IBM 1401 DATA PROCESSING SYSTEM BULLETIN

AUTOCODER SPECIFICATIONS

Applied Program No. 1401-AU-037

This edition, J24-1434-2, is a major revision and obsoletes both J24-1434-1 and Technical Newsletter N24-0022.

Autocoder is an advanced symbolic programming system for the IBM 1401 Data Processing System. It supplements and extends, but does not replace, the 1401 Symbolic Programming System, SPS.

A more powerful language, the IBM 1401 *Autocoder* includes macro-instructions, provides a free-form coding sheet for greater programming flexibility, and reduces card handling by using magnetic tape for program manipulation during assembly. The *Autocoder* processor can assemble programs designed to operate on IBM 1401 systems with a maximum storage capacity of 16,000 positions.

With *Autocoder,* the user can provide library routines for operations that are common to many source programs. These routines are extracted from the library and tailored automatically by the processor to satisfy particular requirements outlined in the source program by the programmer.

Machine Requirements

The *Autocoder* processor can assemble programs for all IBM 1401 systems. However, the machine used to assemble a program written in *Autocoder* language must have at least:

4,000 positions of core storage
FOUR IBM 729 II, 729 IV or 7330 Magnetic Tape Units
(NOTE: A fifth magnetic tape unit can be used for delayed multiple program output.)
IBM 1403 Printer, Model 2
IBM 1402 Card Read-Punch
Advanced Programming Features
High-Low-Equal Compare Feature
Sense Switches (Not required for original assembly from a source program card deck, but necessary for all other *Autocoder* operations.)

This bulletin contains the language specifications for the IBM 1401 *Autocoder.* The system card deck containing the processor itself, a listing of the processor program, and operating instructions for program assembly are available from the IBM Program Library.

Address comments regarding this publication to IBM Product Publications, Endicott New York

Programming with Autocoder

The IBM 1401 *Autocoder* is divided into two major categories – the symbolic language used by the programmer, and the processor program that translates this symbolic language into actual machine language and assembles the object program automatically.

Before the programmer begins to code his program in symbolic language, he draws a block diagram of the procedure the program must take to accomplish a desired end result. From this block diagram he must determine what constants and work areas are needed and define them. Constants are fixed data, such as a standard FICA limit of $4800 for tax calculation, and work areas are places within core storage where data can be manipulated, such as an input and output area, accumulator fields, etc. Then he writes the instructions for the program. The IBM 1401 *Autocoder* permits the programmer to control the processor program by using special commands.

These programming procedures can be divided into four major categories:

1. Declarative operations
2. Imperative operations
3. Macro operations
4. Control operations

The particular information needed by the processor to perform these operations is written by the programmer on a special coding sheet.

Coding Sheet

The 1401/1410 *Autocoder* coding sheet (Figure 1) is free-form, (the operand portion of each line is not subdivided into fields) thus allowing the programmer greater coding flexibility. The SPS coding sheet is fixed-form (the operand portion of each line is divided into specific fields).

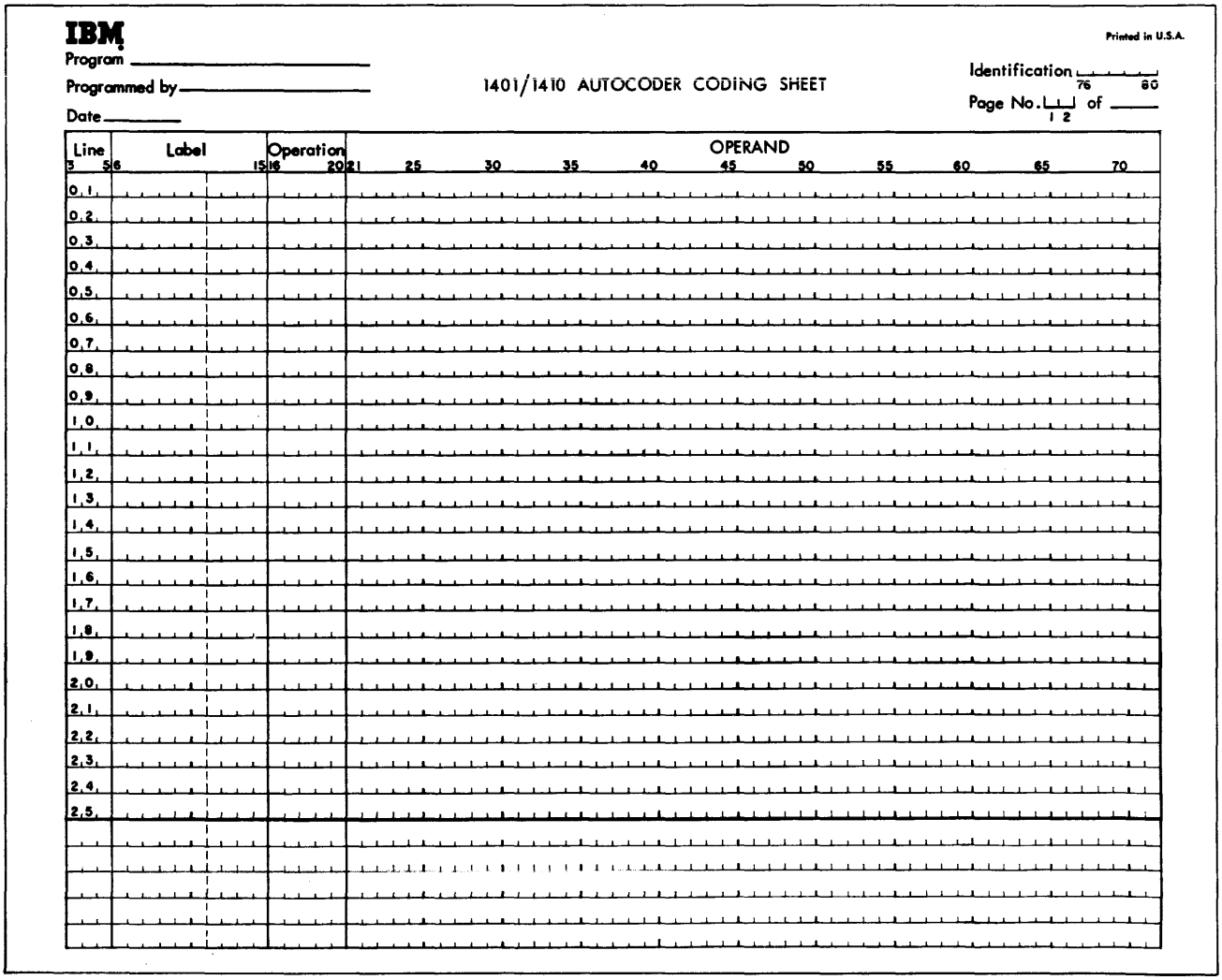


Figure 1. IBM 1401/1410 Autocoder Coding Sheet

DECLARATIVE OPERATIONS

Mnemonic Op Code	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

IMPERATIVE OPERATIONS

Type	Mnemonic Op Code	Description	Machine Op Code	Language d-char.
Arithmetic	A	Add	A	
	D	Divide	%	
	M	Multiply	@	
	S	Subtract	S	
	ZA	Zero and Add	?	
	ZS	Zero and Subtract	!	
Data Control	MBC	Move and Binary Code	M	B
	MBD	Move and Binary Decode	M	A
	MCE	Move Characters and Edit	E	
	MCS	Move Characters and Suppress Zeros	Z	
	MIZ	Move and Insert Zeros	X	
	MLC, MCW	Move Characters to Word Mark	M	
	MLCWA, LCA	Move Characters and Word Marks to Word Mark in A-Field	L	
	MLNS, MN	Move Numerical Portion of Single Character	D	
	MLZS, MZ	Move Single Zone	Y	
	MRCM, MCM	Move Characters to Record Mark or Group Mark-Word Mark	P	
Logic	B	Branch Unconditional	B	
	BAV	Branch on Arithmetic Overflow	B	Z
	†BBE	Branch if Bit Equal	W	d
	BC9	Branch on Carriage Channel 9	B	9
	BCV	Branch on Carriage Overflow (12)	B	@
	BE	Branch on Equal Compare (B = A)	B	S
	BEF	Branch on End of File or End of Reel	B	K
	BER	Branch on Tape Transmission Error	B	L
	BH	Branch on High Compare (B > A)	B	U
	†BIN	Branch on Indicator	B	d
	BL	Branch on Low Compare (B < A)	B	T
	BLC	Branch on Last Card (Sense Switch A)	B	A
	BM	Branch on Minus (11-zone)	V	K
	BPCB	Branch Printer Carriage Busy	B	R
	BPB	Branch Printer Busy	B	P
	BU	Branch on Unequal Compare (B ≠ A)	B	/
	BW	Branch on Word Mark	V	1
	†BWZ	Branch on Word Mark or Zone	V	d
	†BCE	Branch if Character Equal	B	d
	†BSS	Branch if Sense Switch On	B	A-G
	C	Compare	C	
I/O Commands	BSP	Backspace Tape	U	B
	†CU	Control Unit	U	d
	DCR	Disengage Character Reader	U	D
	ECR	Engage Character Reader	U	E
	†LU	Load Unit	L	d
	†MU	Move Unit	M	d
	P	Punch	4	
	PCB	Punch Column Binary	4	C
	R	Read	1	
	RCB	Read Column Binary	1	C
	RD	Read Disk Single Record	M	R
	RDT	Read Disk Full Track	M	R
	RDW	Read Disk Single Record With Word Marks	L	R
	RDTW	Read Disk Full Track With Word Marks	L	R
	RF	Read Punch Feed	4	R
	RP	Read and Punch	5	
	RT	Read Tape	M	R
	RTB	Read Tape Binary	M	R
	RTW	Read Tape With Word Marks	L	R
	RWD	Rewind Tape	U	R
	RWU	Rewind and Unload Tape	U	U
	SD	Seek Disk	M	R
	SKP	Skip and Blank Tape	U	E
	SPF	Start Punch Feed	9	
	SRF	Start Read Feed	8	
	W	Write	2	
	WD	Write Disk Single Record	M	W
	WDC	Write Disk Check	M	W
	WDCW	Write Disk Check With Word Marks	L	W
	WDT	Write Disk Full Track	M	W
	WDTW	Write Disk Full Track With Word Marks	L	W
	WDW	Write Disk Single Record With Word Marks	L	W
	WM	Write Word Marks	2	□
	WP	Write and Punch	6	
	WR	Write and Read	3	
	WRF	Write and Read Punch Feed	5	R
	WRP	Write, Read and Punch	7	
	WT	Write Tape	M	W
	WTB	Write Tape Binary	M	W
	WTM	Write Tape Mark	U	M
	WTW	Write Tape With Word Marks	L	W
Miscellaneous	†CC	Carriage Control	F	d
	†CCB	Carriage Control and Branch	F	d
	CS	Clear Storage	/	
	CW	Clear Word Mark	□	
	H	Halt	•	
	MA	Modify Address	#	
	NOP	No Operation	N	
	SAR	Store A-Address Register	Q	
	SBR	Store B-Address Register	H	
	†SS	Select Stacker	K	1, 2, 4, 8
	†SSB	Select Stacker and Branch	K	1, 2, 4, 8
	SW	Set Word Mark	,	

CONTROL OPERATIONS

Mnemonic	Description	Mnemonic	Description
CTL	Control	XFR	Transfer
END	End	SFX	Suffix
ENT	Enter New Coding Mode	JOB	Job
		INSER	Insert
EX	Execute	ALTER	Alter
LTORG	Literal Origin	DELET	Delete
ORG	Origin		

†d-Character must be placed in operand when coding in *Autocoder*.

Figure 2. IBM 1401 Autocoder Mnemonic Operation Codes

All *Autocoder* entries are entered on the *Autocoder* coding sheet. Column numbers on the coding sheet indicate the punching format for all input cards in the source deck. Each line of the coding sheet is punched into a separate card. (If the source program is entered by magnetic tape, the contents of the cards prepared from the coding sheet must be written in one-card-per-tape-record format.) The function of each portion of the coding sheet is explained in the following paragraphs.

Page Number (Columns 1 and 2)

This two-character entry provides sequencing for coding sheets. Any alphamerical characters may be used. Standard collating sequence for the IBM 1401 should be followed when sequencing pages.

Line Number (Columns 3-5)

A three-character line number sequences entries on each coding sheet. The first 25 lines are prenumbered 01-25. The third position can be left blank (blank is the lowest character in the collating sequence). The five unnumbered lines at the bottom of each sheet can be used to continue line numbering or to make insertions between entries elsewhere on the sheet. The units position of the line number is used to indicate the sequence of inserts. Any alphamerical character can be used, but standard collating sequence should be used. For example, if an insert is to be made between lines 02 and 03, it could be numbered 021. Line numbers do not necessarily have to be consecutive, but the deck should be in collating sequence, for sorting purposes.

The programmer should note that insertions can affect address adjustment. An insertion might make it necessary to change the adjustment factor in the operand of one or more entries.

Label (Columns 6-15)

Autocoder permits symbolic labels in the label field. A symbolic label can have as many as six alphamerical characters, but the first character must be alphabetic. The label is always written starting in column 6.

Operation (Columns 16-20)

Mnemonic operation codes are written in the operation field starting in column 16. Figure 2 is a chart showing 1401 *Autocoder* mnemonics.

Operand (Columns 21-72)

The operand field in an imperative instruction contains the actual or symbolic addresses of the data to be acted upon by the command in the operation field, literals, or address constants. Address adjustment and indexing can be used in conjunction with actual or symbolic addresses.

Unlike the SPS coding sheet, which specifies particular fields for the A-operand, B-operand, address adjustment, indexing and the d-character, the *Autocoder* coding sheet has a free-form operand field. The A-operand, the B-operand, and the d-character must be separated by commas. If address adjustment or indexing or both are to be performed, these notations must immediately follow the address being modified. Figures 3, 4, and 5 show typical *Autocoder* entries.

Figure 3 shows an imperative instruction that causes the contents of the field whose low-order core-storage location is 3101 to be added algebraically to the contents of the field whose low-order location is 140. This entry will be assembled as a machine language instruction:

A A01 140

Note that high-order zeros can be eliminated when coding actual addresses for *Autocoder*.

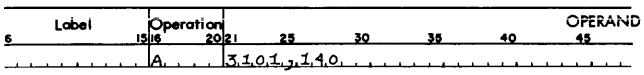


Figure 3. Autocoder Instruction With Actual Addresses

Figure 4 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. An indexed address is always followed by a plus sign (+), an X to indicate indexing, and a 1, 2, or 3 to specify which index location is to be used. TOTAL is the label for location 3 1 0 1 and ACCUM is the label for location 1 4 0. The assembled machine-language instruction for this entry is: M A01 1M0. The M in the tens position of the B-address is a 4-punch with an 11-overpunch. The 11-overpunch is the B-bit tag for index location 2.

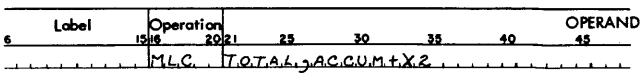


Figure 4. Autocoder Instruction with Symbolic Addresses and Indexing

Figure 5 shows an imperative instruction with address adjustment and indexing on a symbolic address. The processor will subtract 12 from the address which 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 3 1 0 1 and ACCUM is the label for location 1 4 0). The Y in the tens position of the A-address is an 8-punch with a zero overpunch. The zero punch is a tag for index location 1. NOTE: The address adjustment factor cannot exceed ± 9 9 9.

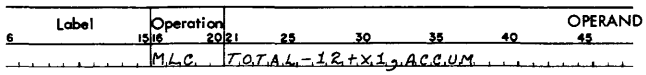


Figure 5. Autocoder Instruction with Address Adjustment and Indexing

Figure 6 is an imperative instruction with two symbolic operands and a d-character. Although many of the augmented operation codes available for use with *Autocoder* eliminate the need to write the d-character in a symbolic instruction, sometimes the d-character must be specified by the programmer. If an instruction requires such a specified d-character, it is written following the A-and B-operands, and is separated from the remainder of the instruction by a comma. The assembled machine-language instruction is: B 392 498 2. It tests a location labeled SWITCH (498) and branches to ENTRYA (392) for the next instruction if SWITCH contains a 2.

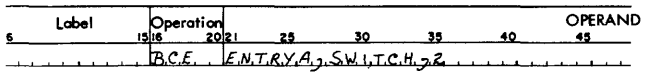


Figure 6. Autocoder Instruction with a d-character

NOTE: Several types of addresses may be placed in the operand. They are discussed in the *Address Types* section.

Comments

A remark can be included anywhere in the operand field of an *Autocoder* statement, if at least two non-significant spaces separate it from the operands.

Entire lines of information can be included anywhere in the program except within a complete DA entry by using a comments card. In such a card, containing comments only, the programmer must put an asterisk in column 6. Columns 7-72 can then be used for the comment itself. Comments inserted in this way appear in the symbolic listing but produce no entry in the object program.

Identification (Columns 76-80)

This entry enables the user to identify a program or program section. When identification is desired, the contents of this line are punched into every card of the source deck. The areas labeled *Program, Programmed By*, and *Date* are for the convenience of the user, but are never punched.

Columns 73-75 are reserved for the processor.

Address Types

Six kinds of address types are valid in the operand field of an *Autocoder* statement: blank, actual, symbolic, asterisk, literals, and address constants.

Blank

A blank operand field is valid:

1. In an instruction that does not require an operand.
2. In instructions where useful A- or B-addresses are supplied by the chaining method.

NOTE: If an instruction is to have addresses stored by other instructions, the operand or operands affected must not be left blank.

Actual

The numerical equivalent of the three-character actual core-storage address is valid in the operand field. High-order zeros in actual addresses can be omitted as shown in Figure 3. Thus, an actual address can consist of from one to five digits.

Symbolic

A symbolic address can consist of as few as one or as many as six alphamerical characters. Special characters are not permitted. Blanks may not be written *within* a symbolic address. Figure 4 shows how symbolic addresses are used.

Asterisk (*)

If an * appears as an operand in the source program, the processor will replace it in the object program with the actual core-storage address of the last character of the instruction in which it appears. For example, the instruction shown in Figure 7 is assigned core-storage locations 340-343. The assembled instruction is • 3 4 0.

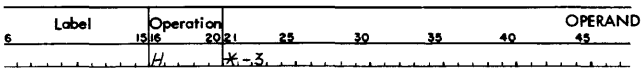


Figure 7. Asterisk Operand in an Autocoder Instruction

Asterisk operands can have address adjustment and indexing.

Literals

The IBM 1401 *Autocoder* permits the user to put in the operand field of a source program statement the actual data to be operated on by an instruction. This data is called a *literal*. The processor allocates storage for

literals and inserts their addresses in the operand or operands of the instructions in which they appear. The processor produces a DCW card that puts a word mark in the high-order position of a literal when it is stored at program load time. (In SPS, literals were not permitted. The actual data to be operated on had to be stored by DCW or DC statements.) Literals are permitted only in the operand field of an *Autocoder* statement and can be numerical or alphamerical. A literal can be any length, provided the operand of the statement that contains the literal does not exceed 52 columns (a statement must be contained in one line of the coding sheet and must not extend beyond column 72). Literals cannot have address adjustment or indexing.

NUMERICAL LITERALS

Numerical literals are written according to the following specifications:

1. A plus or minus sign must precede a numerical literal. The processor puts the sign over the units position of the number when it is assigned a storage location. NOTE: To store an unsigned number, use an alphamerical literal.
2. A numerical literal of from one to five digits (no blanks) and a sign is assigned a storage location only once per program or *program section* no matter how many times it appears in the source program. NOTE: A program section is defined as the source program entries that precede a Literal Origin, End, or Execute statement. In some programs several program sections are needed because the entire object program exceeds the total available storage capacity of the object machine. In these cases, individual program sections are loaded into storage from cards, tapes, or random access storage and are executed as they are needed. Program sections are sometimes called *overlays*.
3. A numerical literal that exceeds five characters and a sign is assigned a storage location each time it is encountered in the source program. To save storage space, use a DCW statement if a long numerical literal is used more than once in the source program.

Figure 8 shows how a numerical literal can be used in an imperative instruction. Assume the literal (+ 10) is assigned storage locations of 584 and 585, and INDEX is assigned 682. The symbolic instruction will cause the processor to produce a machine language instruction (A 585 682) that causes + 10 to be added to the contents of INDEX.

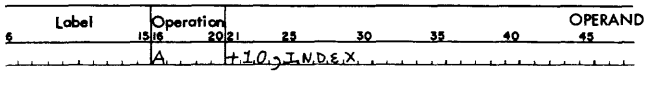


Figure 8. Numerical Literal

ALPHAMERICAL LITERALS

Alphamerical literals are written according to the following specifications:

1. An alphamerical literal must be preceded and followed by the @ symbol. The literal, itself, can contain blanks, alphabetic, numerical, and special characters (including the @ symbol). However, a comment on the same line as an alphamerical literal must not contain the @ symbol.
2. An alphamerical literal of from one to four characters with preceding and following @ symbols is assigned a storage location only once per program or program section no matter how many times it is used in the source program.
3. Longer alphamerical literals are assigned a storage location each time they are encountered in the source program. To save storage space in these cases, use a DCW statement.

NOTE: Only one alphamerical literal may be written on one line of the coding sheet.

Figure 9 shows how an alphamerical literal can be used in an imperative instruction. Assume that the literal JANUARY 28, 1961 is assigned a storage location of 906 and DATE is assigned 230. The machine language instruction (M 906 230) causes the literal JANUARY 28, 1961 to be moved to DATE.

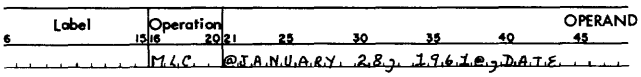


Figure 9. Alphamerical Literal

AREA-DEFINING LITERAL

With *Autocoder*, it is possible to reserve storage for a working area by using an area-defining literal.

1. An area of 52 positions or less may be defined in any instruction which has, as an operand, the symbol which references it.
2. A # symbol (8-3 punch) must precede the number that specifies how many core-storage positions are needed for the work area. (Note the # symbol is represented in the Fortran character set as an = symbol.)

Figure 10 shows an imperative instruction with an area-defining literal. This entry causes the processor to allocate 6 storage locations for WKAREA. Six blanks will be loaded in storage at object program load time by a DCW automatically produced by the processor. Assuming that AMOUNT is in storage location 796 and WKAREA is in 596, the assembled machine-language instruction that moves AMOUNT to WKAREA is M 796 596.

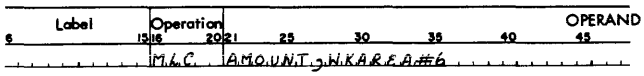


Figure 10. Area-defining Literal

Address Constants

The actual 3-character machine address which is assigned to a label by the processor can be defined as an *address constant*. In SPS, a DSA statement is needed to define an address constant. However, *Autocoder* permits address constants to be coded symbolically in the instructions that require them:

1. The symbol for an address constant can contain as many as six characters.
2. A plus or minus sign must precede the symbol. If a plus sign is used, the address constant is the actual address which was assigned to the label by the processor. If a minus sign is used, the address constant is the 16,000's complement of the actual address.
3. The label being defined must appear elsewhere in the symbolic program.
4. The address constant is assigned a core-storage address, as are all constants, and a DCW card is created automatically by the processor.

Figure 11 shows how an address constant can be used. Assume that CASH is used as a label elsewhere in the program and has been assigned a machine address of 600. The address constant (600) has been assigned storage location 797. The first character in the second instruction is in core storage at address 401. Thus, the address of INST + 3 is 404.

The assembled machine language instruction for the first symbolic instruction in Figure 11 is: M 797 404.

WORK is storage location 729. The assembled machine-language instruction for the second symbolic program entry is M 000 729. When the first instruction is executed in the object program, 600 is moved to 404 and the second instruction becomes M 600 729. When the second instruction is executed, the contents of CASH are moved to WORK.

Thus, the programmer can write an instruction that will move a machine address into the operand of another instruction at program execution time, even though he does not know what that address is.

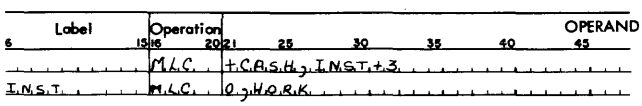


Figure 11. Address Constants

NOTE: Character adjusted and/or indexed address constants can be written symbolically. The address of the label is adjusted or indexed in these cases.

Index Locations

The IBM 1401 has three index locations: + X1, + X2, or + X3 following an operand, specify index locations 1, 2, or 3, respectively. When the processor encounters an indexed operand, a tag is automatically inserted in the tens position of the assembled machine-language address as shown in Figure 5.

Declarative Operations

The IBM 1401 *Autocoder* provides six different declarative operations for reserving work areas and storing constants:

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: A DCW statement is used to enter a numerical, alphamerical, or address constant with a word mark into a core-storage area.

The programmer:

1. Writes the operation code (DCW) in the operation field.
2. May write an actual or symbolic label in the label field. The programmer may refer to the constant later by writing this label in the operand portion of subsequent instructions.
3. Writes the constant in the operand field beginning in column 21.

NUMERICAL CONSTANTS

1. A numerical constant can be preceded by a plus or minus sign. A plus sign causes AB-bits to be placed over the units position of the constant; a minus sign causes a B-bit to be put there. If a numerical constant is unsigned in the DCW statement, it will be stored as an unsigned field.
2. The first blank column appearing in the operand field terminates a numerical constant.
3. The maximum size of a numerical constant is 51 digits and a sign, or 52 digits with no sign.

Example: Figure 12 shows the number, + 10, defined as a numerical constant. The address of the constant will be inserted in the object instruction wherever TEN appears in the operand field of another symbolic instruction.

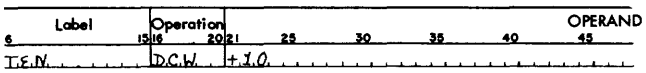


Figure 12. Numerical Constant Defined by a DCW Statement

ALPHAMERICAL CONSTANTS

1. An alphamerical constant must be preceded and followed by the @ symbol. Blanks and the @ symbol can appear within an alphamerical constant, but the @ symbol cannot appear in a comment on the same line as an alphamerical constant.
2. The alphamerical constant, itself, can be as large as 50 characters.

Example: Figure 13 shows the alphamerical constant, JANUARY 28, 1961, defined in a DCW statement. The address of the constant will be inserted in the object program instruction wherever DATE appears in the operand field of another symbolic program entry.

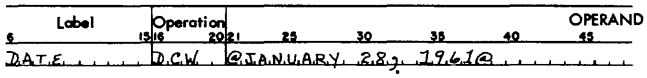


Figure 13. Alphamerical Constant Defined by a DCW Statement

BLANK CONSTANTS

A # symbol precedes a number indicating how many blank storage positions are to be defined. This permits the programmer to reserve a field of blanks with a word mark in the high-order position of the field. The maximum size of this field is 52 blanks.

Example: Figure 14 shows an 11-character blank field defined by a DCW statement. The address of this blank field will be inserted in an object program instruction whenever the symbol BLANK appears as the operand of another symbolic program entry.

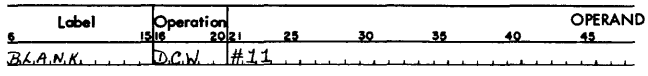


Figure 14. Blank Constant Defined by a DCW Statement

ADDRESS CONSTANTS

An address constant can be preceded by a plus or minus sign. If a plus sign or no sign is used, the constant is the actual machine language address of the field whose associated label is included in the operand. If a minus sign is used, the constant is the 16,000 complement of the actual machine address of that field. NOTE: Address constants may be address-adjusted and indexed.

Example: Figure 15 shows an address constant (the address of MANNO) defined by a DCW statement. The address of the address constant (MANNO) will be inserted in an object program instruction whenever SERIAL appears as the operand of another symbolic program entry.

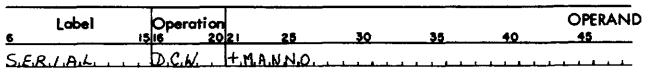


Figure 15. Address Constant Defined by a DCW Statement

The processor:

1. Allocates a field in core storage that will be used to store the actual constant. If the DCW statement has a symbolic address in the label field, the processor assigns an address equal to the low-order position of this field.
2. Inserts the assigned address wherever the symbol in the label field appears in the operand of another symbolic program entry.

Result: A constant with a high-order word mark is loaded with the object program each time the job is run.

DC — Define Constant (No Word Mark)

This statement has the same characteristics as the DCW statement. The only difference is that the processor does not cause a word mark to be set at the high-order position of the constant when the constant is loaded with the object program.

DS — Define Symbol

General Description: A DS statement bypasses and labels an area of core storage. It differs from a DCW or DC statement in that no information (constant) is loaded into this area at program load time.

The programmer:

1. Writes the operation code (DS) in the operation field.
2. May write a symbolic address in the label field. Actual addresses cannot be used in the label field and indexing is not permitted.
3. Writes a number in the operand field to indicate how many storage positions are to be bypassed.

The processor:

1. Assigns an actual address to the low-order position of the reserved area.
2. Inserts this address in the instruction wherever the symbol in the label field appears in the operand field of another symbolic program entry.

Example: Figure 16 shows how a 10-position core-storage area can be bypassed. The programmer can refer to the label by putting ACCUM in the operand field of another symbolic program entry.

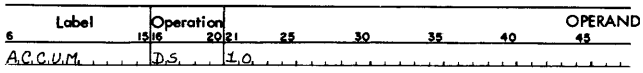


Figure 16. DS Statement

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 symbolic program is desired. (The address constants previously discussed were created from symbolic addresses.)

The programmer:

1. Writes the mnemonic operation code (DSA) in the operation field.
2. May write in the label field, the symbol which will be used to make reference to the address constant.
3. Writes the actual address to be defined in the operand field. This address may be address-adjusted and indexed.

The processor:

1. Produces a constant containing the three-character machine address of the storage address written in the operand field.
2. Assigns this address constant an address in core storage and labels it using the symbol 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: To create and store an address constant for an actual address, the entry shown in Figure 17 is made.

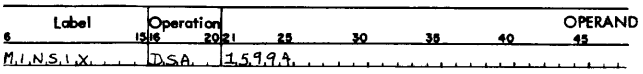


Figure 17. Defining the Address Constant of an Actual Address

Assume that the address assigned to the label (MINSIX) is 892. Storage locations 890, 891, and 892 will contain I 9 D (the three-character machine address of 15994). If index location 1 has been assigned the label INDEX 1, the instruction shown in Figure 18 will cause I 9 D to be moved to index location 1 (storage locations 087-089). The assembled machine language instruction for the statement shown in Figure 18 is M 892 089.

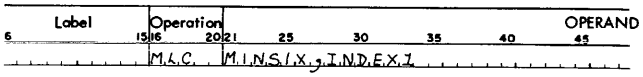


Figure 18. Moving the Address Constant to an Index Location

NOTE: This example shows how the 16,000's complement of an amount to be subtracted from an actual address can be stored in an index location to decrease an indexed address. In this case the amount is 6, which has a 16k complement equal to 15994.

DA — Define Area

General Description: DA statements reserve and define portions of core storage, such as input, output, or work areas. They can also define more than one area, if all these areas are identical in format. A DA statement differs from a DCW statement in that a DA statement can, in addition to defining the large area, also define several fields within it. The DA statement furnishes the processor with the lengths, names, and relative positions of fields within the defined area.

The programmer:

1. Constructs a header line for the DA entry as follows:

   a. Writes the operation code (DA) in the operation field.

   b. May write an actual or symbolic address in the label field. This address represents the high-order position of the entire area defined by the DA statement.

   c. Indicates in the operand field the required size of the area in the form B X L. B is the number of identical areas to be defined, and L is the length of each area. For example, if four identical areas, each 100 characters long, are to be defined, the first entry in the operand field is 4 X 100 as shown in Figure 19. If only one area is to be defined, the first entry is 1 X 100.

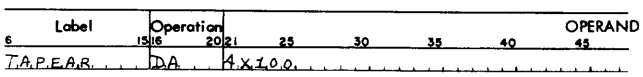


Figure 19. Four Areas Defined

Indexing: To index a DA statement place a comma and the number of the index location (X1, X2, or X3) after the B X L indication. All fields and subfields defined in the DA entry, including labels, will be indexed by the specified index location when they appear in instructions, *unless the instruction referring to the field is itself indexed.* For example, if INAREA is defined by the statement shown in Figure 20, ACCUM is indexed by index location 1. If the entry shown in Figure 21 appears as an instruction elsewhere in the program, ACCUM (for this instruction only) will be indexed by the contents of index location 2. Because the instruction in Figure 21 has indexing, this indexing overrides the indexing prescribed by the DA statement.

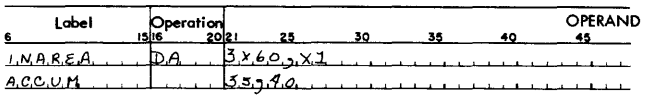


Figure 20. Indexing a DA Entry

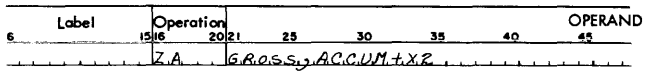


Figure 21. Overriding Indexing in a DA Statement

NOTE: The programmer can negate the effect of indexing on a field or subfield by putting an X0 in the operand field of each instruction in which indexing is not wanted.

Record Marks: Can be inserted to separate records in the defined area. The processor will cause a ‡ to be placed in storage immediately following each identically defined area if a ‡ follows the B X L entry in the operand field. B X L does not include an allowance for the record mark. For example, 2 X 100 will cause 200 positions to be reserved for the defined area, but 2 X 100, ‡ will cause 202 positions to be reserved.

Group Mark with Word Mark: The user can cause the processor to put a group mark with a word mark one position to the right of the entire defined area by writing a G, preceded by a comma, in the operand field.

NOTE: The programmer may write a comma followed by a C if the defined area is to be cleared before word marks, etc., are set at program load time. The ‡, index code, G, and C entries can appear in any order in the operand field of a DA header statement provided that they follow the B X L entry.

2. Constructs the balance of the DA statement which defines fields and subfields for each area as follows:
   a. Leaves the operation field blank.
   b. May write a symbolic label in the label field.
   c. Specifies the relative location of a defined field within the area by putting two numbers in the operand fields. The first location of the defined area is considered location 1. The high-order and low-order positions of the field are written beginning in column 21. These two numbers must be separated by a comma.
   d. A subfield is a field within a defined field and is defined by putting the number representing the low-order position in the operand field.

NOTE: The processor causes word marks to be set in the high-order position of each defined field, but does not so identify subfields. If a word mark is desired in a one-position field, the relative position number must be written twice with the two numbers separated by a comma.

Fields defined in a DA statement can be listed in any order, and all positions within the defined area do not have to be included in the defined fields.

The processor:

1. Allocates an area in core storage equal to B X L plus positions for record marks and a group mark if they are specified in the heading line of the DA entry, and assigns actual addresses to the defined fields and subfields.
2. Inserts the assigned address of the high-order position of the entire defined area wherever the contents of the heading line label field appear as the operand of another symbolic program entry.
3. Inserts the assigned addresses of the low-order positions of fields and subfields in the place of symbols corresponding to the labels of the field-defining entries.

Result: At object program load time:

1. A word mark is set in the high-order position of the entire defined area. If more than one area is defined (for example, 3 X 100), the high-order position of each area is identified by a word mark.
2. Word marks are set for field definition as noted previously.
3. A group mark and record marks are loaded as specified in the heading line.

Example: In this example, data is to be read from magnetic tape into an area of storage where it is to be processed. It is a payroll operation, and each record refers to a different employee. The records are written on tape in blocks of three. Each record is eighty characters long and has the following format:

Positions 4-8	Man Number
Positions 11-26	Employee Name
Positions 32-37	Date
Positions 45-64	Gross Wages
Positions 66-71	Withholding Tax
Positions 74-79	FICA Deduction

Remaining positions contain data not used in this 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 to be placed in storage immediately following the third area.

The DA statement in Figure 22 defines three adjacent identical areas into which each block of three records will be read. It also defines the fields and subfields that are to receive the data listed. The notation 3 X 80 in the header line indicates that three consecutive areas of eighty locations each are to be reserved. The entire 240-location area can be referred to by its high-order label, RDAREA+X0. The G in the header line will cause a group mark with a word mark to be placed in the 241st position. The reference to index location 2 in the header line indicates that the labels RDAREA, NAME, MANNO, DATE, GROSS, FICA, and MONTH, when referred to in symbolic instructions, will be indexed by index location 2.

The user can now, in his symbolic program, give an instruction to read data from tape into a storage area labeled RDAREA+X0. This causes a block of three data records to be placed in the 240 reserved core locations. As a result, the significant data is read into the appropriately labeled fields. This data can now be referred to via the labels DATE, MANNO, FICA, etc., and the user need not concern himself with actual machine addresses. In this example, the user begins by setting index locations 2 to zero. He then processes the significant data in the first record, increments index location 2 by eighty, and branches back to the first instruction of the particular routine. Because all labels defined by this DA statement are incremented by the contents of index location 2, the program will now be processing the second record read into storage. When this routine is performed three times, the user has processed three input records and is ready to read three more records into storage. This has all been performed without any reference to actual machine addresses.

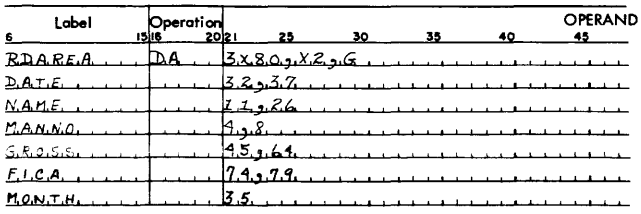


Figure 22. DA Entry

NOTES:

1. An area can be reserved for a record with variable fields by defining all possible fields as subfields. In this case no word marks will be set in the area (except in the high-order position) but the programmer can control data transfer by setting word marks in the receiving fields.
2. If the length of the whole record can also vary, the programmer should reserve an area equal to the largest possible record size.

EQU — Equate

General Description: An EQU statement assigns a symbolic label to an actual or symbolic address. Thus, the user can assign different labels to the same storage location in different parts of his source program.

The programmer:

1. Writes the operation code (EQU) in the operation field.
2. Writes a symbolic address for the new label in the label field.
3. Writes an actual or symbolic address in the operand field. This address can have indexing, and address adjustment.

The processor:

1. Assigns to the label of the equate statement the same actual address that is assigned to the symbol in the operand field (with appropriate alteration if indexing and address adjustments are indicated).
2. Inserts this actual address wherever the label appears as the operand of another symbolic program entry.

Result: The programmer can now refer to a storage location by using either name.

Examples: Figure 23 shows the label INDIV equated to MANNO which has been assigned storage location 1976. Whenever either MANNO or INDIV appear in a symbolic program, 1976 will be used as the actual address.

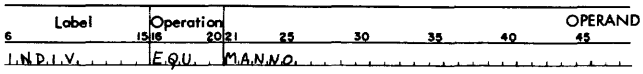


Figure 23. Equating Two Symbolic Addresses

Figure 24 shows an equate statement with address adjustment. If FICA is assigned location 890, WHTAX will be equated to FICA-10 (880). WHTAX now refers to a field whose units position is 880.

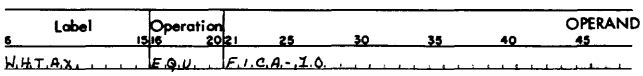


Figure 24. Address Adjustment in an EQU Statement

Figure 25 shows a label assigned to an actual address. Assume that an input card contains NETPAY in card columns 76-80. When this card is read into storage, the area locations 076-080 contain net pay. This field can be referred to as NETPAY if the EQU statement in Figure 25 is written in the source program.

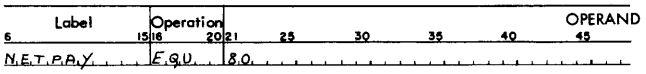


Figure 25. Assigning a Label to an Actual Address

Figure 26 shows how an equate statement can be indexed. With indexing, the label is indexed by the index location specified in the EQU statement, whenever it appears as an operand in a symbolic program entry, unless the operand in which it appears is itself indexed. In Figure 26, the address assigned the symbolic label CUSTNO is equated to the actual address of JOB + the contents of index location 3. However, if CUSTNO + X2 or CUSTNO + X1 appears as the operand of another symbolic program entry, the actual address of JOB will be added to the contents of index location 2 or 1. Thus, the indexing in an instruction takes precedence.

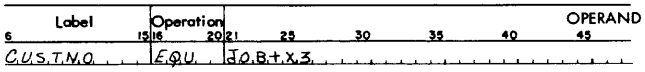


Figure 26. Indexing an EQU Statement

Figure 27 shows the symbol FIELDA equated to an asterisk address. The asterisk refers to the rightmost position of the last instruction or data whose location was assigned by the processor. Assume that this address is 698. FIELDA is now equal to 698.

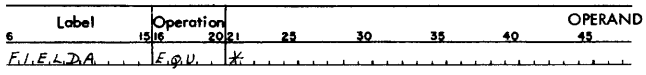


Figure 27. Equating with an * Operand

Figure 28 shows how a label can be assigned to an index location. Because the actual core-storage address of index location 1 in the IBM 1401 is 089, the EQU statement assigns the label INDEX 1 to that index location. INDEX 1 is now equal to 089. NOTE: An index location so equated must still be coded X1, X2, or X3 when used to index an operand.

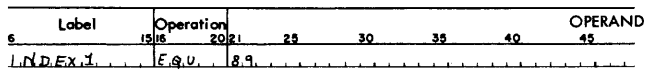


Figure 28. Assigning a Label to an Index Location

Figure 29 shows how a tape unit can be assigned a label. In this case, the programmer wishes to refer to tape 4 as INPUT, which is now equal to %U4.

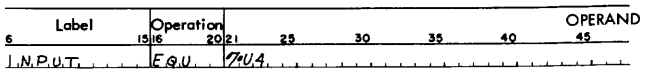


Figure 29. Assigning a Label to a Tape Unit

Imperative Operations

General Description: Autocoder imperative operations are direct commands to the object computer to act upon data, constants, auxiliary devices, or other instructions. These are the symbolic statements for the instructions to be executed in the object program. Most of the statements written in a source program will be imperative instructions. Although the *Autocoder* processor can assemble instructions with all the imperative operation code mnemonics which are shown in Figure 2, the programmer must keep in mind the particular special features and devices that will be included in the object machine that will be used to execute the program he is writing.

The programmer:

1. Writes the mnemonic operation code for the instruction in the operation field.
2. If the instruction is an entry point for a branch instruction elsewhere in the program or if the programmer wishes to make other reference to it, it must have a label. This label will be assigned an actual address equal to the address of the operation code of the assembled machine-language instruction. Thus, the programmer can use this label as the symbolic I-address of a branch instruction elsewhere in the program (see example, Figure 33).
3. Writes the symbolic address of the data, devices, or constants in the operand field. The first symbol will be used as the A- or I-address of the imperative instruction. If the instruction also requires a B-address, a comma is written following the first symbol and its address adjustment and/or indexing codes (if any); then the symbol for the B-address is written. If the instruction requires that a d-character be specified, a comma and the actual d-character follow the symbolic entries for the B-address or A/I-address if the B-address is not needed (see also *Address Types*).

NOTES

Unique Mnemonics. Several mnemonic operation codes have been developed to relieve the programmer of coding the d-character in the operand field of

symbolic imperative instructions. However, some operation codes have so many valid d-characters that it is impractical to provide a separate mnemonic for each. In these cases, the programmer supplies the d-character as previously described. In the listing of mnemonic operation codes for imperative instructions (Figure 2) all mnemonics which require that the d-character be included in the operand field are indicated by a †.

Mnemonics referring to magnetic tape do not require d-characters. However, it is necessary to specify, in the operand, the number of the tape unit needed for the operation. This can be done in one of three ways.

The programmer can:

a. Assign a label to the tape unit as described in EQU and use it as the A-operand of a tape instruction.

b. Write the number of the tape unit in column 21 of the tape instruction. The assembled instruction for the symbolic entry shown in Figure 30 will cause a record to be written on tape unit 4 using the data beginning in a storage area labeled OUTPUT.

c. Write the actual address (for example, %U4) in the A-operand field.

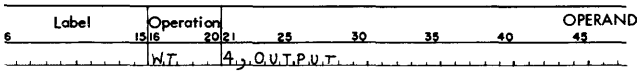


Figure 30. Write Tape

Compatability with IBM 1410 *Autocoder*. To make IBM 1401 *Autocoder* language compatible with its IBM 1410 counterpart, five new mnemonic op codes are provided that have the same function as five mnemonics presently available in SPS. When coding in *Autocoder* language, the programmer can use either mnemonic. These new mnemonics are shown in Figure 31, together with their meanings and their SPS equivalents.

Autocoder Mnemonic	SPS Mnemonic	Meaning
MLC	MCW	Move Characters to Word Mark
MLCWA	LCA	Move Characters and Word Marks to Word Mark in A-Field
MLNS	MN	Move Numerical Portion of Single Character
MLZS	MZ	Move Single Zone
MRCM	MCM	Move Characters to Record Mark or Group Mark-Word Mark

Figure 31. Alternate Move Mnemonics

The processor:

1. Assembles the object instruction as follows:
   a. Substitutes the actual machine language operation code for the mnemonic written in the operation field.
   b. Substitutes the actual addresses of symbols used in the operand field to specify the A- or I-, and B-addresses of the instructions. If address adjustment and/or indexing is indicated, the substituted address will reflect these notations (tag bits will be inserted for indexing and addresses will be altered by adding or subtracting the adjustment factor if address-adjustment is specified). The d-character will be supplied automatically for unique mnemonics, or will be taken from the operand field if the programmer has supplied it.
   c. Assigns the actual machine language instruction an area in storage. The address of this area is the position which the operation code occupies in object machine core storage. This address is assigned to the label if one appears in the label field.

Result: This instruction will be placed in the self-loading object program deck or tape. A word mark will be set in the operation code position by the loading routine at program load time.

Examples: Figure 32 shows an imperative instruction with I- and B-operands and a mnemonic which requires that the programmer include the d-character. A branch to a location labeled READ will occur if the location labeled TEST has a 5 in it. Assume that the address of READ is 596 and TEST is in 782. The assembled instruction is B 596 782 5.

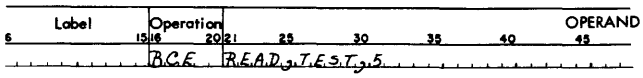


Figure 32. Branch if Character Equal

Figure 33 shows an imperative instruction with a unique mnemonic. A branch to a location labeled OVFLO will occur if an arithmetic overflow has occurred. Assume that the address of OVFLO is 896. The assembled machine language instruction is B 896 Z.

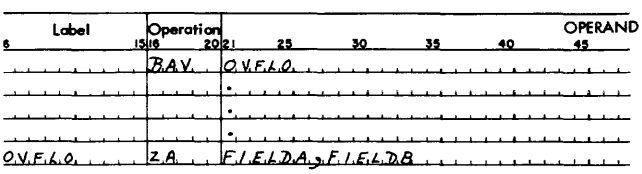


Figure 33. Branch if Arithmetic Overflow

Processor Control Operations

Autocoder has several control operations that enable the user to exercise some control over the assembly process. They are:

Op Code	Purpose
JOB	Job Card
CTL	Control Card
ORG	Origin
LTORG	Literal Origin
EX	Execute
XFR	Transfer
SFX	Suffix
ENT	Enter New Coding Mode
END	End Assembly
ALTER	Alter

JOB — Job

General Description: This is the first card in the user's source program deck. It is used to print a heading line on each page of the output listing from the assembly process and to identify the self-loading program deck or tape.

The programmer:

1. Writes the mnemonic operation code (JOB) in the operation field.
2. Writes in the operand field the indicative information to be printed in the heading line. This information may be any combination of valid 1401 characters and appears in columns 21-72.
3. Writes in the identification field the information to be contained in the self-loading program deck or tape.

The processor:

1. Prints the information, the identification number from columns 76-80, and a page number from the JOB card on each page of the output listing. If there is no JOB card, the processor will generate one. In this case nothing will be printed in the heading line, except the page number.
2. Punches the identification number (cols. 76-80) in all condensed cards produced for the object program. If another JOB card (or cards) appear elsewhere in the source program, the new identification number will be punched in subsequent condensed cards. This new JOB card will also cause the carriage to restore during listing, and the new information will appear in the heading line.

Result: The programmer can identify a job or parts of a job in the output listing.

CTL — Control

General Description: The control statement is the second entry (card) in the source program deck. The user prepares this card to specify the size of the processing machine, the size of the object machine, the type of output he wishes, and the presence or absence of the Modify-Address feature. NOTE: The modify address (MA) instruction is standard in IBM 1401 systems with 8-, 12-, and 16-thousand positions of core storage. For an object machine *not* equipped with the MA feature, the *Autocoder* processor automatically assembles a routine to simulate the MODIFY-ADDRESS instruction.

The programmer:

1. Writes the operation code (CTL) in the operation field.
2. Writes codes in the operand field as follows:

Column 21 indicates the storage size of the machine to be used to process the *Autocoder* entries.

Storage Size	Code
4,000	3
8,000	4
12,000	5
16,000	6

Column 22 indicates the storage size of the object machine.

Storage Size	Code
1,400	1
2,000	2
4,000	3
8,000	4
12,000	5
16,000	6

Column 23 indicates the type of *Autocoder* output desired.

Output	Code
Printed listing containing the symbolic source program and the machine-language object program.	Blank or 0
Printed listing and self-loading condensed program card deck.	1
Printed listing and self-loading program tape.	2
Printed listing, condensed card deck, and self-loading program tape.	3

Printed listing and one-instruction - per - card resequenced source deck.	4
Printed listing, condensed card deck and one-instruction-per-card resequenced source deck.	5
Printed listing, self-loading program tape, and one-instruction - per - card resequenced source deck.	6
All output options.	7
Error — list only	Any other code

Column 24 indicates the presence or absence of the Modify-Address feature in the object machine. The code 1 in column 24 specifies that MA is present. If column 24 is blank, the processor treats the mnemonic operation code MA as a macro-instruction and generates the instructions necessary to modify an instruction address (SET WORD MARK, ADD AND CLEAR WORD MARK) for object machines less than 8k.

Column 25. A code 1 in column 25 indicates the presence of a fifth tape which will contain the output listing and images of the condensed cards.

Column 26. A code 1 in column 26 indicates the presence of the Read-Punch Release Feature.

The processor: Interprets the codes and processes the source program accordingly.

NOTE: If columns 21 and 22 are left blank or are coded incorrectly, or if the CTL card is missing, the processor assumes that both the processing machine and the object machine have 4,000 positions of core storage. If column 23 is left blank or punched incorrectly, or if the CTL card is missing, the processor provides a listing only.

ORG — Origin

General Description: An origin statement can be used by the programmer to specify a storage address at which the processor should begin assigning locations to instructions, constants and work areas in the symbolic program.

The programmer:

1. Writes the mnemonic operation code (ORG) in the operation field.
2. Writes the symbolic, actual, blank, or asterisk address in the operand field. Symbolic or blank, or * addresses can have address adjustment (including X00) but indexing is *not* permitted in ORG statements.
3. If a symbolic label appears in the operand field of an ORG statement, it must appear in the label field elsewhere in the program sequence. It *need* not precede the ORG statement.

The processor:

1. Assigns addresses to instructions, constants and to work areas as specified in the operand field of the ORG statement.
2. If there is no ORG statement preceding the first symbolic program entry, the processor automatically begins assigning storage locations at 333 (the first storage location following the fixed 1401 read, punch, and print areas).
3. An ORG statement inserted at any point within the symbolic program causes the processor to assign subsequent addresses beginning at the address specified in the operand field of the new ORG statement.

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

Examples: Figure 34 shows an ORG statement with an actual address. The first symbolic program entry following this ORG statement will be assigned with storage location 500 as a reference point. (If the first entry is an instruction, the op code position (I-address) of that instruction will be 500; if the first entry is a 5-character DCW, it will be assigned address 504, etc.)

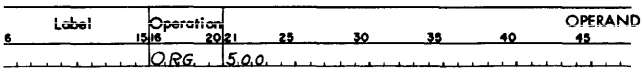


Figure 34. ORG Statement with an Actual Address

The ORG statement in Figure 35 shows how the programmer can direct the processor to save the address of the last storage location allocated. The label ADDR is the symbolic address of the storage locations used to save this address. The processor will continue to assign addresses beginning at the actual address of START.

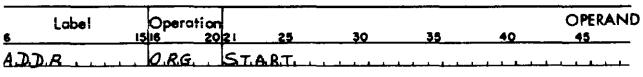


Figure 35. Saving the Address of the Last Storage Allocation

The programmer can insert another ORG statement later in the source program to direct the processor to begin assigning storage at ADDR (Figure 36).

NOTE: If a symbolic label appears in the label field of

an ORG or LTORG statement, it cannot be used in any other place except as the operand of another ORG or LTORG statement.

Figure 36 shows an ORG statement that directs the processor to start assigning addresses with the actual address assigned to ADDR (see step 3 *Programmer*).

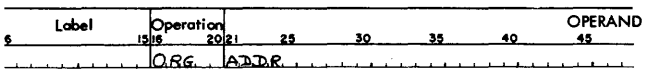


Figure 36. ORG Statement with a Symbolic Address

Figure 37 shows an ORG statement that directs the processor to bypass 200 positions of core storage when assigning addresses. This statement is the type that is included within the source program (see step 3 *Processor*).

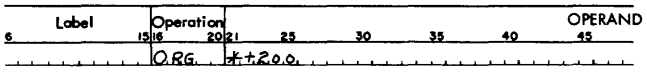


Figure 37. ORG Statement with an Asterisk Operand and Address Adjustment

When the processor encounters the statement shown in Figure 38, it will assign subsequent addresses beginning with the next available storage location whose address is a multiple of 100. For example, if the last constant was assigned location 525, the next instruction would have an address of 600.

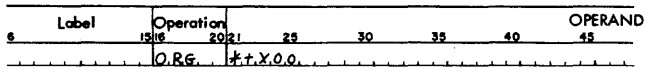


Figure 38. ORG Statement Advancing Address Assignment to the next Available Address which is a Multiple of 100

NOTE: +X00 is permitted as character adjustment in any ORG or LTORG statement.

Figure 39 shows an ORG statement with a blank operand. The processor will assign addresses to subsequent entries beginning with the lowest numbered storage address (beyond 332) which has not yet been assigned to other entries.

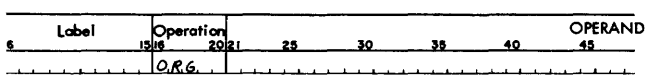


Figure 39. ORG Statement with a Blank Operand

LTORG — Literal Origin

General Description: LTORG statements are coded in the same way as ORG statements. They direct the processor to assign storage locations to previously encountered literals and closed library routines, beginning with the address written in the operand field of the LTORG statement. LTORG statements can appear anywhere in the source program.

If no LTORG statement appears in the source program, the processor begins assigning addresses to literals and closed library routines when it encounters an EX or END statement.

Example: Figure 40 shows how the programmer can direct the processor to begin assigning storage locations to literals and closed library routines.

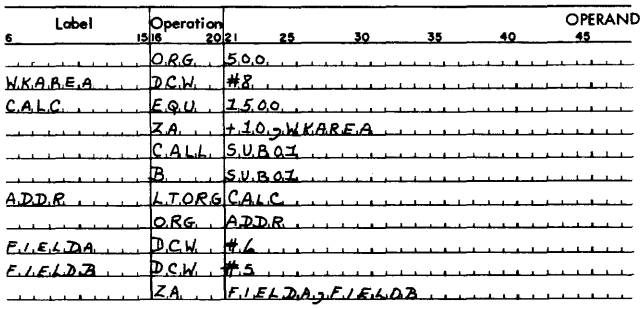


Figure 40. Using a LTORG Statement

The programmer has instructed the processor to begin storage allocation at 500. All instructions, constants, and work areas (ending with BSUB01) will be assigned storage. However, the literal (+10) in the statement ZA +10, WKAREA, and the library routine (SUB 01) extracted by the CALL macro (see *Call*) will not be assigned storage until the LTORG statement is encountered. The first instruction in the library routine (SUB 01) will be assigned address 1500 (V00) because CALC has been equated to 1500. After all instructions in SUB 01 have been assigned storage locations, the literal +10 will be assigned an address. The processor will begin assigning the rest of the instructions, constants, and work areas with the storage location immediately to the right of the area occupied by the instruction BSUB01. Thus, if BSUB01 (BV00) is assigned locations 591-594, FIELDA will be assigned storage locations 595-600.

EX — Execute

General Description: During the loading of the assembled machine-language program, the programmer may want to discontinue the loading process temporarily to execute a portion of the program just loaded. This is especially the case when the program has more than one section or overlay. The EX statement is used for this purpose.

The programmer:

1. Writes the mnemonic operation code (EX) in the operation field.
2. Writes an actual or symbolic address in the operand field. This address must be the same symbol that appears in the label field of the first instruction to be executed.

The processor:

1. Incorporates closed library routines, literals, and address constants in the program.
2. Assembles a branch instruction, the I-address of which is the address assigned to the instruction referenced by the symbol in the operand field (an unconditional branch to the first instruction to be executed). This instruction does not become part of the assembled machine-language program, but it causes the processor-produced loading routine to halt the loading process at the appropriate time and execute the branch instruction.

NOTE: To continue the loading process after the desired portion of the program has been executed, the programmer must provide re-entry to the load routine.

Figure 41 shows an example of this coding when the condensed card deck is used. The read area is cleared, word marks are set in 024, 056, 063 and 067; and a card is read with a branch to 056.

NOTE: The programmer must be sure that a word mark is present in the location following the R056 instruction at program execution time.

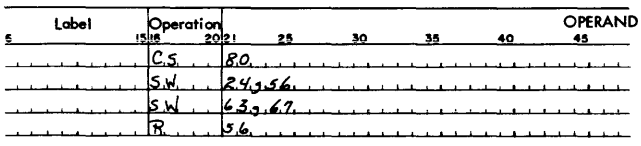


Figure 41. Re-entry to the LOAD Routine

Result: The programmer can use several program sections if his total program exceeds the limits of available storage capacity. For example, if input to the program is on magnetic tape and the program is also on tape, one tape unit can be assigned to the program and another can be assigned to the input data.

Example: Figure 42 shows how an EX statement can be coded. When this statement is encountered in the loading data, the loading process halts and a branch to the instruction whose label is ENTRYA occurs.

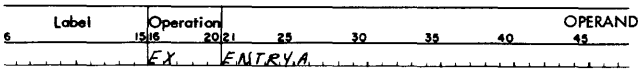


Figure 42. EX Statement

XFR — Transfer

General Description: This entry has the same function as an EX statement except that literals, closed library routines, and address constants are not stored. An XFR statement transfers to and executes instructions which have been previously loaded.

END — End

General Description: This is always the last card in the source deck. It is used to signal the processor that all of the source program entries have been read, and to provide the processor with the information necessary to create a bootstrap card. This bootstrap card causes a transfer to the first instruction in the object program after it has been loaded into the machine at program load time. Thus, program execution begins automatically.

The programmer:

1. Writes the mnemonic operation code (END) in the operation field.
2. Writes in the operand field, the symbolic or actual address of the first instruction to be executed after the program has been loaded.

The processor: Creates a clear and branch instruction which is used as part of the loading data. The read area is cleared.

SFX — Suffix

General Description: This statement directs the processor to put a suffix code in the sixth position of all labels in the symbolic program which 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.

The programmer:

1. Writes the mnemonic operation code (SFX) in the operation field.
2. Writes the character (which can be any valid 1401 character) to be used for the suffix code in the operand field.

The processor:

1. Inserts the suffix code in the sixth position of all labels in the source program which have fewer than 6 characters.
2. Changes the suffix code when a new SFX card is encountered.

Result: Each program section has unique labels.

Example: Figure 43 is an example of coding for a suffixing operation.

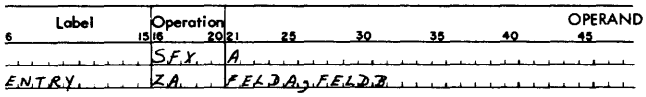


Figure 43. Specifying a SUFFIX Operation

Figure 44 shows how the processor suffixes the entries shown in Figure 43.

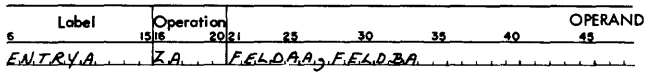


Figure 44. Suffixed Entry

ENT — Enter New Coding Mode

General Description: The 1401 *Autocoder* processor accepts source programs coded in either free-form *Autocoder* language or in fixed-form SPS language. It is also possible to assemble a single program coded in a combination of the two languages. An ENT statement is used by the programmer to inform the processor that a change in coding form follows.

The programmer:

1. Writes the mnemonic operation code (ENT) in columns 16, 17, and 18 when entering the SPS mode from the *Autocoder* mode; and columns 14, 15, and 16 when entering the *Autocoder* mode from the SPS mode.
2. Writes SPS in columns 21, 22, and 23 to enter the SPS mode from *Autocoder*; and AUTOCODER in columns 17-25 to enter the *Autocoder* mode from SPS.

NOTE: If the program is coded entirely in SPS form, the program must be preceded by an ENT statement. If this ENT card is missing, or if a coding form change is encountered with no ENT card preceding it, an error condition will result. Before assembly, remove the SPS control card from the original SPS source deck and replace it by an *Autocoder* JOB card, an *Autocoder* CTL card, and an ENT card in *Autocoder* format.

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

Result: Programs prepared wholly or partially in SPS format can be reassembled by the *Autocoder* processor.

Example: Figures 45 and 46 are ENT statements to be used with *Autocoder.*

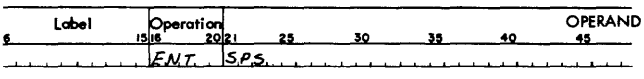


Figure 45. ENT Statement for Entering SPS Mode

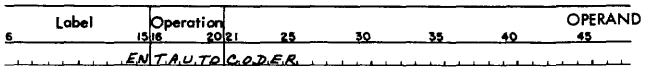


Figure 46. ENT Statement for Entering Autocoder Mode

ALTER — Alter

General Description: An ALTER statement makes it possible to add, delete, or substitute instructions in the object program after the original assembly has been completed.

By saving tape 4 which, at the end of assembly, contains a source program, it is possible to reassemble the program easily by processing ALTER cards. During each assembly, each statement that can be altered by an ALTER entry is assigned a sequence number. This number is listed in the first column of the output listing. These numbers are used in the ALTER entries to reference statements to be changed during the reassembly.

ADDITIONS

The programmer:

1. Writes the mnemonic operation code (ALTER) in the operation field of the ALTER statement.
2. Writes a number in the operand field in column 21. This number represents the sequence number after which the entries following the ALTER statement should be included.
3. Writes the statements to be included.

The processor: Adds the new statements and reassembles the object program.

Example: The programmer wishes to insert two statements after the statement whose sequence number is 32. The three entries shown in Figure 47 are used.

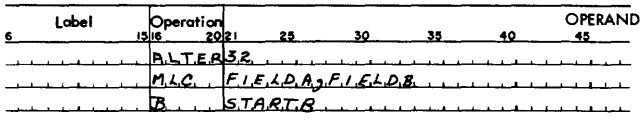


Figure 47. Adding Statements to an Assembled Object Program

NOTE: All statements following an ALTER entry will be included in the object program until the next control card or last card has been read.

DELETIONS

The programmer:

1. Writes the mnemonic operation code (ALTER) in the operation field of the ALTER statement.
2. Writes two numbers separated by commas in the operand field. The first of these numbers is the

sequence number of the first statement to be deleted. The second number is the sequence number of the last statement to be deleted. NOTE: If only one statement is to be deleted, only the sequence number is written twice in the operand field.

The processor: Deletes object program statements included between the two sequence numbers in the operand field.

Example: If the programmer wishes to delete object program statements 92 through 103, he uses the entry shown in Figure 48.

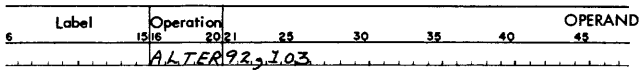


Figure 48. Deleting Statements from an Assembled Object Program

SUBSTITUTIONS

The programmer:

1. Writes the ALTER statement exactly as described under deletions.
2. Writes the statements to be substituted.

The processor:

1. Deletes the statements included by the sequence numbers in the operand field.
2. Substitutes the statements following the ALTER entry.

Example: The entries shown in Figure 49 cause the processor to delete the statement whose alter number is 62 and add in its place the MLC and B instructions.

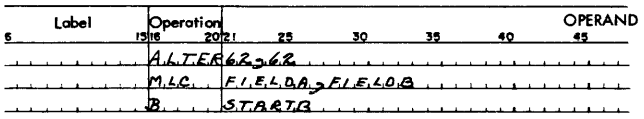


Figure 49. Substituting Statements in an Assembled Object Program

The Macro System

Many of the routines that must be incorporated in programs written for the IBM 1401 are general in nature and can be used repeatedly with little or no alteration. The IBM 1401 *Autocoder* makes it possible for the user to write a single symbolic instruction (a *macro-instruction*) that causes a series of machine language instructions to be inserted automatically in the object program. Thus, the ability of *Autocoder* to process macro-instructions relieves the programmer of much repetitive coding. With a macro-instruction, the programmer can call, from a library of routines, a sequence of instructions tailored by the processor to fit his particular program.

Definition of Terms

Several programming terms describe the requirements and operational characteristics of the macro system. These terms are explained here as they are applied in the following discussions.

Object Routine. The specific machine language instructions needed to perform the functions specified by the macro-instruction. If the object routine is inserted directly in a larger routine (e.g., the main routine) without a linkage or calling sequence, it is called an *open routine* (or in-line routine). If the routine is not inserted as a block of instructions within a larger routine, but is entered by basic linkage from the main routine, it is called a *closed routine* (or off-line routine).

Model Statement. A general outline of a symbolic program entry. Model statements are used only in flexible library routines.

Library Routine. The complete set of instructions or model statements from which the object routine is developed. If the library routine can not be altered, it is *inflexible*. If the library routine is designed so that symbolic program entries can be deleted from certain object routines (at the discretion of the programmer), or if parameters can be inserted, it is *flexible*.

Library. The complete set of library routines, stored on magnetic tape with an identifying label for each routine, that can be extracted by a macro-instruction. Several macro-instructions and library routines are provided by IBM (see *Supplied Macros*). Others are designed by the user to suit particular processing requirements.

Librarian. The phase of the processor that creates the library tape from card input. After the original writing of the library tape, this phase is used to insert additional library routines and their identifying labels. This phase is omitted during program assembly.

Parameters. The symbolic addresses of data fields, control names, or information to be inserted in the symbolic program entries outlined by the model statements. By placing parameters in the operand field of a macro-instruction, the programmer can specify symbolically the data to be operated on. The actual addresses of the data (or other information) are inserted in the object routine by the processor during assembly.

Macro Operations

To illustrate the basic operation of the macro system, a macro called COMPR with a simple flexible library routine is used. The routine is designed to read a card, compare an input field to another field, test the compare indicator for a high, low, or equal condition or any combination of the three. For example, in some programs it will be necessary to test only for an equal condition; in others, high or equal, etc.

The library entry, a macro-instruction specifying that all instructions in the library routine appear in the object program, and the symbolic program entries created during the macro phase of *Autocoder* are shown in Figure 50. The symbolic program entries are inserted in the source program behind the macro-instruction. During assembly of the object program, the symbolic program entries will be translated to actual machine language instructions with the actual addresses of the parameters inserted in the label, operation, and operand fields.

The Library Entry

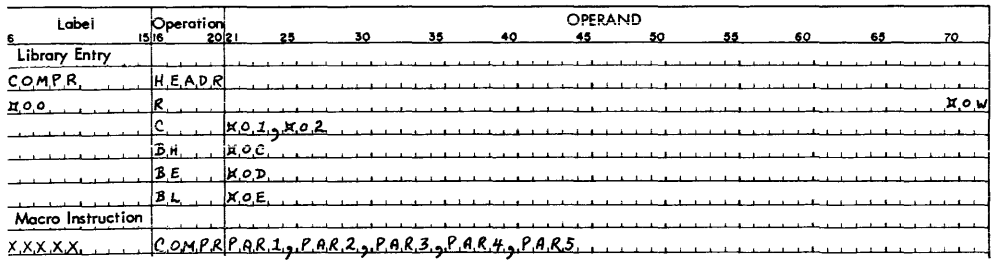
The library entry for the COMPR macro was created by writing a header statement and five model statements as shown in Figure 50.

HEADR — Header

General Description: A header statement identifies a library routine. This identification precedes the library routine in the library tape.

The programmer:

1. Writes the operation code (HEADR) in the operation field.
2. Writes the five-character label for the library routine in the label field. This label will be the same as the name that appears in the operation field of the associated macro-instruction (except when either the CALL or INCLD macro is used). The first



Assembled Symbolic Program Entries

XXXXX	R	
	C	PAR1, PAR2
	BH	PAR3
	BE	PAR4
	BL	PAR5

Figure 50. Macro Operations

three characters must be unique (no two library entry labels may have the same first three characters).

The processor: Puts the indicative information ahead of the model statements in the library tape during the librarian phase of *Autocoder.*

Result: During assembly, the header label is matched with the macro name in the operation field of the macro-instruction. The model statements following the header label in the library tape are used to assemble the symbolic program entries as specified by the macro-instruction.

Model Statements

General Description: Model statements establish the conditions for insertion of parameters in the object routine and define the basic structure of the symbolic program entries.

The programmer:

1. Designs a general routine to perform a specific function when it is executed in the object program.
2. Writes the model statements as follows:

   a. If the entry is complete, it is written exactly as though it were an entry in a source program. This entry will be included in all object routines unless a bypass condition exists.

Example: Read a card (Figure 51).

b. If the entry is incomplete, the programmer writes a special three-character code to indicate that a certain parameter from the macro-instruction operand field *must* be inserted (substituted) in its place. This code is a □ followed by a number from 01 to 99 (the position of the parameter in the macro-instruction). This entry will be inserted in all object routines unless a bypass condition exists.

Example: Insert parameters 01 and 02 specified by the COMPR macro-instruction as shown in Figure 52.

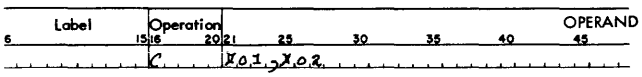


Figure 52. Model Statement Specifying that the First and Second Parameters Be Present in the Associated Macro-Instruction

c. If the entry is incomplete, the programmer writes a □ followed by a number from 01-99 with AB-bits over the units position (parameter 01 is □ 0 A; parameter 02 is □ 0 B; etc.) to indicate that the entry is to be included in the object routine only if the parameter is specified by the macro-instruction and no bypass condition exists.

Example: Insert parameter 03 in the following instruction if it is specified by the macro-instruction. If

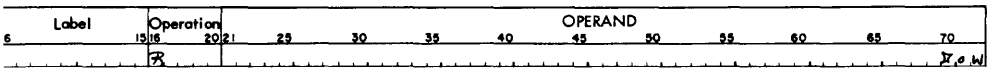


Figure 51. Model Statement for a Complete Instruction

parameter 03 does not appear in the macro-instruction, the instruction shown in Figure 53 will be deleted from the object routine.

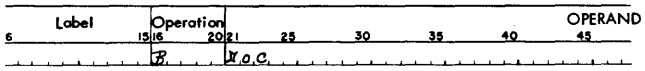


Figure 53. Model Statement for an Incomplete Instruction with Conditional Parameters

NOTE: Substitution codes can be used to substitute a parameter in any part of a model statement. For example, it is possible to substitute an operation code, any part of a literal, a label, etc.

Bypassing. The 1401 *Autocoder* processor permits the programmer to establish multiple conditions for bypassing model statements in the library routine. Any of the three basic types of model statements can be bypassed if certain parameters are missing from or present in the macro-instruction and if special condition codes are included in the right-hand portion of the operand field (comments field). The first code may be placed in columns 70, 71, and 72; the second code in 67, 68 and 69, etc. These codes are interpreted by the processor as follows:

a. If the code is a □ followed by a number from 01 to 99 with AB-bits over the units position (for example □ 0 A), the model statement will be bypassed if the indicated parameter is missing from the macro-instruction.

Example: Bypass the model statement shown in Figure 54 only if either parameter 04 or 05 is missing from the operand field of the macro-instruction.

b. If the code is a □ followed by a number from 01 to 99 with an A-bit over the units position (for example □ 0 /), the model statement will be bypassed if the indicated parameter is present in the macro-instruction.

Example: Bypass the model statement shown in Figure 55 if either parameter 04 or 05 is present in the operand field of the macro-instruction.

c. Combinations of the two types of conditions for the same model statement are permissible.

Example: Bypass the model statement shown in Figure 56 if parameter 04 is present or if parameter 05 is missing.

NOTE: The processor scans the condition codes from right to left. If a bypass condition is encountered, the model statement is not used for the object routine. There must be at least two non-significant blank spaces between the operand(s) of the model statement and the leftmost condition code.

Labelling. If the model statement represents an instruction that is the entry point for a branch instruction elsewhere in the program, it must have a label.

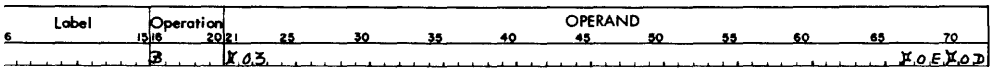


Figure 54. Condition Codes for Bypassing if Parameters Are Missing from the Associated Macro-Instruction

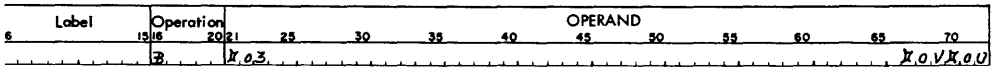


Figure 55. Condition Codes for Bypassing if Parameters Are Present in the Associated Macro-Instruction

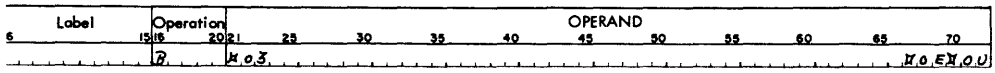


Figure 56. Condition Codes Combined

A □ 0 0 code in the first such model statement causes the contents of the label field of the macro-instruction to be inserted in the label field of the assembled symbolic entry as shown in Figure 57.

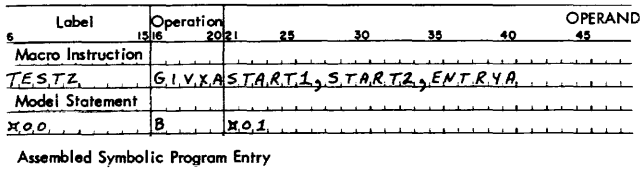


Figure 57. Labelling

If additional external labels are required and specified as parameters in the macro-instruction, they can be inserted in the label field of the symbolic program entry by using a □ 01-99 code.

Example: Insert parameter 02 in the label field of the assembled symbolic program entry as shown in Figure 58.

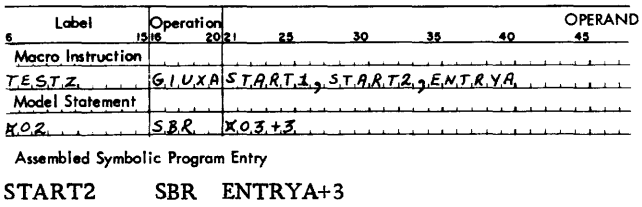


Figure 58. Additional External Label

Symbolic Addressing within the Library Routine. To allow symbolic reference to other instructions in a flexible library routine, a □ followed by a number from 01 to 99 with a B-bit over the units position (□ 0 J = symbolic address 1; □ 0 K = symbolic address 2, etc.) can be used. The processor generates the symbolic address if the code (for example, □ 0 J) is used as a label for one entry and as an operand of at least one other entry in the same library routine.

Internal labels within flexible routines are generated in the form □ nnmmm, where *nn* is the code (0J-9R), and *mmm* is the number of the macro within the source program. This avoids duplicate address assignments for labels.

Example: Use the generated symbolic address of (□ 0 J) as an operand for entry 3 and as the label for entry 6. UPDAT is the 23rd macro encountered in the source program (Figure 59).

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

If address adjustment is used in both the parameter and the instruction, the assembled instruction will be adjusted to the algebraic sum of the two.

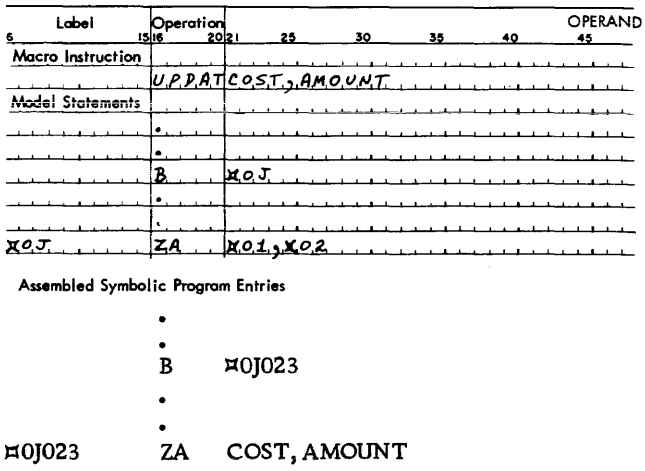


Figure 59. Internal Labels

For example, if the address adjustment of one is + 7 and the other is − 4, the assembled instruction will have address adjustment equal to + 3.

Operands may be indexed in the library routine. If a parameter supplied by the macro-instruction is indexed, the leftmost indexed code in the assembled model statement takes precedence.

Literals: Operands of instructions in flexible routines may use literals as required.

NOTES:

1. A model statement in the library routine for a macro-instruction may not be another macro-instruction, except the CALL, INCLD, or CHAIN macro (see *Call*).
2. Literal Origin, Ex and End statements cannot be used in library routines.

The processor: Enters model statements in the library tape immediately following the header statement during the librarian phase of *Autocoder.*

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

Figure 60 is a summary of the codes that can be used in the model statements of flexible library routines.

CODE	POSITION	FUNCTION
□01-□99	Statement	Substitute parameter (parameter must be present)
□0A-□91	Statement	Substitute parameter (if parameter is missing, delete statement)
□0A-□91	Comments Field (right-hand portion of operand field)	If parameter is missing, delete statement
□0/-□9Z	Comments Field	If parameter is present, delete statement.
□00	Label Field	Substitute contents of macro-instruction label field
□0J-□9R	Label field and Operand Field	Assign internal label

Figure 60. Model Statement Codes

Macro-Instructions

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

The programmer:

1. Writes the name of the library routine in the operation field. This name must be the same five characters that appear in the label field of the header statement of the library entry.
2. Writes, in the label field, the label that is to be substituted in the model statement that contains □00, if such a model statement appears in the library entry. If the □00 model statement is bypassed, the label is transferred to the next included statement.
3. Writes in the operand field the parameters that are to be used by the model statements required for the particular object routine desired as follows:

   a. Parameters must be written in the sequence in which they are to be used by the codes in the model statements. For example, if COST is parameter 1, it must be written first so that it will be (1) substituted wherever a □01, or □0A appears as an operation code or operand of a model statement and (2) tested for a missing or present condition wherever a bypass condition code (□0A or □0/) appears in the right-hand portion of the operand field.

   b. May use as many parameters as can be contained in the operand fields of five or fewer coding sheet lines. If more than one line is needed for a macro-instruction, the label and operation fields of the additional lines must be left blank. Parameters must be separated by a comma. They cannot contain blanks unless the blanks appear between @ symbols. If parameters for a single macro-instruction require more than one coding sheet line, the last parameter in each line must be followed immediately by a comma. The last parameter in a macro-instruction must not be followed by a comma.

   c. Parameters that are not required for the particular object routine desired can be omitted from the operand field of the macro-instruction. However, a comma must be inserted in place of the omitted parameter to indicate that it is missing, unless the omitted parameter is the last parameter in the macro-instruction.

   Figures 61, 62, and 63 show how parameters can be omitted. The hypothetical macro-instruction called EXACT is used. EXACT can have as many as 9 parameters.

The processor: Extracts the library routine and selects the model statements required for the object routine as specified by the parameters in the macro-instructions and by the substitution and condition codes in the model statements.

Result: The resulting program entries are merged with the source program entries behind the macro-instruction. In the listing of the source and object programs,

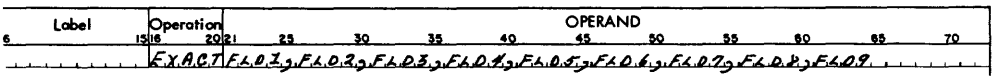


Figure 61. All Parameters Are Present

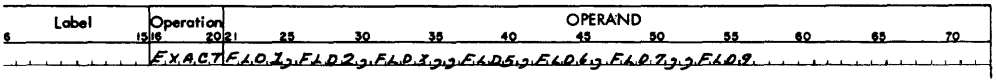


Figure 62. Parameters 4 and 8 Are Missing

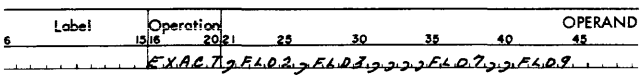


Figure 63. Parameters 1, 4, 5, 6, and 8 Are Missing

(produced by the listing and condensed cards phase of *Autocoder*) the macro-instruction is identified by a MACRO tag and the symbolic program entries generated by the processor are identified by GEN (Generated) tags.

Call Routines

The 1401 *Autocoder* processor permits the user to add inflexible routines to the library tape. These are commonly used sequences of instructions that can be extracted for an object program by the CALL macro. They differ from the routines processed by other macro-instructions in several ways:

1. All instructions must be complete (no parameters can be inserted).
2. All instructions in the routine are incorporated.
3. A CALL routine is not inserted at the point where the CALL macro was encountered in the source program. Instead, it is inserted only once as a closed routine elsewhere in the object program or program section. Linkage to the routine is provided automatically by the processor whenever its particular CALL macro is encountered in the source program. (The processor does not produce automatic linkage to the routines incorporated by other macro-instructions because these routines are inserted as open routines where the associated macro-instructions were encountered in the source program.)
4. Data needed by a CALL routine must be in the locations indicated by the symbols in the operand fields of its instructions.

Requirements: CALL routines have several specific requirements that must be considered when the routine is created:

1. Every entry point in a CALL routine must have a label. These labels (and all other symbols used in a CALL routine) must be five characters in length, and each of these must have the same first three characters. The first of these three characters must be alphabetic. The last four characters of each symbol can be alphamerical (no special characters).

   CALL routines are stored at the time and place where a Literal Origin, End, or Execute processor control statement is encountered. Duplicate symbols can occur if a CALL routine is used in more than one program overlay (if the same CALL routine is named in CALL macros that are separated by a Literal Origin or Execute statement). To eliminate this possibility the *Autocoder* processor provides a Suffix (see *SFX*) operation. The programmer should use a suffix statement containing a new character in each program section.
2. The first instruction at each entry point in a CALL routine must store the contents of the B-address register (SBR) in an index location or in the last instruction executed in the CALL routine. This provides for re-entry at the proper place in the main routine after the CALL routine is executed.
3. All macro-instruction operation codes except CALL, INCLD, and CHAIN are invalid in CALL routines. All other symbolic entries acceptable to *Autocoder*, except Literal, Origin, Execute, and End can be used. A CALL macro:

   a. allows one CALL routine to be used at some point in another CALL routine or,

   b. can be used as a model statement in the library routine for a regular macro-instruction.

IBM Supplied Macros

Six macro-instructions are currently available as part of the *Autocoder* Processor. They are: CALL, INCLD, CHAIN, MA, OVLAY, and TOVLY.

CALL Macro

General Description: The CALL macro provides access to inflexible routines written by the user and stored in the library tape. It establishes linkage to a closed routine and inserts that routine elsewhere in the program. The CALL macro is part of the *Autocoder* processor.

The programmer:

1. Writes the name of the macro (CALL) in the operation field.
2. Writes the label of the library statement which is the desired entry point in the library routine starting in column 21 of the operand field. The first three characters of this label must be the same as the first three characters in the label field of the header statement that was used to enter the routine in the library tape (see *Headr*).

   a. If the CALL routine is constructed so that all the data it requires must be taken from specifically-labeled areas of storage, the remainder of the operand field must be left blank. For example, a CALL routine whose entry point is SQR01 requires that

the number whose square root is to be computed must be placed in a location labeled SQR02. The CALL macro is written as shown in Figure 64.

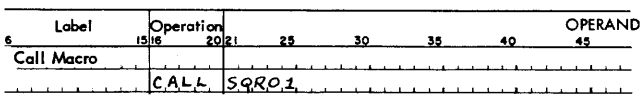


Figure 64. CALL Statement Specifying That Data be in Specifically Labelled Areas of Storage

b. If the CALL routine is constructed so that the data it requires can be located in arbitrarily-labeled areas of core storage, the symbols for these areas must be included immediately following the label in the operand field. These symbols must be entered in the order in which they are required by the CALL routine. This makes it possible to design CALL routines in which the required data can be placed in locations labeled in any way the programmer desires. This frees the source program writer from the restriction that he insert data in locations labeled according to the requirements of the CALL routine. However, CALL routines to be used in this manner must be coded to utilize the address constants that will be created from the symbols in the operand field.

Example: Call a routine whose entry point is SUB01 (Figure 65). The addresses of DATA 1, DATA 2, and DATA 3 are needed by the CALL routine.

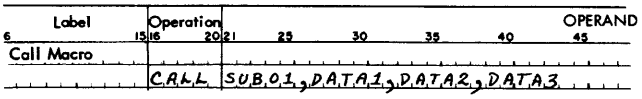


Figure 65. CALL Statement for a Routine with Arbitrary Data Storage Assignments

The processor:

1. Establishes linkage from the main routine to the CALL routine by assembling a symbolic program entry for an unconditional branch instruction. The operand for this branch instruction is the entry point given in the operand field of the CALL macro as shown in Figures 64 and 65. The branch instruction follows the CALL macro.
2. Creates address constants for other symbols appearing in the operand field of the CALL macro, and inserts them following the unconditional branch instruction as shown in Figure 65. Note that these address constants are defined in the order in which the associated symbols appear in the CALL operand.

Result: A given CALL routine is inserted once per program or program section in a location determined by a processor control statement. Branch instructions are inserted as many times as an associated CALL macro is encountered in the source program. Thus the CALL routine can be entered from several points in the main routine.

Example: Assume that a library routine to compute the value of X + Z is associated with a regular macro-instruction called TAKSQ. There is also a CALL routine in the library tape named SQR01 which calculates the square root of a number in a work area (SQR02) and places the answer in another work area (SQR03). The programmer can design a library entry for the TAKSQ macro that will provide linkage to the CALL routine as shown in Figure 66.

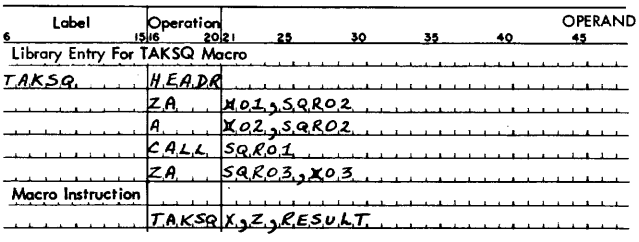
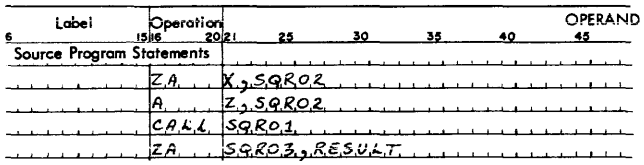


Figure 66. CALL Statement within a Library Routine for a Macro-Instruction

When the object routine is executed, X + Z will be stored in SQR02. Then the program will branch to the CALL routine where the square root of X + Z will be calculated and the result stored in SQR03. The last instruction in the SQR01 routine will cause an unconditional branch to the last instruction in the TAKSQ routine which puts the answer in an area labeled RESULT. NOTE: This illustration is designed to show the combination of a regular macro and the CALL macro. The same result could be achieved by writing entries in the source program as shown in Figure 67.



Assembled Symbolic Program Entries

```
ZA      X, SQR02
A       Z, SQR02
CALL    SQR01
B       SQR01
ZA      SQR03, RESULT
```

Figure 67. Alternative Source Program Entries

INCLD Macro

General Description: This macro extracts an inflexible library routine from the library tape. However, the INCLD macro does not insert a branch instruction following the INCLD statement in the source program as does the CALL statement. The programmer establishes his own linkage to the closed routine. INCLD statements are constructed in the same manner as CALL statements.

Example: Figure 68 shows an INCLD statement that causes a library routine named SUB01 to be incorporated in the object program.

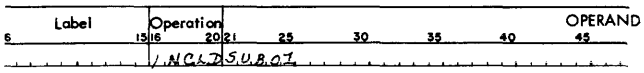


Figure 68. INCLD Statement

The processor does not produce a branch instruction. The programmer must insert a branch at the place in the main routine at which the exit to the closed routine is needed. Several INCLD statements can be written in a group in a source program to cause the associated library routines to be stored at LTORG, END, or EX time, by the processor. Thus, one exit from the main routine can cause several library routines to be executed at object time. The INCLD macro also enables the programmer to extract library routines in alphabetic sequence if he so desires. This saves assembly time because all library routines are stored in alpha sequence in the library tape.

NOTE: CALL and INCLD statements may appear in either flexible or inflexible library routines. Also, an inflexible library routine may, in turn, have CALL or INCLD statements.

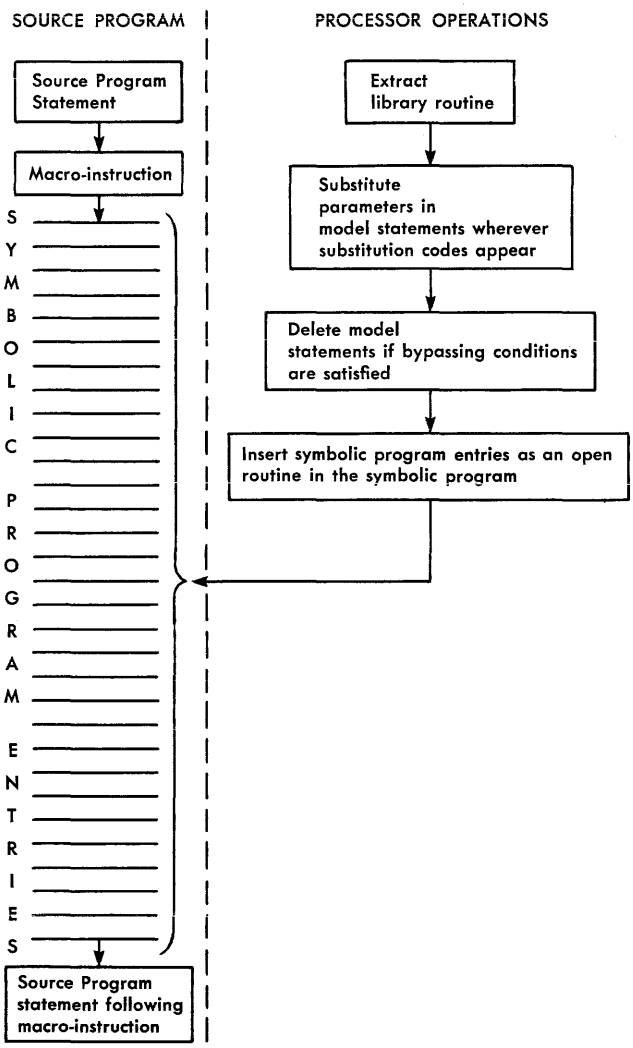
If CALL or INCLD are written *within* a library routine, only a single operand is permitted in the CALL or INCLD statement. This single operand is the name or entry point of the closed library routine. (See *Call Macro.*)

Macro Processing

Figures 69, 70, and 71 are diagrams showing the effects of the three different uses of library routines:

1. As extracted by a regular macro-instruction.
2. As extracted by the CALL macro.
3. As extracted by the INCLD macro.

The symbolic programs that result from the processor actions described in Figures 69, 70, and 71 are later processed as though the user had inserted all the entries in the source program (Symbolic entries are translated to machine-language instructions; constants cards are produced, etc.).



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

Figure 69. MACRO Processing

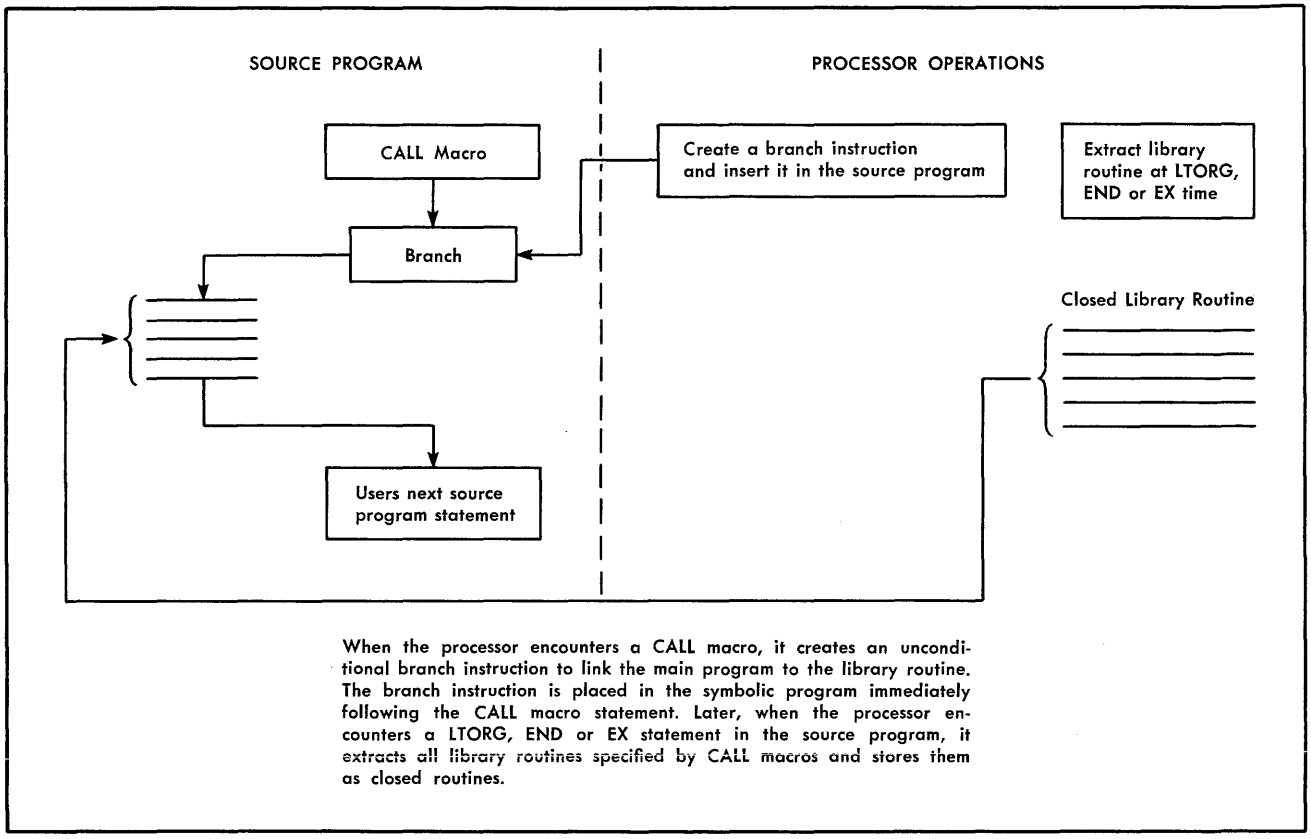


Figure 70. CALL Processing

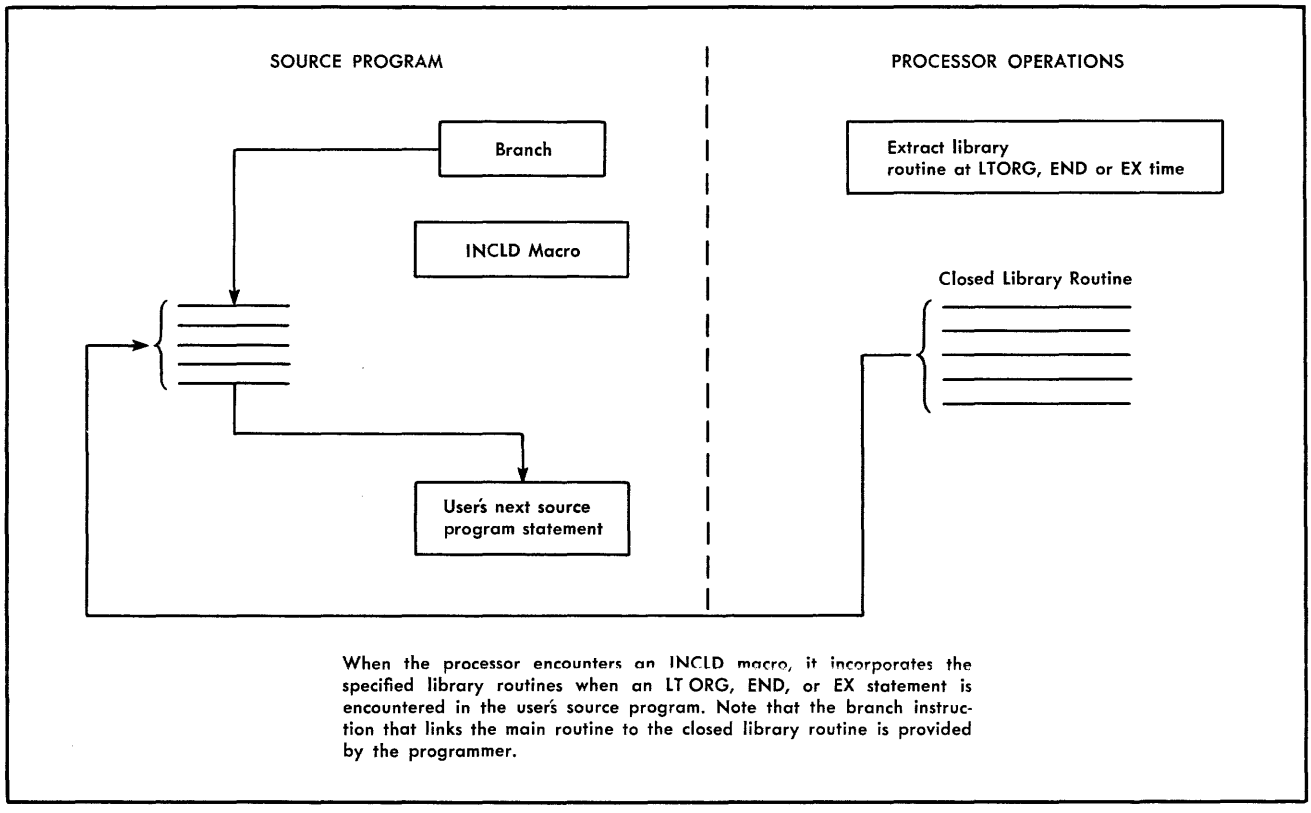


Figure 71. INCLD Processing

CHAIN Macro

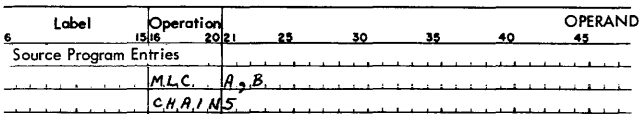
The CHAIN macro makes it easier for the programmer to code chained instructions.

The programmer:

1. Writes the instruction to be chained as usual.
2. Writes the chain statement 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 processor: Produces the desired number of additional operation codes.

Example: Figure 72 shows how an MLC statement can be chained five times.



Assembled Symbolic Program Entries

```
MLC
MLC
MLC
MLC
MLC
```

Figure 72. CHAIN Macro

OVLAY Macro — Card Overlay

General Description: This statement prepares storage and loads a new program section (overlay) from cards. The library routine for the OVLAY macro-instruction is shown in Figure 73.

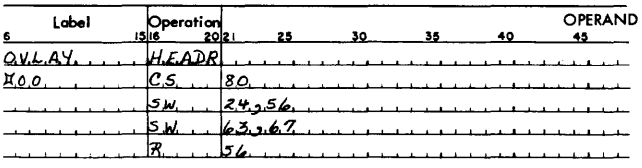


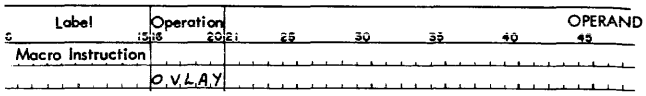
Figure 73. Library Routine for OVLAY Macro

The programmer: Writes the macro-instruction as follows:

1. Writes the name of the macro (OVLAY) in the operation field.
2. Writes in the label field, the label to be inserted in the first statement in the library routine.

Result: The library routine is extracted and the label (if any) is substituted for □00.

Example: At the end of a program section the programmer places an OVLAY macro-instruction in the source program as shown in Figure 74.



Assembled Symbolic Program Entries

```
CS    80
SW    24,56
SW    63,67
R     56
```

Figure 74. Using the OVLAY Macro

TOVLY Macro — Tape Overlay

General Description: The TOVLY macro prepares storage for and loads a new program section from magnetic tape. The library routine for the TOVLY macro-instruction is shown in Figure 75.

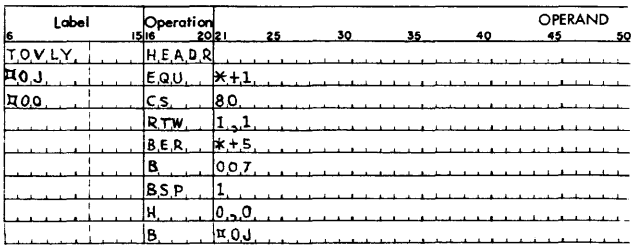


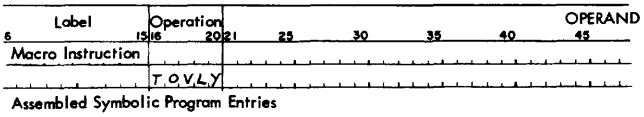
Figure 75. Library Routine for TOVLY Macro

The programmer: Writes the macro-instruction as follows:

1. Writes the name of the macro (TOVLY) in the operation field.
2. Writes in the label field the label to be inserted in the first statement in the library routine.

Result: The library routine is extracted and the label is substituted for □00.

Example: In the source program the programmer inserts the TOVLY macro as shown in Figure 76.



Assembled Symbolic Program Entries

```
¤0J023   EQU    *+1
         CS     80
         RTW    1,1
         BER    *+5
         B      007
         BSP    1
         H      0,0
         B      ¤0J023
```

Figure 76. Tape Overlay

MA Macro — Modify Address

General Description: This library routine makes it possible to modify addresses with two addresses, or a single address when MA hardware is not available. The library entry is shown in Figure 77.

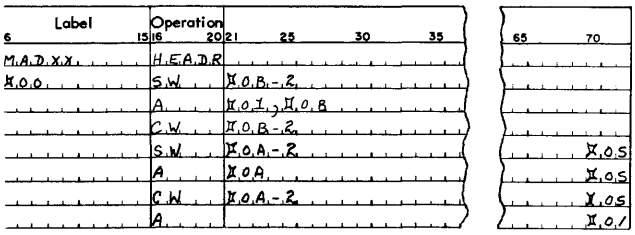


Figure 77. Library Entry for MA Macro

The programmer:

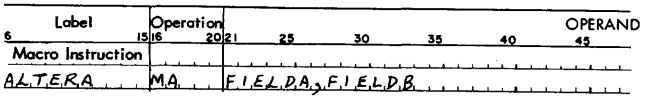
1. Writes the mnemonic operation code (MA) in the operation field.
2. May write a label in the label field.
3. Writes the macro-instruction with one or two parameters.

The processor:

1. Selects the model statements indicated by the substitution and condition codes in the library routine and the parameters in the macro-instruction.
2. Inserts the label (if any) in the first model statement used in the object routine.

Result: A group of tailored symbolic program entries is inserted as an open routine behind the macro-instruction in the source program.

Examples: Figure 78 shows the MA macro-instruction with parameters for both A- and B-addresses. The symbolic routine developed by the processor is also shown.

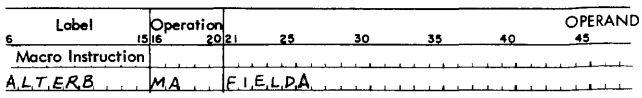


Assembled Symbolic Program Entries

```
ALTERA   SW   FIELDB-2
         A    FIELDA,FIELDB
         CW   FIELDB-2
```

Figure 78. MA Macro with Two Parameters

Figure 79 shows the MA macro-instruction with a parameter for the A-address only. The symbolic routine developed by the processor is also shown.



Assembled Symbolic Program Entries

```
ALTERB   SW   FIELDA-2
         A    FIELDA
         CW   FIELDA-2
```

Figure 79. MA Macro with One Parameter

The System Tape

The *Autocoder* system tape contains the *Autocoder* processor and the library entries which can be extracted by macro-instructions. All library routines must be stored on the system tape in alpha sequence. The IBM 1401/1410 standard collating sequence must be used.

Insertion and deletion of all or part of a library routine can also be made. The INSER and DELET statements are used for these purposes. The PRINT and PUNCH statements produce listings and punched card documents containing the library routines.

DELET — Delete

General Description: This entry deletes a library routine or parts of a library routine from the library tape.

The programmer:

1. Writes the mnemonic operation code (DELET) in the operation field.
2. Writes the name of the library routine in the label field.
3. Writes, in the operand field, the number of the model statement to be deleted. If a whole routine is to be deleted, the operand field is left blank. If more than one model statement of a continuous sequence is to be deleted, the first and last numbers must be written separated by a comma.

The processor:

1. Deletes the model statement or statements specified in the operand field.
2. Lists the action taken.

Result: The new library tape contains the modified library routine.

Examples: Figure 80 is a DELET statement that will cause the whole COMPR library routine to be removed from the library.

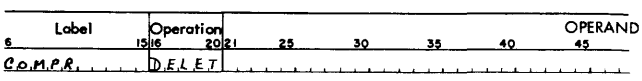


Figure 80. Deleting an Entire Library Routine

Figure 81 is a DELET statement that will cause the first model statement to be deleted from the COMPR library routine.

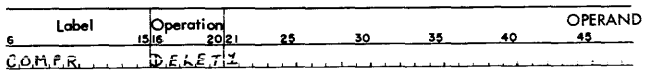


Figure 81. Deleting a Single Model Statement

Figure 82 is a DELET statement that will cause model statements 2, 3, 4, and 5 to be deleted.

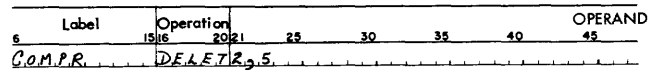


Figure 82. Deleting Multiple Model Statements

INSER — Insert

General Description: This entry inserts a whole library routine or part of a library routine in the library tape.

The programmer:

1. Writes the mnemonic operation code (INSER) in the operation field.
2. Writes the name of the library routine in the label field.
3. Writes the line number of the model statement after which the insertion is to be made. If two operands, separated by a comma, are written, the implied deletion will take place.

The processor:

1. Deletes model statements, if necessary and inserts the new model statement(s) in the library routine.
2. Lists the action taken.

Result: The library tape contains the modified library routine.

Examples: Figure 83 is an INSER statement that will cause a library routine named COMPR to be inserted in the library tape.

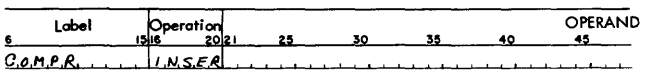


Figure 83. Inserting an Entire Library Routine

Figure 84 is an INSER statement that will cause new model statement 1 to be inserted in the COMPR library routine.

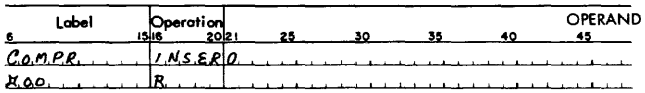


Figure 84. Inserting a Single Model Statement

Figure 85 is an INSER statement that will cause the first model statement that is presently in the library routine to be deleted and the model statement shown below to be inserted in its place.

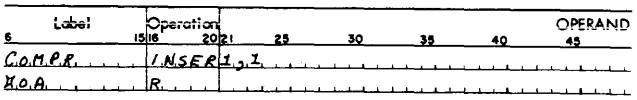


Figure 85. Substituting One Model Statement for Another

Figure 86 is an INSER statement that causes model statements 1 and 2 to be deleted and the model statements shown below to be inserted in their places.

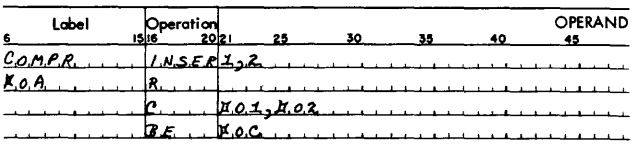


Figure 86. Substituting Multiple Model Statements

PRINT — Print Library Routine

General Description: This entry causes the processor to list a library routine with sequence numbers assigned as follows: HEADR Statement, 00; First Model Statement, 01; Second Model Statement, 02; etc.

The programmer:

1. Writes the mnemonic operation code (PRINT) in the operation field.
2. Writes the name of the library routine in the label field.

The processor: Extracts and lists the library routine.

Result: The line numbers can be used for making insertions and deletions to the library.

Example: The statement shown in Figure 87 causes the COMPR routine to be listed by the IBM 1403 printer.

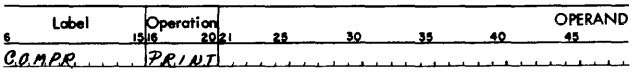


Figure 87. PRINT Statement

PUNCH — Print and Punch Library Routine

General Description: This entry causes the processor to list and punch a specified library routine.

The programmer:

1. Writes the mnemonic operation code (PUNCH) in the operation field.
2. Writes the name of the library routine in the label field.

The processor: Extracts, lists, and punches the library routine.

Result: The user has a numbered listing and a deck of cards containing all entries in the library routine.

Example: The statement shown in Figure 88 causes the library routine called COMPR to be printed and punched.

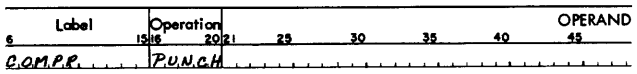


Figure 88. PUNCH Statement

Additional Language Specifications

Machine Language Coding

To permit the user to code instructions for systems equipped with special features and devices that are not otherwise handled by the 1401 *Autocoder* mnemonics, actual operation codes and d-characters may be written in *Autocoder* imperative statements.

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 actual machine language. If no d-character is needed, column 20 must be left blank.
3. May write a label in the label field as described in *Imperative Operations,* Programmer Step 2.
4. Writes in the operand field a blank, actual, symbolic, or asterisk address, or a literal or address constant. The operand field must not contain the d-character. The actual address of an input-output unit must be used unless the actual address has been equated to a symbol. For example,

LABEL	OPERATION	OPERAND	
	MR	%U1, INPUT	or
TAPE1	EQU	%U1	
	MR	TAPE1, INPUT	

are correct but,

	MR	1, INPUT	is incorrect

RAMAC® Operands

In *Autocoder* statements that use mnemonic operands, it is not necessary to specify the A-operand, and it is incorrect to use a comma to indicate that the A-address is missing. Thus, the statement

LABEL	OPERATION	OPERAND
	RD	INPUTA

results in M %F1 xxx R which reads a single record without word marks into a core-storage area whose high-order position is xxx. Two other *Autocoder* statements could be used to achieve the same machine language instruction:

LABEL	OPERATION	OPERAND	
	MCW	%F1, INPUTA, R	or
	MR	%F1, INPUTA	

Auxiliary I/O Devices

Input and output devices are available with 1401 systems for which unique mnemonics are not provided. The programmer may use the actual operation code or existing mnemonics in *Autocoder* statements that involve these devices. For example:

1. READ FROM CONSOLE PRINTER WITH WORD MARKS, statements:

LABEL	OPERATION	OPERAND	
	LCA	%TO, INPUTB, R	or
CONPR	EQU	%TO	
	LCA	CONPR, INPUTB, R	or
	LR	CONPR, INPUTB	or
	LR	%TO, INPUTB	

produce the actual machine language instruction

L %T0 xxx

2. For SELECT STACKER 9 on Magnetic Character Reader statements:

LABEL	OPERATION	OPERAND	
	SS	L	or
	KL		

produce the actual machine language instruction K L.

3. For ENGAGE optical-character-reader statements:

LABEL	OPERATION	OPERAND	
	CU	%S2, E	or
OPTRD	EQU	%S2	
	CU	OPTRD, E	or
	UE	OPTRD	or
	UE	%S2	

produce the actual machine language instruction U %S2 E.

4. For MOVE CHARACTER TO TRANSMITTING 1009 Data Transmission Unit statements:

LABEL	OPERATION	OPERAND	
	MCW	%D1, INPUTC, W	or
DTUNIT	EQU	%D1	
	MCW	DTUNIT, INPUTC, W	or
	MW	DTUNIT, INPUTC	or
	MW	%D1, INPUTC	

produce the actual machine language instruction M %D1 xxx W.

Processing Overlap

Because the A-address required for an overlap operation (@xx) contains the @ symbol which the 1401 *Autocoder* recognizes as the leftmost end of an alphamerical literal, special coding is required for statements which use the processing overlap feature. To code overlap instructions in *Autocoder*, it is recommended that the programmer use the macro facility of *Autocoder* until mnemonics for these instructions are made available. A typical library routine and macro-instruction to read a tape record in the overlap mode are:

```
LABEL   OPERATION   OPERAND
        RTOXX       3, INPUT
RTOXX   HEADR
        DCW         @M@U ¤01@
        DC          ¤02
        DC          @R@
```

The macro-instruction will cause the machine-language instruction M @ U3 xxx R (where xxx is the equivalent address of INPUT) to be inserted into the object program.

Index

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York

Printed in U.S.A. J24-1434-2 620620MVO