405
RDTA	Read Disk Track Record with Address	M	RD	None	%F@	R	1311, 1301
RDTAW	Read Disk Track Record with Address and Word Marks	L	RD	None	%F@	R	1311, 1301
RDTR	Read Disk Track Record	M	RD	None	%F2	R	1311, 1301
RDTRW	Read Disk Track Record with Word Marks	L	RD	None	%F2	R	1311, 1301
RDTW	Read Disk Track Sectors with Addresses and Word Marks	L	RD	None	%F6	R	1311, 1301
RDTW	Read Disk Full Track with Word Marks	L	RD	None	%F2	R	1405
RDW	Read Disk Sector(s) with Word Marks	L	RD	None	%F1	R	1405, 1311, 1301
SD	Seek Disk	M	RD	None	%F0	R	1405, 1311, 1301
SDE	Scan Disk Equal	M	RD	None	%F8	W	1311, 1301*
SDEW	Scan Disk Equal with Word Marks	L	RD	None	%F8	W	1311, 1301*
SDH	Scan Disk High, Equal	M	RD	None	%F9	W	1311, 1301*

*Special Feature.
**d-Character must be placed in operand when coding in Autocoder.
***1445 on 1440/1460 Systems only.
†(See Figure 120)
††(See Figure 121)

Figure 119. Imperative Operations (Part 3 of 4)

Mnemonic	Description	Op Code	Operand Sequence	Operand Deletion	X-Control Field	d-Character	Object System or Device
DISK OPERATIONS (CONT.)							
SDHW	Scan Disk High, Equal with Word Marks	L	RD	None	%F9	W	1311, 1301
SDL	Scan Disk Low, Equal	M	RD	None	%F7	W	1311, 1301
SDLW	Scan Disk Low, Equal with Word Marks	L	RD	None	%F7	W	1311, 1301
WD	Write Disk Sector(s)	M	RD	None	%F1	W	1405, 1311, 1301
WDC	Write Disk Check	M	RD	None	%F3	W	1405, 1311, 1301
WDCO	Write Disk with Sector Count Overlay	M	RD	None	%F5	W	1311, 1301
WDCOW	Write Disk with Sector Count Overlay with Word Marks	L	RD	None	%F5	W	1311, 1301
WDCW	Write Disk Check with Word Marks	L	RD	None	%F3	W	1405, 1311, 1301
WDT	Write Disk Track Sectors with Addresses	M	RD	None	%F6	W	1311, 1301
WDT	Write Disk Full Track	M	RD	None	%F2	W	1405
WDTA	Write Disk Track Record with Address	M	RD	None	%F@	W	1311, 1301
WDTAW	Write Disk Track Record with Address and Word Marks	L	RD	None	%F@	W	1311, 1301
WDTR	Write Disk Track Record	M	RD	None	%F2	W	1311, 1301
WDTRW	Write Disk Track Record with Word Marks	L	RD	None	%F2	W	1311, 1301
WDTW	Write Disk Track Sectors with Addresses and Word Marks	L	RD	None	%F6	W	1311, 1301
WDTW	Write Disk Full Track with Word Marks	L	RD	None	%F2	W	1405
WDW	Write Disk Sector(s) with Word Marks	L	RD	None	%F1	W	1405, 1311, 1301

* Special Feature
** d-Character must be placed in operand when coding in Autocoder.
† (See Figure 120)
†† (See Figure 121)

Figure 119. Imperative Operations (Part 4 of 4)

d	Immediate skip to	d	Skip after print to
1	Channel 1	A	Channel 1
2	Channel 2	B	Channel 2
3	Channel 3	C	Channel 3
4	Channel 4	D	Channel 4
5	Channel 5	E	Channel 5
6	Channel 6	F	Channel 6
7	Channel 7	G	Channel 7
8	Channel 8	H	Channel 8
9	Channel 9	I	Channel 9
0	Channel 10	?	Channel 10
#	Channel 11	.	Channel 11
@	Channel 12	¤	Channel 12
d	**Immediate space**	**d**	**After print-space**
J	1 space	/	1 space
K	2 spaces	S	2 spaces
L	3 spaces	T	3 spaces

Figure 120. Control Carriage d-Characters

Select Stacker (1402)		
d	Feed	Stacker Pocket
1	Read	1
2	Read	8/2
4	Punch	4
8	Punch	8/2

Select Stacker (1442, 1444)				
Unit	(Device)	d	Feed	Stacker Pocket
1	(1442)	2	Read/Punch	2
2	(1442)	0	Read/Punch	2
3	(1444)	#	Punch	2

Figure 121. Select Stacker d-Characters

Index

C24-3258-2

IBM 1401/1440/1460 Printed in U.S.A. C24-3258-2

IBM
International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N. Y. 10601