1410 DATA PROCESSING SYSTEM BULLETIN

AUTOCODER: PRELIMINARY SPECIFICATIONS

This bulletin is a minor revision of the previous edition, Form J24-1433-1. It incorporates the amendments published in Technical Newsletters Nos. N28-0015 and N28-0017. No other changes have been made.

Autocoder is an advanced symbolic programming system for the IBM 1410 Data Processing System. It supplements and extends, but does not replace, the basic *Autocoder* for the IBM 1410.

A more powerful language, the IBM 1410 *Autocoder* includes the ability to process macro-instructions, and reduces card handling by using magnetic tape for program manipulation during assembly. The *Autocoder* processor can assemble programs designed to operate on all IBM 1410 systems. The macro-instructions described in the IBM *1410 Input/Output Control System: Preliminary Specifications,* Form J29-1432, can also be used when coding in *Autocoder* language.

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 1410 systems. However, the machine used to assemble a program written in *Autocoder* language must have at least:

20,000 positions of core storage

Four IBM 729 II, 729 IV, or 7330 Magnetic Tape Units

IBM 1403 Printer, Model 2, or listing on tape

IBM 1402 Card Read-Punch or tape input/self-loading tape output

For a completely tape-oriented system, two additional tape units are necessary.

This bulletin contains preliminary language specifications for the IBM 1410 Autocoder. The system tape containing the processor itself, a listing of the processor program, and operating instructions for program assembly, will soon be made available from the IBM Program Applications Library.

Programming with Autocoder

The IBM 1410 *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, adding new constants and work areas as the need arises. The IBM 1410 *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.

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. Follow standard collating sequence for the IBM 1410 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. Use the units position of the line number to indicate the sequence of inserts. Any alphamerical character may 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. See *Address Adjustment.*

Label (Columns 6-15)

Labeling is a method of providing meaningful alphamerical symbols for storage locations, constants, and instructions used in a program. All labels are assigned actual core-storage addresses during the assembly of an object program. When an entry is assigned a label, the programmer can refer to that entry symbolically by putting the label in the operand portion of a subsequent source program statement. Thus, the programmer need not concern himself with actual addresses of data and instructions, but must remember only the symbol which represents that address. Labels should be assigned only if subsequent reference to the items they represent is needed, because unnecessary labels delay the assembly process.

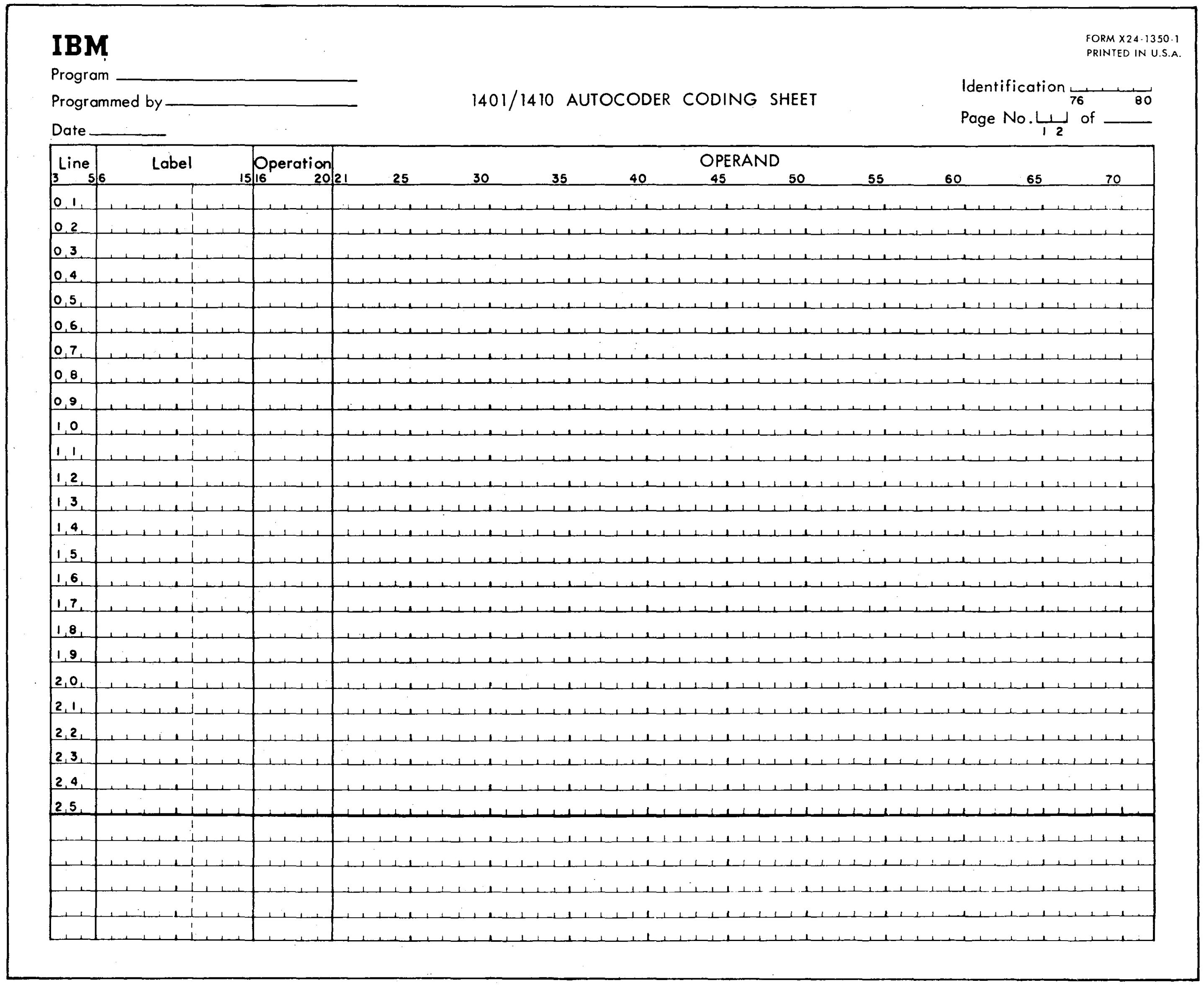
Autocoder labels can be symbolic or actual. A symbolic label can have as many as ten alphamerical characters, but the first character must be alphabetic. Special characters are not permitted in the label field.

Symbolic labels are written left-justified in the label field except as described in DC or DCW.

Actual labels are always written left-justified in the label field. This actual address refers to the high-order position of the instruction, constant, or defined field. Actual labels have no effect on the address assignment counters.

Operation (Columns 16-20)

The operation field contains the mnemonic (easily remembered) operation code for an actual machine-language operation code. Figure 2 shows a complete list of valid mnemonic operation codes and their machine-language equivalents.



IBM

FORM X24-1350-1
PRINTED IN U.S.A.

Program ______

Programmed by ______

Date ______

1401/1410 AUTOCODER CODING SHEET

Identification ______ 76 80

Page No. ______ 1 2 of ______

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

Figure 1. IBM 1401/1410 Autocoder Coding Sheet

Several machine-language operation codes require operation modifiers (d-characters). With a few exceptions, these d-characters are incorporated into *Autocoder* mnemonics and do not have to be coded on the coding sheet. Thus, a single machine-language operation code may have two or more mnemonic equivalents. For example, the machine-language Op code $\check{V}$ *(TEST FOR WORK-MARK OR ZONE AND BRANCH)* has three mnemonic equivalents: BW (BRANCH IF WORD-MARK), BZN (BRANCH IF ZONE), and BWZ (BRANCH IF WORD-MARK AND/OR ZONE).

Operand (Columns 21-72)

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

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 and 4 show typical *Autocoder* entries.

Comments

A comment can be included anywhere in the operand field of an *Autocoder* statement, if at least two spaces separate it from the last character of the operand.

Entire lines of information can be included anywhere in the program 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 identifies a program or program section. This identification number is printed on the output listing but does not appear in the object deck except as described in JOB. The areas labeled *Program, Programmed By,* and *Date* are for the conveniences of the user, but they are never punched.

DECLARATIVE OPERATIONS

MNEMONIC OP CODE	DESCRIPTION
DA	Define Area
DCW	Define Constant with Word mark
DC	Define Constant (no word mark)
DS	Define Symbol
EQU	Equate

IMPERATIVE OPERATIONS

			MACHINE LANGUAGE	
TYPE	MNEMONIC OP CODE	DESCRIPTION	OP CODE	d-CHAR.
Arithmetic	A	Add	A	
	S	Subtract	S	
	ZA	Zero and Add	?	
	ZS	Zero and Subtract	!	
	M	Multiply	@	
	D	Divide	%	

Figure 2. IBM 1410 Mnemonic Operation Codes

IMPERATIVE OPERATIONS

TYPE	MNEMONIC OP CODE	DESCRIPTION	MACHINE LANGUAGE OP CODE	MACHINE LANGUAGE d-CHAR.
Data Control	MRNR	Move left to Right Numerical data; stop at Record mark in A-field	D	Z
	MRZR	Move left to Right Zone data; stop at Record mark in A-field	D	‡
	MLNA	Move right to Left Numerical data; stop at word mark in A-field	D	/
	MLZA	Move right to Left Zone data; stop at word mark in A-field	D	S
	MLCA	Move right to Left whole Characters; stop at word mark in A-field	D	T
	MLWA	Move right to Left Word marks; stop at word mark in A-field	D	U
	MLNWA	Move right to Left Numerical data and Word marks; stop at word mark in A-field	D	V
	MLZWA	Move right to Left Zone data and Word marks; stop at word mark in A-field	D	W
	MLCWA	Move right to Left whole Characters and Word marks; stop at word mark in A-field	D	X
	MLNB	Move right to Left Numerical data; stop at word mark in B-field	D	J
	MLZB	Move right to Left Zone data; stop at word mark in B-field	D	K
	MLCB	Move right to Left whole Characters; stop at word mark in B-field	D	L
	MLWB	Move right to Left Word marks; stop at word mark in B-field	D	M
	MLNWB	Move right to Left Numerical data and Word marks; stop at word mark in B-field	D	N
	MLZWB	Move right to Left Zone data and Word marks; stop at word mark in B-field	D	O
	MLCWB	Move right to Left whole Characters and Word marks; stop at word mark in B-field	D	P
	MRCR	Move left to Right whole Characters; stop at Record mark in A-field	D	, (comma)
	MRWR	Move left to Right Word marks; stop at Record mark in A-field	D	%
	MRNWR	Move left to Right Numerical data and Word marks; stop at Record mark in A-field	D	=
	MRZWR	Move left to Right Zone data and Word marks; stop at Record mark in A-field	D	' (apostrophe)
	MRCWR	Move left to Right whole Characters and Word marks; stop at Record mark in A-field	D	"
	MRNG	Move left to Right Numerical data; stop at Group mark, word mark in A-field	D	R
	MRZG	Move left to Right Zone data; stop at Group mark, word mark in A-field	D	!
	MRCG	Move left to Right whole Characters; stop at Group mark, word mark in A-field	D	$

Figure 2. IBM 1410 Mnemonic Operation Codes (Continued)

TYPE	MNEMONIC OP CODE	IMPERATIVE OPERATIONS DESCRIPTION	MACHINE LANGUAGE OP CODE	d-CHAR.
	MRWG	Move left to Right Word marks; stop at Group mark, word mark in A-field	D	*
	MRNWG	Move left to Right Numerical data and Word marks; stop at Group mark, word mark in A-field	D	)
	MRZWG	Move left to Right Zone data and Word marks; stop at Group mark, word mark in A-field	D	;
	MRCWG	Move left to Right whole Characters and Word marks; stop at Group mark, word mark in A-field	D	Δ
	MRNM	Move left to Right Numerical data; stop at record Mark or group mark, word mark in A-field	D	I
	MRZM	Move left to Right Zone data; stop at record Mark or group mark, word mark in A-field	D	?
	MRCM	Move left to Right whole Characters; stop at record Mark or group mark, word mark in A-field	D	.
	MRWM	Move left to Right Word marks; stop at record Mark or group mark, word mark in A-field	D	⌑
	MRNWM	Move left to Right Numerical data and Word marks; stop at record Mark or group mark, word mark in A-field	D	(
	MRZWM	Move left to Right Zone data and Word marks; stop at record Mark or group mark, word mark in A-field	D	<
	MRCWM	Move left to Right whole Characters and Word marks; stop at record Mark or group mark, word mark in A-field	D	≢
	MRN	Move left to Right Numerical data; stop at word mark in either field	D	9
	MRZ	Move left to Right Zone data; stop at word mark in either field	D	0
	MRC	Move left to Right whole Characters; stop at word mark in either field	D	#
	MRW	Move left to Right Word marks; stop at word mark in either field	D	@
	MRNW	Move left to Right Numerical data and Word marks; stop at word mark in either field	D	:
	MRZW	Move left to Right Zone data and Word marks; stop at word mark in either field	D	>
	MRCW	Move left to Right whole Characters and Word marks; stop at word mark in either field	D	√
	MLN	Move right to Left Numerical data; stop at word mark in either field	D	A
	MLZ	Move right to Left Zone data; stop at word mark in either field	D	B
	MLC	Move right to Left whole Characters; stop at word mark in either field	D	C
	MLW	Move right to Left Word marks; stop at word mark in either field	D	D
	MLNW	Move right to Left Numerical data and Word marks; stop at word mark in either field	D	E
	MLZW	Move right to Left Zone data and Word marks; stop at word mark in either field	D	F
	MLCW	Move right to Left whole Characters and Word marks; stop at word mark in either field	D	G

Figure 2. IBM 1410 Mnemonic Operation Codes (Continued)

IMPERATIVE OPERATIONS				
TYPE	MNEMONIC OP CODE	DESCRIPTION	MACHINE LANGUAGE OP CODE	d-CHAR.
	MLNS	Move right to Left a Single position of Numerical data	D	1
	MLZS	Move right to Left a Single position of Zone data	D	2
	MLCS	Move right to Left a Single whole Character	D	3
	MLWS	Move right to Left a Single Word mark	D	4
	MLNWS	Move right to Left a Single position of Numerical data and Word mark	D	5
	MLZWS	Move right to Left a Single position of Zone data and Word mark	D	6
	MLCWS	Move right to Left a Single whole Character and Word mark	D	7
	SCNRR	Scan Right for Record mark in A-field	D	Y
	SCNRG	Scan Right for Group mark, word mark in A-field	D	Q
	SCNRM	Scan Right for record Mark or group mark, word mark in A-field	D	H
	SCNR	Scan Right for word mark in either field	D	8
	SCNLA	Scan Left for word mark in A-field	D	¢
	SCNLB	Scan Left for word mark in B-field	D	—
	SCNL	Scan Left for word mark in either field	D	&
	SCNLS	Scan Left a Single position	D	(blank)
	MCS	Move Characters and Suppress zeros	Z	
	MCE	Move Characters and Edit	E	
	C	Compare	C	
	LL	Lookup Low	T	1
	LE	Lookup Equal	T	2
	LLE	Lookup Low or Equal	T	3
	LH	Lookup High	T	4
	LLH	Lookup Low or High	T	5
	LEH	Lookup Equal or High	T	6
Logical	BW	Branch if Word mark	V	1
Operations	BZN	Branch if Zone	V	2
	BWZ	Branch if Word mark or Zone	V	3
	BCE	Branch if Character Equal	B	char.
	BBE	Branch if Bit Equal	W	char.
	B	Branch unconditional	J	(blank)
	BC9	Branch Carriage channel 9	J	9
	BCV	Branch Carriage overflow	J	@
	BU	Branch Unequal	J	/
	BE	Branch Equal	J	S
	BL	Branch Low	J	T
	BH	Branch High	J	U
	BPCB	Branch Printer Carriage Busy	J	R
	BZ	Branch Zero result	J	V
	BAV	Branch Arithmetic overflow	J	Z

Figure 2. IBM 1410 Mnemonic Operation Codes (Continued)

TYPE	MNEMONIC OP CODE	IMPERATIVE OPERATIONS DESCRIPTION	MACHINE LANGUAGE OP CODE	d-CHAR.
	BDV	Branch Divide overflow	J	W
	BNQ	Branch inquiry	J	Q
	†BEX1	Branch on External indicator — channel 1	R	d
	†BEX2	Branch on External indicator — channel 2	X	d
	BA1	Branch Any external indicator — channel 1	R	⧧
	BA2	Branch Any external indicator — channel 2	X	⧧
	BNR1	Branch I/O Not Ready — channel 1	R	1
	BNR2	Branch I/O Not Ready — channel 2	X	1
	BCB1	Branch I/O to Channel Busy — channel 1	R	2
	BCB2	Branch I/O to Channel Busy — channel 2	X	2
	BEF1	Branch I/O to End-of-File — channel 1	R	8
	BEF2	Branch I/O to End-of-File — channel 2	X	8
	BNT1	Branch No Transfer — channel 1	R	¢
	BNT2	Branch No Transfer — channel 2	X	¢
	BWL1	Branch Wrong Length record — channel 1	R	—
	BWL2	Branch Wrong Length record — channel 2	X	—
	BER1	Branch Error — channel 1	R	4
	BER2	Branch Error —channel 2	X	4
	BOL1	Branch Overlap in process — channel 1	J	1
	BOL2	Branch Overlap in process — channel 2	J	2
	BRC1	Branch Read back Check — channel 1	R	@
	BRC2	Branch Read back Check — channel 2	X	@
Tape Utility Operations	BSP	Backspace Tape	U	B
	SKP	Skip and blank tape	U	E
	WTM	Write Tape Mark	U	M
	RWD	Rewind tape	U	R
	RWU	Rewind Unload	U	U
	†CU	Control Unit	U	d
Priority Operations	BXPA	Branch to Exit Priority Alert	Y	X
	BEPA	Branch to Enter Priority Alert	Y	E
	BSPR1	Branch if Seek Priority Request — channel 1	Y	S
	BSPR2	Branch if Seek Priority Request — channel 2	Y	T
	BOPR1	Branch if Overlap Priority Request — channel 1	Y	1
	BOPR2	Branch if Overlap Priority Request — channel 2	Y	2
	BIPR	Branch if Inquiry Priority Request	Y	Q
	BUPR	Branch if Unit Record Priority Request	Y	U
Miscel- laneous	SAR	Store A-address Register	G	A
	SBR	Store B-address Register	G	B
	SER	Store E-address Register	G	E
	SFR	Store F-address Register	G	F
	SW	Set Word mark	,	
	CW	Clear Word mark	⌑	
	CS	Clear Storage, Clear Storage and branch	/	
	H	Halt, Halt and branch	.	
	NOP	No Operation	N	

Figure 2. IBM 1410 Mnemonic Operation Codes (Continued)

TYPE	MNEMONIC OP CODE	IMPERATIVE OPERATIONS DESCRIPTION	MACHINE LANGUAGE OP CODE	d-CHAR
	NOPWM	No Operation Word Mark	N	
	†CC	Control Carriage	F	d (Forms control character)

TYPE	MNEMONIC OP CODE	DESCRIPTION	MACHINE LANGUAGE X-ADDR	d-CHAR.
I/O	R	Read card	%1n	R
Commands	P	Punch card (n-character must appear in operand field of Autocoder instruction)	%4n (n denotes selected pocket)	W
	W	Write a line	%20	W
	RCP	Read Console Printer	%T0	R
	WCP	Write Console Printer	%T0	W
	WM	Write word Marks	%21	W
	RD	Read Disk single record	%F1	R
	RDT	Read Disk full Track	%F2	R
	WD	Write Disk single record	%F1	W
	WDC	Write Disk Check	%F3	W
	WDT	Write Disk full Track	%F2	W
	RT	Read Tape	%Un	R
	WT	Write Tape	%Un	W
	RTB	Read Tape Binary	%Bn	R
	WTB	Write Tape Binary	%Bn	W
	NOTE: The preceding mnemonic Op Codes, except WM, may be followed by the letter W to indicate transfer of word marks and/or followed by the letter O, including WM, to indicate overlapping.			
	† MU	Move Unit — Actual X-ADDR. and d-CHAR. must be coded in Autocoder instruction.	x x x	d
	† LU	Load Unit	x x x	d
	RTG	Read Tape; stop only at interrecord Gap	%Un	$
	RTBG	Read Tape Binary; stop only at interrecord Gap	%Bn	$
	WTE	Write Tape; stop at End of core	%Un	X
	WTBE	Write Tape Binary; stop at End of core	%Bn	X
	SD	Seek Disk	%F0	R
	NOTE: Only the SD Mnemonic Op Code may be followed by the letter O to indicate overlapping. RTG, RTBG, WTE, WTBE can be followed by the letter W.			
	† SSF	Select Stacker and Feed card (d-character must appear in operand field of Autocoder instruction)	K (OP CODE)	b, 1, 2 (denotes selected pocket)

CONTROL OPERATIONS

MNEMONIC OP CODE	DESCRIPTION	MNEMONIC OP CODE	DESCRIPTION
CTL	Control	SFX	Suffix
ORG	Origin	RUN	Run
LTORG	Literal Origin	RESEQ	Re-sequence
EX	Execute	LOAD	Load
END	End	EJECT	Eject
PST	Print Symbol Table	INSER	Insert
JOB	Job Description	DELET	Delete

†d-character must appear in operand field of Autocoder instruction

Figure 2. IBM 1410 Mnemonic Operation Codes (Continued)

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 valid A- and/or B-addresses are supplied by the chaining method. For example,

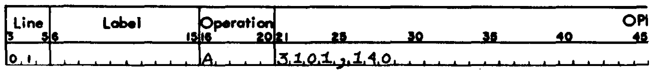
```
MLCA  A, B
MLCA
```

NOTE: If an instruction is to have addresses stored by other instructions, the operand or operands affected must not be left blank. For example, $\overset{\vee}{B}$ 0 is recommended if the address of the branch instruction is to be supplied during the running of the object program.

Actual

The actual core-storage address of a data field 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.

Figure 3 shows an imperative instruction that causes the contents of core-storage location 3101 to be added algebraically to the contents of location 140. This entry will be assembled as a machine-language instruction: $\overset{\vee}{A}$ 03101 00140. Note that high-order zeros can be eliminated when coding actual addresses for *Autocoder*.



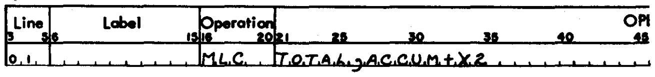
Line 3 5	Label 6 15	Operation 16 20	OPI 21 25 30 35 40 45
0 1		A	3101,140

Figure 3. *Autocoder* Instruction with Actual Addresses

Symbolic

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

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 may be followed by a plus sign (+), an X to indicate indexing, and a number from 1 to 15 to specify which index location is to be used. TOTAL is the label for locations 3 1 0 1 and ACCUM is the label for location 1 4 0. The assembled machine-language instruction for this entry is: $\overset{\vee}{D}$ 03101 001M0 C. 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.



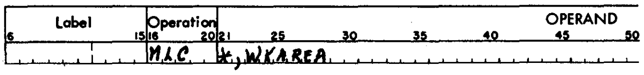
Line 3 5	Label 6 15	Operation 16 20	OPI 21 25 30 35 40 45
0 1		MLC	TOTAL,ACCUM+X2

Figure 4. *Autocoder* Instruction with Symbolic Addresses

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 5 is assigned core storage locations 00340-00351. The actual address of WKAREA is 00598. The assembled instruction is $\overset{\vee}{D}$ 00351 00598 C. When the instruction is executed in the object program, $\overset{\vee}{D}$ 00351 00598 C will be placed in WKAREA.

Asterisk operands can have address adjustment and indexing.



Label 6 15	Operation 16 20	OPERAND 21 25 30 35 40 45 50
	MLC	*,WKAREA

Figure 5. Asterisk Operand in *Autocoder* Instruction

Literals

The IBM 1410 *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. Literals are permitted only in the operand field of an *Autocoder* statement and can be numerical or alphamerical. A literal can be up to 52 characters in length, including the sign; i. e., it must be contained in one line of the coding sheet, and it must not extend beyond column 72. Literal addresses may make use of address-adjustment and/or indexing.

Types of Literals

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. When a numerical literal does not exceed nine digits plus sign (blanks are not allowed), it is assigned a storage location only once per program or *program section*, no matter how many times it appears in the source program or program section. 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*.

Figure 6 shows how a numerical literal can be used in an imperative instruction. Assume the literal (+ 10) is assigned a storage location of 00584 and 00585 and INDEX is assigned 00682. The symbolic instruction will cause the processor to produce a machine-language instruction (Ǎ 00585 00682) that causes + 10 to be added to the contents of INDEX.

Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 OPI 45
0 1		A	+10,INDEX

Figure 6. 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.

   Upon encountering an alphamerical literal, the processor proceeds to column 72 of the card and searches right to left for the terminal @ symbol. If it encounters any @ symbol, it will assume this is the legitimate terminal.

2. An alphamerical literal of from one to nine 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 cases where multiple use of long literals is necessary, use a DCW statement.

Figure 7 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 00906, and DATE is assigned 00230. The machine-language instruction (Ď 00906 00230 C) causes the literal JANUARY 28, 1961 to be moved to DATE.

Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 OPI 45
0 1		MLC	@JANUARY 28, 1961@,DATE

Figure 7. Alphamerical Literal

AREA-DEFINING LITERAL

The 1410 *Autocoder* allows the user to define an area to be reserved by placing an area-defining literal in the operand field of a symbolic program entry as follows:

1. An area of any size may be defined in any instruction which has as an operand the symbol which references it; for example, WKAREA#2.

2. A # symbol (8-3 punch) must precede the number that specifies how many core-storage locations are needed for the work area. Note that the # symbol is represented in the Fortran character set as an = symbol.

3. A word mark is placed over the high-order position of the area.

4. If the user refers to a portion of the same defined area, such as WKAREA#6, he will be given a multiple definition flag in his output listing.

5. Address adjustment and indexing are permitted when using area-defining literals.

Figure 8 shows an imperative instruction with an area-defining literal. This entry causes the processor to allocate six storage locations for WKAREA. Six blanks will be loaded in storage at object program load time by a DCW card automatically produced by the processor. Assuming that AMOUNT is in storage location 00796 and WKAREA is in 00596, the assembled machine-language instruction that moves AMOUNT to WKAREA is Ď 00796 00596 C.

Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 OPI 45
0 1		MLC	AMOUNT,WKAREA#6

Figure 8. Area-Defining Literal

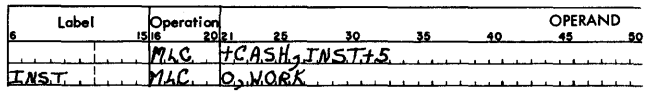
ADDRESS CONSTANT LITERALS

The actual 5-character machine address which is assigned to a label by the processor can be defined as an *address constant*. *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 ten characters.
2. A plus sign must precede the symbol. The address constant is the actual address which was assigned to the label by the processor.
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. The address constant literal is unsigned in core storage.

NOTE: If address adjustment and indexing occur, they modify the address of the literal, not the literal itself.

Figure 9 shows how an address constant literal can be used. Assume that CASH is used as a label elsewhere in the program and has been assigned a machine address of 00600. The address constant (00600) has been assigned storage location 00797. The first character in the second instruction is in core storage at address 00401. Thus, the address of INST + 5 is 00406.



Label	Operation	OPERAND
	MLC	+CASH,INST+5
INST	MLC	0,WORK

Figure 9. Address Constant

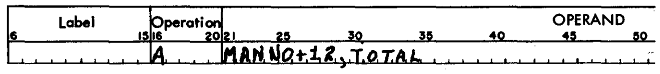
The assembled machine-language instruction for the first symbolic instruction in Figure 9 is $\overset{\vee}{D}$ 00797 00406 C.

WORK is in storage location 00729. The assembled machine-language instruction for the second symbolic program entry is $\overset{\vee}{D}$ 00000 00729 C. When the first instruction is executed in the object program, the constant 00600 is moved to 00406 and the second instruction becomes $\overset{\vee}{D}$ 00600 00729 C. 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.

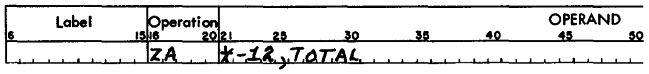
Address Adjustment

Address adjustment is valid in the operand field on all symbolic addresses, including the asterisk. It enables the programmer to refer to an entry in his source program that is a specified number of locations away from a symbolic address. Its usage reduces the number of symbolic labels required. Address adjustment is indicated by writing after the symbolic address a plus or minus sign followed by one to five digits (Figure 10).



Label	Operation	OPERAND
	A	MANNO+12,TOTAL

Figure 10. Address Adjustment



Label	Operation	OPERAND
	ZA	*-12,TOTAL

Figure 11. Address Adjustment with an Asterisk Operand

When the label MANNO is assigned location 05000 and TOTAL is assigned the location 00075, the assembled instruction is $\overset{\vee}{A}$ 05012 00075.

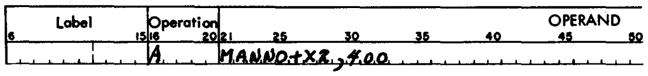
If the instruction in Figure 11 is assigned the address 05000, the assembled instruction is $\overset{\vee}{?}$ 04998 00075, because * refers to the rightmost position of the instruction (05010). When using address adjustment, the programmer should remember that insertions or deletions in the source program can affect adjustment addresses.

Index Registers

Indexing is accomplished by tagging an address in the operand field with an indicator telling the processor which index register is to be used. The IBM 1410 system has 15 index registers that can be referred to in *Autocoder* language by placing an X before their number. Thus, X10 denotes index register 10. The X enables the processor to distinguish between address adjustment and indexing.

An index register can also be referred to symbolically. X0 through X15 are not acceptable as symbolic names. The index label must be preceded by a plus. It follows the operand address and the address adjustment, if any. Figure 12 shows an example of indexing.

The contents of the location with the address, MANNO *plus the contents of index register 2,* is algebraically added to the contents of location 00400. For example, if the label MANNO is assigned location 05000 and index register 2 contains 500, then the preceding instruction causes the contents of location 05500 to be added to the contents of location 00400.



Label	Operation	OPERAND
	A	MANNO+X2,400

Figure 12. Indexing

Indexing is not acceptable in DS, ORG, or LTORG declarative operations nor control operations.

An index register can be specified in the operand field for other than indexing purposes. For example, a numerical value can be added to the contents of an index register. In this case, the index register may be referred to by its actual label (X1, X2, etc.) or its symbolic label (see *EQU*).

Index Register Reservation

The processor assigns index registers referred to in the symbolic program.

Those index registers that are coded in actual notation (X1, X2, etc.) and those equated to a symbolic address by an EQU statement are assigned first. Then the remaining index registers are assigned to symbols the programmer has used to represent index registers. For example, the programmer may use the symbolic instruction shown in Figure 13.

In this case, CONST is the symbolic label for an index register. Its contents will modify the address assigned to the label (WHTAX). The instruction in Figure 13 may be followed by the instruction shown in Figure 14. This instruction puts the numerical value 25 in the index register which the processor assigns to CONST.

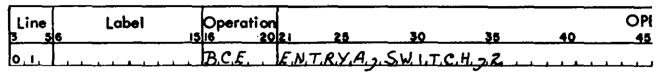
Autocoder Coding Examples

Figure 15 shows an imperative instruction with address adjustment and indexing on a symbolic address. The processor will subtract 12 from the address assigned the label TOTAL. The effective address of the A-operand is the sum of TOTAL −12 plus the contents of index location 1. The assembled instruction $\overset{\vee}{\text{D}}$ 030Y9 00140 C 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 overpunch is a tag for index location 1.

NOTE: Address adjustment and indexing are permitted in the same operand. Multiple address adjustment causes the algebraic sum of the factors to be used. With multiple indexing, only the rightmost index notation is effective. For example:

A TOTAL +3 +X1 −12 +X2, ACCUM −5 +X2 +35
will be interpreted as:
A TOTAL −9 +X2, ACCUM +30 +X2
which is equivalent to:
A TOTAL +X2 −9, ACCUM +X2 +30

Figure 16 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: $\overset{\vee}{\text{B}}$ 00392 00498 2. It branches to ENTRY A (00392) if the location labeled SWITCH contains a 2.



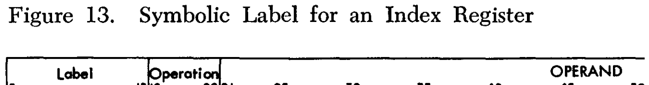
Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
01		BCE	ENTRYA,SWITCH,2

Figure 16. *Autocoder* Instruction with a d-Character

Declarative Operations

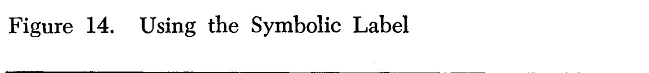
A program for the 1410 usually requires the use of *work areas* and *constants.* A work area is a portion of storage into which data is transferred for processing. It can be used for the accumulation of totals or for the assembling of data to be printed out or punched into cards. A constant is a fixed quantity or item of information that is required again and again or that must remain the same throughout the course of the program. For example, a date can be considered a constant.

The use of *Autocoder* enables the programmer to refer to work areas and constants by their descriptive names without regard to their actual location in core storage. For example, assume that the programmer wants to reserve twenty consecutive core locations for accumulating a final sales total. A declarative operation enables the programmer to reserve such an area and to refer to it by a symbolic label without concerning himself with the actual address of the field.



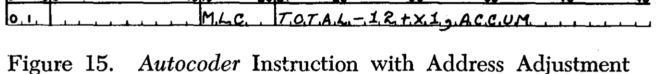
Label	Operation	OPERAND
6 15	16 20	21 25 30 35 40 45 50
	A	WHTAX+CONST,YTDTAX

Figure 13. Symbolic Label for an Index Register



Label	Operation	OPERAND
6 15	16 20	21 25 30 35 40 45 50
	ZA	+25,CONST

Figure 14. Using the Symbolic Label



Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
01		MLC	TOTAL-12+X1,ACCUM

Figure 15. *Autocoder* Instruction with Address Adjustment and Indexing

Declarative operations are definitions rather than instructions. As such they are acted upon during assembly but are not executed during the running of the object program. For this reason the programmer should keep declaratives separate from imperatives (machine instructions) when writing the symbolic program. If they are placed in the body of the program, care must be taken to branch around them so they will not be treated as instructions.

The IBM 1410 *Autocoder* provides five 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
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. Symbolic labels address the low-order position of the constant. Word marks are set in the high-order positions of all constants. If a symbolic label is indented one position, the address of the high-order position of the constant will be assigned to the symbol. Actual labels always refer to the high-order position of the defined constant.

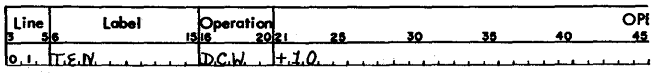
The programmer:

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 17 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.



Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
0 1	TEN	DCW	+10

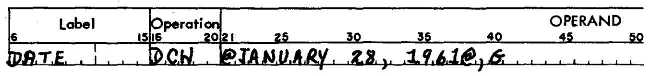
Figure 17. Numerical Constant Defined in a DCW Statement

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.
3. If no terminal @ is present, a 51-character constant will be produced.

Example: Figure 18 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.

NOTE: A comma G following the trailing @ symbol of an alphamerical constant causes the processor to put a group-mark word-mark in storage following the last character of the constant. The associated label, if any, will refer to the last character of the constant, not the group-mark word-mark.



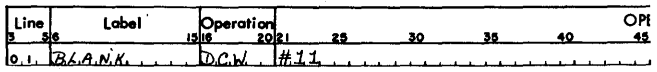
Label	Operation	OPERAND
6 15	16 20	21 25 30 35 40 45 50
DATE	DCW	@JANUARY 28, 1961@,G

Figure 18. Alphamerical Constant Defined in 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. Maximum size of this field is limited only by the available storage capacity.

Example: Figure 19 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.



Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
0 1	BLANK	DCW	#11

Figure 19. Blank Constant Defined by a DCW Statement

ADDRESS CONSTANTS

An address constant can be preceded by a plus or a minus sign, or it can be left unsigned. The constant is the actual machine-language address of the field whose associated label is included in the operand. The units position of the constant will have the sign which the user placed before the operand.

NOTE: Address constants may be address adjusted and indexed.

Example: Figure 20 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.

Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
0 1	SERIAL	DCW	+MANNO

Figure 20. Address Constant Defined by a DCW Statement

DC — Define Constant (No Word Mark)

General Description: 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 produced in the object deck.

DS — Define Symbol

General Description: A DS statement reserves 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.
3. Writes a number in the operand field to indicate how many storage positions are to be reserved.

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

Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
0 1	ACCUM	DS	10

Figure 21. DS Statement

DA — Define Area

General Description: DA statements reserve and define portions of core storage, such as input or 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.

HEADER LINE

The programmer constructs a header line for the DA entry as follows:

1. Writes the operation code (DA) in the operation field.
2. 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.
3. 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 22. If only one area is to be defined, the first entry is 1 X 100.

Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
0 1	TAPEAR	DA	4X100

Figure 22. Four Areas Defined

Indexing: To index a DA statement place a comma and the number of the index location (X1, X2, X3, etc.) after the B X L indication. All labels in the entries following the header line will be indexed by the specified index register when they appear in instructions, *unless the instruction referring to the field is itself indexed.* For example, if IN AREA is defined by the statement shown in Figure 23, ACCUM is indexed by index location 1. If the entry shown in Figure 24 appears as an instruction elsewhere in the program, ACCUM (for this instruction only) will be indexed by

Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
01	INAREA	DA	3X60,X1
02	ACCUM		35,40

Figure 23. Indexing a DA Statement

the contents of index location 2. Because the instruction in Figure 24 has indexing, this indexing overrides the indexing prescribed by the 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. Symbolic names for index registers may be specified in the heading line of a DA statement only if previously defined by an EQU statement.

Record Marks: 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 comma ‡ 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.

Relative to Zero Addressing: By writing a comma zero after the B X L entry, the user can cause the processor to assign addresses to the labels of fields and sub-fields as though the high-order position of the defined area was core-storage location zero. The label of the DA statement is assigned the address of the high-order position of the area actually reserved by the processor.

NOTES

1. A user of 1410 IOCS must define areas to be used for blocked records using indexing, with relative to zero addressing.
2. The programmer may write the ‡, index code, G and 0 entries in any order in the operand field of the DA header statement provided that they follow the B X L entry.

OTHER DA ENTRIES

The programmer constructs the balance of the DA statement which defines fields and subfields for each area as follows:

1. Leaves the operation field blank.

Line	Label	Operation	OPI
3 5	6 15	16 20	21 25 30 35 40 45
01		ZA	GROSS,ACCUM+X2

Figure 24. Overriding Indexing in a DA Statement

2. Writes a symbolic label in the label field if one is desired.
3. Specifies the relative location of defined fields within the area by putting two numbers in the operand field. 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.
4. 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.

NOTES: 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 defined fields and subfields in the place of symbols corresponding to the labels of the field-defining entries.

Result: At object program load time:

1. Word marks are set for field definition as noted previously.
2. 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

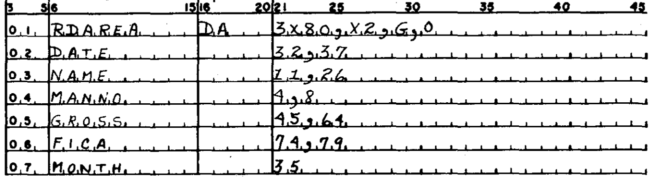
Remaining positions contain data not used in this operation. Positions 34 and 35, which indicate the month within the date, will be defined as a subfield. A group mark with a word mark is to be placed in storage immediately following the third area.

The DA statement in Figure 25 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. 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 NAME, MANNO, DATE, GROSS, FICA and MONTH, when referred to in symbolic instructions, will be indexed by index location 2.

The IOCS will give an instruction to read data from tape into a storage area labeled RDAREA. 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 IOCS begins by setting index location 2 to the address of the input area. The user then processes the significant data in the first record. The subsequent GET macro will increment index location 2 by eighty, and the user can branch 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.

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 an individual area, 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.



Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 OPE 45
01	RDAREA	DA	3X80,X2,G,0
02	DATE		32,37
03	NAME		11,26
04	MANNO		4,8
05	GROSS		45,64
06	FICA		74,79
07	MONTH		35

Figure 25. DA Statement

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. May write 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:

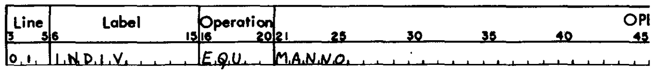
1. Assigns to the label of the equate statement the same actual address that is assigned to the symbol in the operand field (with appropriate alteration if indexing and address adjustment are indicated).
2. Inserts this actual address wherever the label appears as the operand of another symbolic program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

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

Examples: Figure 26 shows the label INDIV equated to MANNO which has been assigned storage location 01976. Whenever either MANNO or INDIV appear in a symbolic program, 01976 will be used as the actual address.

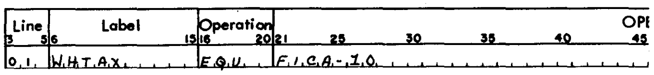
Figure 27 shows an equate statement with address adjustment. If FICA is assigned location 00890, WHTAX will be equated to FICA—10 (00880). WHTAX now refers to a field whose units position is 00880.

Figure 28 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 01076-01080 contain net pay



Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 OPE 45
01	INDIV	EQU	MANNO

Figure 26. EQU Statement



Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 OPE 45
01	WHTAX	EQU	FICA-10

Figure 27. Address Adjustment in an EQU Statement

(if the read area is 01001-01080). This field can be referred to as NETPAY if the EQU statement in Figure 28 is written in the source program.

Figure 29 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 29, 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 appear 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, and index register 3 is ignored.

Figure 30 shows the symbol FIELDA equated to an asterisk address. The asterisk refers to the current position of the processor assignment counter. (This will be the first position of the instruction or data to be next assigned.) Assume that this address is 00698. FIELDA is now equal to 00698.

Figure 31 shows how a label can be assigned to an index location. The operand contains a number from 1 to 15, followed by a comma, followed by the letter X to indicate the specific index register. INDEX 1 is now equal to 00029. Figure 31 also shows an alternative method for equating a label to an index register.

Figure 32 shows how a tape unit can be assigned a label. In this case, the programmer wishes to refer to tape 4 on channel 1 as INPUT.

A tape unit may also be equated to a symbolic name by using the actual X-control field (for example % U 4) as the operand, as shown in Figure 33.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
NETPAY	EQU	1080

Figure 28. Assigning a Label to an Actual Address

Line (3–5)	Label (6–15)	Operation (16–20)	OP (21–45)
01	CUSTNO	EQU	JOB+X3

Figure 29. Indexing an EQU Statement

Line (3–5)	Label (6–15)	Operation (16–20)	OP (21–45)
01	FIELDA	EQU	*

Figure 30. Equating with an * Operand

Label (6–15)	Operation (16–20)	OPERAND (21–50)
INDEX1	EQU	1,X
INDEX1	EQU	X1

Figure 31. Assigning a Label to an Index Location

Label (6–15)	Operation (16–20)	OPERAND (21–50)
INPUT	EQU	14,CU

Figure 32. Assigning a Label to a Tape Unit

Label (6–15)	Operation (16–20)	OPERAND (21–50)
OUTPUT	EQU	%U4
	RT	OUTPUT,AREAB

Figure 33. Actual X-Control Field

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 remember 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 should 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 41).
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 an I-address if the B-address is not needed.

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 †.

Coding

Figure 34 shows a brief routine illustrating a section of *Autocoder* coding. Note that remarks can appear anywhere in the operand field, provided at least two blank spaces separate the remarks from the operand of the instruction.

Several imperative operations are governed by special rules, and care must be taken when coding with these instructions. The special cases are described in the following paragraphs.

Data-Move Instructions

The data-move command is controlled in machine language by the Op code $\overset{\vee}{D}$. The actual conditions of the various types of data move instructions are regulated by the d-character. To make the *Autocoder* language more meaningful, each of these move and scan instructions has a different mnemonic op code. Each of these mnemonics specifies the type of operation, the direction of the move or scan, the nature of the data to be moved, and what terminates the operation. The following rules apply in constructing the mnemonics for data-move commands:

Move Mnemonics

1. The first character of the mnemonic is M.
2. The second character specifies the direction of data movement.
   - L — Right-to-left movement.
   - R — Left-to-right movement.

Label (6–15)	Operation (16–20)	OPERAND (21–50)
181	DC	@‡@
233	DCW	@‡@
START	SW	181
	R	1,101
	BA1	ERROR
	BZN	PRINT,180,B
	P	4,101
	BA1	ERROR
	B	START
PRINT	CW	181
	W	101
	BA1	ERROR
	B	START
	END	START

Figure 34. Autocoder Coding

3. The third section of the mnemonic specifies the portion of data moved.
   - N — Move numerical portion of data.
   - Z — Move zone portion of data.
   - C — Move whole characters.
   - W — Move word marks.
   - NW — Move numerical portion and word marks.
   - ZW — Move zone portion and word marks.
   - CW — Move whole characters and word marks.
4. The final character of the mnemonic specifies what terminates the move.
   a. To terminate right-to-left move:
      - A — Word mark in A-field.
      - B — Word mark in B-field.
      - (Blank)— Word mark in either field.
      - S — Move single location only.

   b. To terminate left-to-right move:
      - R — Record mark in A-field.
      - G — Group mark with a word mark in A-field.
      - M — Record mark or group mark with a word mark in A-field.
      - (Blank) — Word mark in either field.

Scan Mnemonics

1. The first three characters are always SCN.
2. The fourth character specifies the direction of scan.
   - L — Right-to-left scan.
   - R — Left-to-right scan.
3. The fifth character specifies what terminates the scan.
   a. To terminate right-to-left scan:
      - A — Word mark in A-field.
      - B — Word mark in B-field.
      - (Blank) — Word mark in either field.
      - S — Scan left single position.

   b. To terminate left-to-right scan:
      - R — Record mark in A-field.
      - G — Group mark with a word mark in A-field.
      - M — Record mark or group mark with a word mark in A-field.
      - (Blank) — Word mark in A- or B-field.

For example, when whole characters and word marks are to be moved from right to left, terminating the move on a word mark in the A-field, the *Autocoder* mnemonic op code is MLCWA.

A complete list of these data move mnemonics is included in the *List of 1410 Autocoder Mnemonic Operation Codes* (see Figure 2).

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	WT	14,OUTPUT

Figure 35. Write Tape

SSF — Select Stacker and Feed

This instruction causes the last card transferred to storage to be selected into the stacker specified in the operand field of the instruction. A blank operand causes the card to be selected into the zero read pocket. A 1 in the operand field causes the card to be selected into stacker 1. A 2 in the operand field causes the card to be selected into stacker 8/2.

Magnetic Tape Commands

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 and channel needed for the operation. This can be done in one of three ways.

The programmer can:

1. Assign a label to the channel and tape unit as described in EQU and use it as the A-operand of a tape instruction, or
2. Write the number of the channel and tape unit in columns 21 and 22 of the tape instruction. The assembled instruction for the symbolic entry shown in Figure 35 will cause a record to be written on tape unit 4 using the data beginning in a storage area labeled OUTPUT, or
3. Write the X-control field as the A-operand of the tape instruction.

Disk Commands

All input-output commands involving disk units must specify the channel (1 or 2) as the first entry of the operand field. If an address is used in the operand, it follows the channel designation and is separated from it by a comma as shown in Figure 36.

BZN — Branch on Zone

The operand of this command takes the form I ADDR, Ch where:

Ch = Zone configuration to be tested for (A, B, AB, blank, —, +, or ¢)
ADDR = Address of character whose zone is to be tested.
I = Address to branch to

ADDR or both addresses can be omitted if this operation is chained. Acceptable forms of this operation are:

```
BZN  I, ADDR, Ch
BZN  I
BZN
```

Label 6–15	Operation 16–20	OPERAND 21 25 30 35 40 45 50
	SD	1, SEEKADDRES

Figure 36. Disk Storage Instruction

BWZ — Branch if Word Mark, Zone, or Both

This operation is the same as BZN except that a branch also takes place if a word mark is present.

BCE — Branch if Character Equal

The operand of this command takes the form I, ADDR, Ch where:

Ch = Character to be matched
ADDR = Address of character to be compared
I = Address to be branched to
Permissible forms of these operations are:

```
BCE  I, ADDR, Ch
BCE  I
BCE
```

BBE — Branch if Bit Equal

The operand of this command will take the form I, ADDR, Ch where:

Ch = Character containing bit(s) to be tested for
ADDR = Address of character to be tested
I = Address to be branched to
Permissible forms of this operation are:

```
BBE  I, ADDR, Ch
BBE  I
BBE
```

CC — Control Carriage

The forms control character must be written in the operand field of this instruction. Standard forms control characters are to be used.

P — Punch

The pocket into which the punched card will be selected must be specified as the first entry of the operand field of this instruction. The address from which data will be punched is specified following the stacker specifications and is separated from it by a comma. A 0-punch selects punched cards into stacker pocket 0; a 4-punch selects punched cards into stacker pocket 4; an 8-punch selects cards into stacker pocket 8/2.

R — Read

A read command must have as the first entry in its operand either the number of the stacker into which the card is to be selected after reading, or an indication that a select stacker command will follow the read command. A 0-punch selects cards into stacker pocket 0; a 1-punch, into stacker pocket 1; and a 2-punch, into stacker pocket 8/2. A 9-punch indicates that a select stacker command will follow. The address (symbolic or actual) of the storage area into which the data from the card is to be read must be the second entry in the operand of a read command.

Input-Output Commands

All I/O mnemonic op codes pertaining to unit record equipment (card punch, card reader, IBM 1403 Printer, console I/O Printer) can be followed by the letter W to indicate transfer of word marks. An O, if present, indicates overlapping. If both W and O are required, the W must precede the O in the instruction.

The instructions thus affected are R, P, W, RCP, WCP, and WM.

All I/O mnemonic op codes pertaining to magnetic tape-and-disk storage except SD, RTG, RTBG, WTE, and WTBE may end in the letter W to indicate that word marks will be transferred. Also, all the I/O mnemonics except RTG, RTGB, WTE, and WTBE can be followed by the letter O to indicate overlapping.

Priority Processing

IBM 1410 Data Processing Systems equipped with the priority-processing feature can process I/O no-op commands. To code these in *Autocoder* language, write an N as the first character of the I/O mnemonic. For example, the instruction shown in Figure 37 will be assembled as $\overset{\vee}{\text{M}}$ %U1 00100 Q.

The instruction shown in Figure 38 will be assembled as $\overset{\vee}{\text{M}}$ %U1 00100 V.

The I/O no-op instruction will set the appropriate I/O external indicators, but no data movement takes place.

NOTE: Like any other I/O instruction, the I/O no-op instruction *always* sets the I/O interlock latch ON. This latch must be set OFF before another I/O instruction can be executed on the same channel.

The I/O interlock latch can be set OFF by one of the following *classes* of instructions:

1. Branch Any External Indicator—Channel (1 or 2) (i. e., BA1 or BA2)
   or
2. Branch on External Indicator—Channel (1 or 2) (i. e., BEX1 d or BEX2 d, where d ≠ ‡), *provided the branch is executed.* (If the branch is *not* executed, the I/O interlock latch is *not* turned OFF.)

NOPWM — No Operation Word Mark

The 1410 *Autocoder* permits the programmer to set programmed no-op switches easily. If the statement shown in Figure 39 is written in the source program, the processor will insert in the object program the operation code $\overset{\vee}{\text{N}}$ (no-op) with a word mark, followed by the branch instruction (B XXXXX) without a word mark in the operation code position. Subsequent instructions in the object program can then be used to set and clear the word mark in the operation code position of the branch instruction as needed. If there

Label 6–15	Operation 16–20	OPERAND 21–50
	NRT	11,100

Figure 37. I/O No-Op Input Command

Label 6–15	Operation 16–20	OPERAND 21–50
	NWT	11,100

Figure 38. I/O No-Op Output Command

Label 6–15	Operation 16–20	OPERAND 21–50
	NOPWM	
	B	START

Figure 39. No-Op Word Mark

is no word mark, the branch instruction will be ignored, and if the word mark is present, the branch instruction will be executed. The assembled instructions produced by the entries shown in Figure 39 are $\overset{\vee}{\text{N}}$ B 00500 (assuming start is in location 500).

BEX1 or BEX2 — Branch on External Indicator

These mnemonics are used for the machine-language op codes R and X. BEX1 is equal to R; BEX2 is equal to X. One of the two, depending on channel, is used when testing for a combination of external indicator conditions for which there is no mnemonic. The symbolic operand must take the form ADDR, d where:

ADDR =	Address to be branched to, if any of the external indicators specified in the d-character have been set as a result of executing an I/O command.
d =	the actual character formed by the combination of d-character-control bits of the individual external condition tests.

For example, a branch to a location labeled EXIT is desired if the channel-busy indicator, the not-ready indicator, or the wrong length record indicator has been set following an I/O command. The appropriate *Autocoder* instruction has BEX1 as the operation code, EXIT as the ADDR and L as the d-character. The d-character L results from a combination of the 2-bit, 1-bit, and B-bit required to interrogate the three indicators just mentioned.

The processor assembles the object instruction as follows:

1. Substitutes the actual machine-language operation code for the mnemonic written in the operation field.
2. Substitutes the actual addresses of symbols used in the operand field to specify the X-control field A or I, and B-addresses of the instructions. If address adjustment or indexing is indicated, the sub-

stituted 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.

3. 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.

Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 45 OPI
0 1		BCE	READ, TEST, 5

Figure 40. Branch-if-Character-Equal

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

Examples: Figure 40 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. Assuming that the address of READ is 00596 and TEST is in 00782, the assembled instruction is B 00596 00782 5.

Figure 41 shows an imperative instruction with a unique mnemonic. A branch to a location labeled OVFLO will occur if an arithmetic overflow has occurred. Assuming that the address of OVFLO is 00896, the assembled machine-language instruction is J 00896 Z.

Line 3 5	Label 6 15	Operation 16 20	21 25 30 35 40 45 OPI
0 1		BAV	OVFLO
0 2			•
0 3			•
0 4			•
0 5	OVFLO	ZA	FIELDA, FIELDB

Figure 41. Branch-if-Arithmetic-Overflow

Processor Control Operations

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

OP CODE	PURPOSE
JOB	Job Card
EJECT	Eject
RESEQ	Resequence
LOAD	Load
RUN	Run
CTL	Control Card
ORG	Origin
LTORG	Literal Origin
EX	Execute
XFR	Transfer
SFX	Suffix
PST	Print Symbol Table
END	End

JOB — Job

General Description: This card in the user's source program deck prints a heading line on each page of the output listing from the assembly process and identifies the object 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 1410 characters and appears in columns 21-72.
3. Writes in the identification field (Columns 76 to 80), the information to be contained in the object deck or tape.

The processor:

1. Prints the information, the identification 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 (columns 76-80) in all condensed cards produced for the object program. If another JOB or RESEQ card (or cards) appear elsewhere in the source program, the new identification number will be punched in subsequent condensed cards.

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

EJECT — Eject

An EJECT control card may be placed in the symbolic source program by the user to cause the carriage to restore at any point in the output listing.

The programmer may now have separate routines or sequences in the output listing.

RESEQ — Resequence

The RESEQ control card resets the card sequence count to 001 in the object program deck and the identification number from columns 76 to 80 will replace the former identification number.

This will allow the user to separate his object deck into logical groups or blocks.

LOAD — Load

The LOAD control card is used to signal the processor that a load program should precede the object deck.

RUN — Run

This is the first card in the user's source program deck. It tells the processor which type of run is desired. There are two types: original and systems.

ORIGINAL

By placing the label ORIGINAL in the label field of a run control card, the user signifies that he desires a compilation of a given deck by the processor. The user's source program deck is placed immediately behind this card.

SYSTEMS

By placing the label SYSTEMS in the label field of a run control card, the user signifies that he desires an updating run. This updating run pertains only to the library entries or routines on the systems tape.

CTL — Control

General Description: The control statement is normally the second entry (card) in the source program deck.

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*
20,000	2
40,000	3
20,000	any other code

Column 22: Indicates the storage size of the machine which will be used to process the object program. These codes are the same as those used to indicate the size of the processing machine with the addition of code 1 to specify a 10,000-character storage size.

The processor interprets the codes and processes the source program accordingly.

NOTE: If the CTL card is missing, the processor assumes that both the processing machine and the object machine have 20,000 positions of core storage.

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. Addresses can have address adjustment, 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 in an entry preceding the ORG statement in the program sequence.
4. If the programmer writes an ORG statement and its operand is followed by ,S, the high-address counter will not be affected. For example, ORG 1000,S is an ORG statement that will not affect the high-address counter.

The processor:

1. Assigns addresses to subsequent instructions, constants and work areas starting with the address 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 00500. NOTE: In the absence of the IOCS entries, normal origin will be 00500.
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 (exception: see Figure 47).
4. The processor maintains a high assignment counter which contains the highest assigned location at any given point of an assembly run.

Result: The programmer chooses the area of storage where the object program, defined constants, etc., will be located.

Examples: Figure 42 shows an ORG statement with an actual address. The first symbolic program entry following this ORG statement will be assigned with storage location 00600 as a reference point (if the first entry is an instruction, the op code position (I-address) of that instruction will be 00600; if the first entry is a 5 character DCW, it will be assigned address 00604, etc.).

The ORG statement in Figure 43 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 next available location before re-origin occurs. The processor will continue to assign addresses beginning at the actual address of START.

The programmer can insert another ORG statement later in the source program to direct the processor to

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	ORG	600

Figure 42. ORG Statement with an Actual Address

Line (3–5)	Label (6–15)	Operation (16–20)	OPE (21–45)
01	ADDR	ORG	START

Figure 43. Saving the Address of Last Storage Allocated

begin assigning storage at ADDR. This statement is shown in Figure 44.

Figure 44 shows an ORG statement that directs the processor to start assigning addresses with the actual address assigned to ADDR (see *Programmer* #3).

Figure 45 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 *Processor* #3).

When the processor encounters the statement shown in Figure 46, 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 00725, the next instruction would have an address of 00800.

Figure 47 shows an ORG statement with a blank operand. The processor will assign addresses to subsequent entries beginning at the location designated by the high assignment counter plus one.

Line (3–5)	Label (6–15)	Operation (16–20)	OPE (21–45)
01		ORG	ADDR

Figure 44. ORG Statement with a Symbolic Address

Line (3–5)	Label (6–15)	Operation (16–20)	OPE (21–45)
01		ORG	*+200

Figure 45. ORG Statement with an Asterisk Operand and Address Adjustment

Line (3–5)	Label (6–15)	Operation (16–20)	OPE (21–45)
01		ORG	*+X00

Figure 46. ORG Statement Advancing Address Assignment to Next Available Multiple of 100

Line (3–5)	Label (6–15)	Operation (16–20)	OPE (21–45)
01		ORG	

Figure 47. ORG Statement with a Blank Operand

LTORG — Literal Origin

General Description: LTORG statements are coded in the same way as ORG statements. Their function is to direct the processor to assign storage locations to previously encountered literals 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 48 shows how the programmer can direct the processor to begin assigning the storage locations to literals and closed library routines.

The programmer has instructed the processor to begin storage allocation at 00600. All instructions, constants, and work areas (ending with BSUBRT 01) will be assigned storage. However, the literal (+10) in the statement ZA + 10, WKAREA, and the library routine (SUBRT 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 rou-

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	ORG	600
WKAREA	DCW	#8
CALC	EQU	1500
	ZA	+10,WKAREA
	CALL	SUBRT01
	B	SUBRT01
ADDR	LTORG	CALC
	ORG	ADDR
FIELDA	DCW	#6
FIELDB	DCW	#5
	ZA	FIELDA,FIELDB

Figure 48. Using a LTORG Statement

tine (SUBRT 01) will be assigned address 01500 (because CALC has been equated to 01500). After all instructions in SUBRT 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 B SUBRT 01. Thus, if B SUBRT 01 (J 01500) is assigned locations 00691-00697, FIELDA will be assigned storage locations 00698-00703.

EX — Execute

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

The programmer:

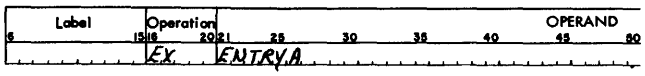
1. Writes the mnemonic operation code EX in the operand field.
2. Writes an actual or symbolic address in the operand field. This address must be the same symbol that appears in the label field of the first instruction to be executed.

The processor:

1. Incorporates closed library routines, literals, and address constants.
2. Assembles a branch instruction (an unconditional branch to the first instruction to be executed), the I-address of which is the address assigned to the instruction referenced by the symbol in the operand field. 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.

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 49 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.



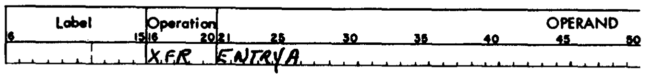
Label (6–15)	Operation (16–20)	OPERAND (21–80)
	EX	ENTRYA

Figure 49. 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 incorporated. An XFR statement transfers to and executes instructions which have been previously loaded.

Example: Figure 50 shows an XFR entry.



Label (6–15)	Operation (16–20)	OPERAND (21–80)
	XFR	ENTRYA

Figure 50. XFR Statement

SFX — Suffix

General Description: This statement directs the processor to put a suffix code in the tenth position of all labels in a source program section which have less than ten 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 1410 character) to be used for the suffix code in the operand field.

The processor:

1. Inserts the suffix code in the tenth position of all labels in the subsequent entries which have less than ten characters.
2. Changes the suffix code when a new SFX card is encountered.

Cross referencing with suffixing: If the programmer wishes to cross reference to a previously used label which is in a section with a different suffix, he may do so by writing the suffix of the different section followed by a dollar sign before the label in the operand.

If JOE appeared as a label in a program section with a suffix A and the given statement is in a section with a suffix B, he may refer to JOE by cross referencing as indicated in Figure 51.

If the programmer desires to suppress suffixing, he may do so by preceding the entry in the operand by a dollar sign (Figure 52).

The programmer can instruct the processor to discontinue suffixing by using an SFX card with a blank operand.

Label 6 … 15	Operation 16 … 20	OPERAND 21 25 30 35 40 45 50
	SFX	B
	ZA	MONEY, A$JOE

Figure 51. Cross Referencing Suffixing

Label 6 … 15	Operation 16 … 20	OPERAND 21 25 30 35 40 45 50
	SFX	B
	ZA	MONEY, $CASH

Figure 52. Suppress Suffixing

PST — Print Symbol Table

General Description: This entry causes the processor to print out the symbol table ahead of the printed listing of the program.

The programmer writes the mnemonic operation code (PST) in the operation field.

The processor lists the symbol table. All labels used in the source program are printed with their assigned core-storage addresses. NOTE: This card can appear anywhere in the source program deck preceding the END card.

END — End

General Description: This is always the last card in the source deck. It signals the processor that all of the source program entries have been read, and provides the processor with the information necessary to create an execute card. This execute card causes a transfer to the first instruction to be executed after the program 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 an unconditional branch instruction which is used as part of the loading data. Other processor functions are the same as for an EX statement.

Example: Figure 53 shows an END card.

Label 6 … 15	Operation 16 … 20	OPERAND 21 25 30 35 40 45 50
	END	START

Figure 53. END Card

The Macro System

Many of the routines that must be incorporated in programs written for the IBM 1410 are general in nature and can be used repeatedly with little or no alteration. The IBM 1410 *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 extract, from a library of routines, a sequence of instructions tailored by the processor to fit his particular program.

Definitions of Terms

Several programming terms are used to 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, for example, 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 out-of-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 cannot be altered, it is *inflexible.* It is *flexible* 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.

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. 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 as well as to update routines. 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. Also, literals and actual addresses can be used.

Pseudo-macro. A macro-instruction that is used internally by the processor to control the production of a series of machine-language instructions. The difference between a pseudo-macro and a macro is that the pseudo-macro is not written in the source program. Instead, it appears only in a flexible library routine which can be extracted by a macro-instruction.

Macro Operations

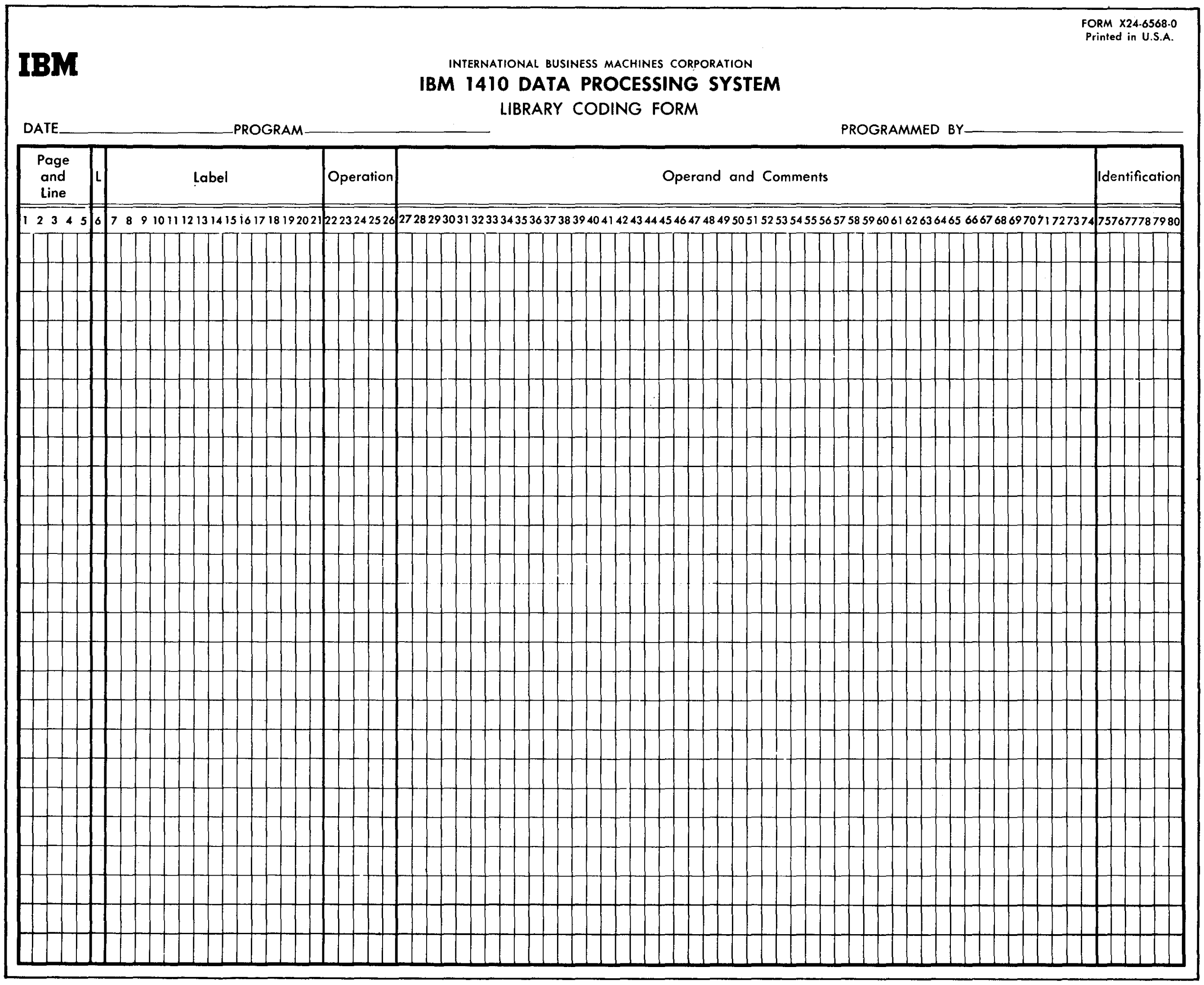
To illustrate the basic operation of the macro system, a hypothetical macro called CHECK with a simple flexible library routine is used. The routine is designed to compare an input field to another field, test the compare indicator for a high, low, or equal condition or any combination of the three.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

Figure 54 is the library coding form which is used with the 1410-Macro-System.

Figure 55 shows 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.* The symbolic program entries are inserted in the source program following 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 CHECK macro was created by writing an INSER statement and four model statements as shown in Figure 55.



FORM X24-6568-0
Printed in U.S.A.

IBM

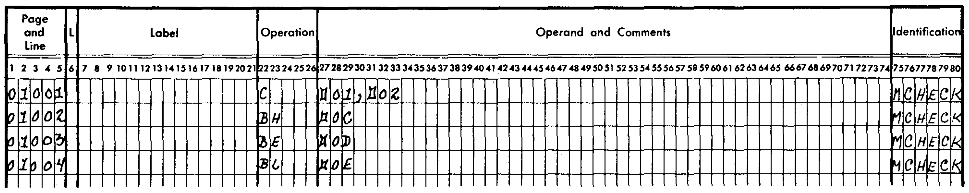
INTERNATIONAL BUSINESS MACHINES CORPORATION
IBM 1410 DATA PROCESSING SYSTEM
LIBRARY CODING FORM

DATE ______ PROGRAM ______ PROGRAMMED BY ______

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

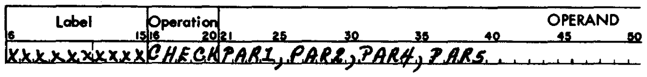
Figure 54. IBM 1410 Library Coding Form

Library Entry



Page and Line	L	Label	Operation	Operand and Comments	Identification
1 2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74	75 76 77 78 79 80
01001			C	□01,□02	MCHECK
01002			BH	□0C	MCHECK
01003			BE	□0D	MCHECK
01004			BL	□0E	MCHECK

Macro Instruction



Label 6–15	Operation 16–20	OPERAND 21 25 30 35 40 45 50
XXXXXXXXXX	CHECK	PAR1, PAR2, PAR4, PAR5

Assembled Symbolic Program Entry

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

Figure 55. Macro Operations

INSER — Insert

General Description: An INSER statement identifies a library routine. This identification precedes the library routine in the library tape.

The programmer:

1. Writes the operation code INSER in the operation field of the *Autocoder* coding sheet.
2. Writes the five-character label for the library routine in the label field. The 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.
3. Writes an M in column 21 of the operand field to indicate a flexible library routine, or an S in column 21 to indicate a CALL or INCLD type library routine.

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

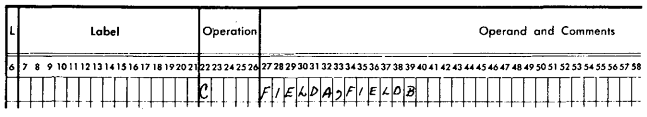
General Description: Model statements establish the conditions for inserting parameters in the object routine and define the basic structure of the symbolic program entries.

The programmer:

1. Designs a general routine to perform many specific functions (depending upon the parameters supplied) when it is executed in the object program.
2. Writes the model statement as follows:
   a. If the entry is complete, it is written exactly the same as though it were an entry in a source program. This entry will be included in all object routines unless a bypass condition exists (see *BOOL).*

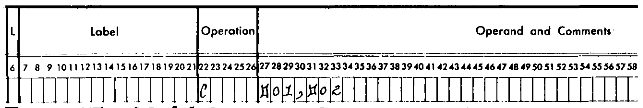
Example. Figure 56.

   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.



L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
		C	FIELDA,FIELDB

Figure 56. Model Statement for a Complete Instruction



L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
		C	⌑01,⌑02

Figure 57. Model Statement for an Incomplete Instruction with Required Parameters

Example: Insert parameters 01 and 02 specified by the CHECK macro-instruction shown in Figure 57.

   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 ⌑ 0B, etc.) to indicate that the entry is to be included in the object routine only if the parameter is specified by the macro-instruction.

Example: Insert parameter 03 in the following instruction if it is specified by the macro-instruction. If parameter 03 does not appear in the macro-instruction, the instruction shown in Figure 58 will be deleted from the object routine.

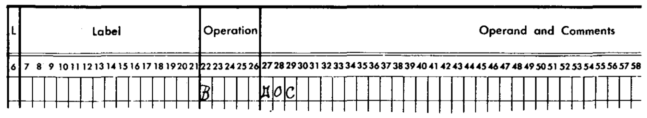
NOTE: Substitution codes can also be used to substitute a parameter in any part of a model statement. For example, it is possible to substitute an operation code, any part of a literal, a label, etc.

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

The macro-instruction label is inserted in the label field of the first model statement included in the assembled object routine as shown in Figure 59. 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 ⌑ 01-99 code.

Example: Insert parameter 02 in the label field of the assembled symbolic program entry as shown in Figure 60.

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 (⌑ 0J = symbolic address 1, ⌑ 0K = symbolic address 2, etc.) can be used. The processor generates



L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
		B	⌑0C

Figure 58. Model Statement for an Incomplete Instruction with Conditional Parameters

Macro Instruction

Label (6–15)	Operation (16–20)	OPERAND (21–50)
TEST2	INVER	START1

Model Statement

L (6)	Label (7–21)	Operation (22–26)	Operand and Comments (27–58)
	□00	B	□01

Assembled Symbolic Program Entry

```
TEST      B      START1
```

Figure 59. Labeling

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 is done to avoid duplicate address assignments for labels.

Example: Use the generated symbolic address of □ 0J as an operand for entry 3 and as the label for entry 6. UPDAT is the 23rd macro encountered in the source program (Figure 61).

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

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

Macro Instruction

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

Model Statement

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

Assembled Symbolic Program Entry

```
START2      SBR   ENTRYA
```

Figure 60. Additional External Labels

Macro Instruction

Label (6–15)	Operation (16–20)	OPERAND (21–50)
	UPDAT	COST, AMOUNT

Model Statement

L (6)	Label (7–21)	Operation (22–26)	Operand and Comments (27–58)
		•	
		•	
		B	□0J
		•	
		•	
	□0J	ZA	□01, □02

Assembled Symbolic Program Entry

```
              •
              •
              B      □0J023
              •
              •
□0J023        ZA     COST,AMOUNT
```

Figure 61. Internal Labels

indexed, it will be cancelled by the indexing in the library routine.

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

NOTES:

1. A model statement in the library routine for a macro-instruction may not be another macro-instruction, except the CALL or INCLD 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 62 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-□9Z	Statement	Substitute parameter (if parameter is missing, delete statement)
□0J-□9R	Label Field and Operand Field	Assign internal label

Figure 62. 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:

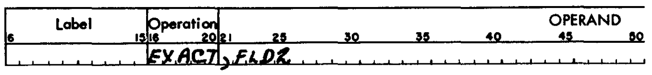
1. Writes the name of the library routine in the operation field. This name must be the same five characters that appear in the label field of the INSER statement of the library entry.
2. Writes in the label field the label that is to be inserted in the label field of the first assembled model statement.
3. Writes in the operand field the parameters that are to be used by the model statements that are 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 substituted wherever a ◻01, or ◻0A appears as a label, operation code, or operand of a model statement.
   b. As many parameters may be used as can be contained in the operand fields of five or fewer coding sheet lines. If more than one line is needed for a macro-instruction, the label and operation fields of the additional lines must be left blank. Parameters must be separated by a comma. They cannot contain blanks or commas unless they appear between @ symbols. The @ symbol itself cannot appear between @ symbols. 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 need not be followed by a comma.
   c. Parameters that are not required for the particular object routine desired can be omitted from the operand field of the macro-instruction. However, if a parameter is omitted, the comma that would have followed the parameter must be included, unless the omitted parameter is behind the last parameter which is included in the macro-instruction. These commas are necessary to count operands up to the last included operand. All operands between the last included operand and operand 99 are assumed by the processor to be absent.

Figures 63, 64, 65, and 66 show how parameters can be omitted. The hypothetical macro-instruction called EXACT is used. EXACT can have as many as nine parameters.

The processor:

1. 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.

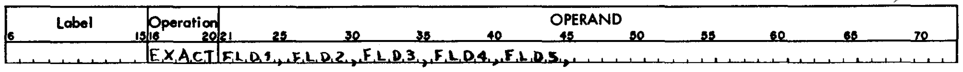


Label (6–15)	Operation (16–20)	OPERAND (21–50)
	EXACT	,FLDZ

Figure 66. Parameters 01, 03, 04, 05, 06, 07, 08, and 09-99 Missing

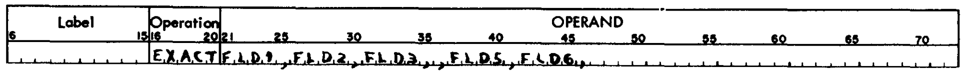
Pseudo-Macro-Instructions

These statements never appear in a user's source program or in the output listing of an assembled 1410 *Autocoder* program. However, they are used in library routines to signal the processor that certain conditions



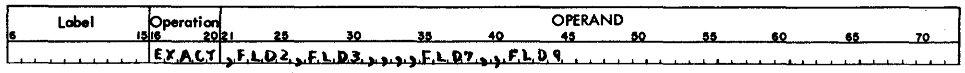
Label (6–15)	Operation (16–20)	OPERAND (21–70)
	EXACT	FLD1,FLD2,FLD3,FLD4,FLD5,

Figure 63. Parameters for EXACT Included; Parameters 10-99 Missing



Label (6–15)	Operation (16–20)	OPERAND (21–70)
	EXACT	FLD1,FLD2,FLD3,,FLD5,FLD6,

Figure 64. Parameters 04, 08, and 10-99 Missing



Label (6–15)	Operation (16–20)	OPERAND (21–70)
	EXACT	,FLD2,FLD3,,,,FLD7,,FLD9

Figure 65. Parameters 01, 04, 05, 06, 08, and 10-99 Missing

exist that can affect the assembly of an object routine. For example, the presence of a pseudo-macro-instruction in a library routine can cause a group of model statements to be deleted. Thus, pseudo-macros provide the writer of library routines with a coding flexibility which exceeds the limitations of the substitution and condition codes described previously.

Pseudo-macro-instructions may be written anywhere in a library routine. The five pseudo-macros incorporated in the 1410 *Autocoder* processor are MATH, BOOL, COMP, NOTE, and MEND.

Permanent and Temporary Switches

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

There are 75 permanent and 99 temporary switches available for recording status conditions. Each switch occupies one core-storage position during the macro phase of *Autocoder*. If a storage position contains the character A (BA 1-bits), the switch is on; if it contains a ? (CBA 82-bits), the switch is off. At the beginning of assembly all switches are off.

PERMANENT SWITCHES

Permanent switches retain status conditions during the entire macro phase unless changed by a pseudo-macro. They are addressed by using a # symbol followed by the two-digit number of the switch to be set or tested. For example, # 01 addresses permanent switch 01; # 02 addresses switch 02; and # 75 addresses switch 75.

TEMPORARY SWITCHES

When the processor encounters a macro-instruction, the temporary switches are set to the condition (presence or absence) of the parameters in the operand of the macro field. If the parameter is present, the corresponding switch is set ON. If the parameter is missing, the switch is set OFF. For example, if parameter 01 is present, temporary switch 01 is turned on. If parameter 02 is missing from the macro-instruction, temporary switch 02 is off. Temporary switches retain status throughout the processing of a macro-instruction unless changed by a pseudo-macro. After the macro-instruction has been completely processed, all temporary switches are set OFF. Temporary switches are addressed by using a □ symbol followed by the two-digit number of the switch to be set or tested. For example, □ 01 addresses temporary switch 01; □ 02 addresses switch 02; and □ 99 addresses switch 99.

For example, if a macro with a maximum of nine parameters is encountered, the processor sets the first nine temporary switches to indicate the presence or absence of these nine parameters. Temporary switches 10-99, which are off, can be used by the pseudo-macros to communicate conditions to the processor while it is working on this particular macro-instruction. This use of temporary switches is recommended because it reserves the permanent switches for communicating information from one macro to another.

MATH — For Solving Algebraic Expressions

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

SUM BOXES

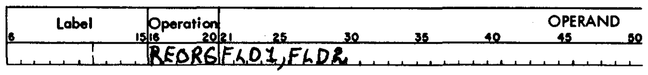
A sum box is a group of five core-storage positions used to store the result of an arithmetic expression. The 1410 *Autocoder* makes available 20 such sum boxes. A sum box is addressed by using a # symbol followed by the two-digit number (ending in zero or five) of the sum box to be referenced. For example, the address of the first sum box is # 05; the address of the second sum box is # 10; and the address of the twentieth sum box is # 00.

At the beginning of the macro phase, a sum box contains $0000\overset{+}{0}$. Any number may be placed in a sum box or added to its contents. The units position of the sum box always contains the sign of the result. Sum boxes retain information placed in them throughout the macro phase and their contents may be used and/or changed from one macro-instruction to another.

Sum boxes can be used by model statements as well as by a psuedo macro. For example, in Figure 67, assume that sum box # 05 contains $1234\overset{+}{5}$ and sum box # 10 contains $0001\overset{-}{5}$.

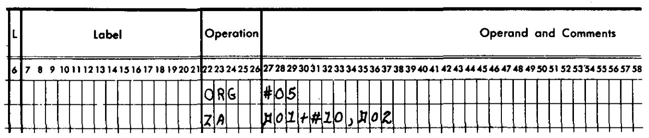
NOTE: ZA FLD1 + 0001N, FLD2 is processed as ZA FLD1—15, FLD2

Macro Instruction



Label (6–15)	Operation (16–20)	OPERAND (21–50)
	REORG	FLD1,FLD2

Model Statement



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

Assembled Symbolic Program Entry

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

Figure 67. Sum Boxes

ARITHMETIC EXPRESSIONS

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

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

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

SIGN SWITCHES

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

The programmer:

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

NOTE: A comma must follow the sum box code, the arithmetic expression, and the individual sign-switch codes. Figure 68 shows the format for a MATH pseudo-macro.

The processor:

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

Example: The math pseudo-macro shown in Figure 69 multiplies parameter 07 by 401 and adds 12 to the result. The answer is stored in SUMBOX 6 (#30). If the result is positive, permanent switch 04 is set ON; if the result is zero, switches 04 and 06 are set ON; if the result is negative, temporary switch 09 is set ON.

BOOL — For Solving Logical Expressions

General Description (the BOOL pseudo-macro can be used):

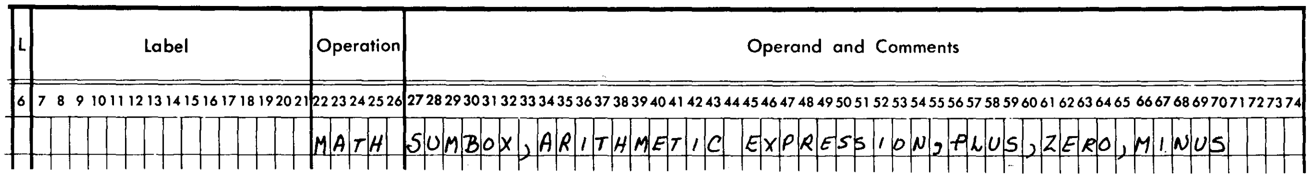
1. To set a permanent or temporary switch as the result of a logical expression.
2. To cause the processor to skip over certain model statements if the logical expression is false. If the statement is true, the processor goes to the next sequential model statement.

The programmer:

1. Writes the name of the pseudo-macro (BOOL) in the operation field.
2. May write a label, the logical expression (statement), and a switch code in the operand field in the format shown in Figure 70.

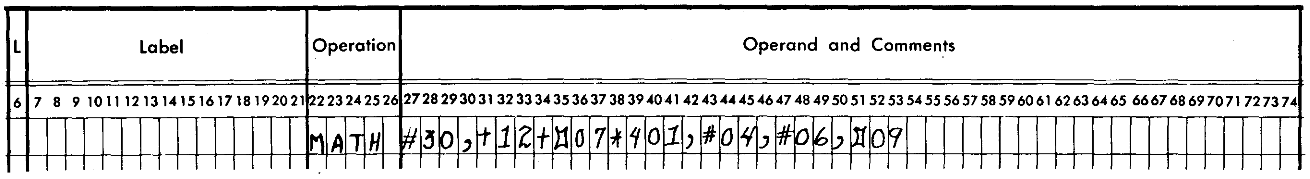
LABELING

A special one-character label permits skipping *forward* in the library routine as the object routine is being



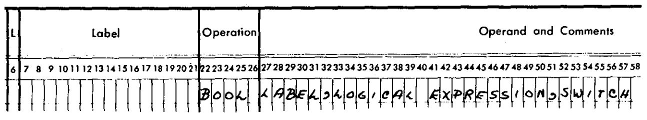
L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
		MATH	SUMBOX,ARITHMETIC EXPRESSION,PLUS,ZERO,MINUS

Figure 68. Format for the MATH Pseudo-Macro



L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
		MATH	#30,+12+⌑07*401,#04,#06,⌑09

Figure 69. MATH Pseudo-Macro



L	Label	Operation	Operand and Comments
		BOOL	LABEL,LOGICAL EXPRESSION,SWITCH

Figure 70. Format for the BOOL Pseudo-Macro

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

LOGICAL EXPRESSION

The BOOL pseudo-macro can have any combination of three logical operations: * (and), + (or), and — (not). The operators are defined in Figure 71. The combination of these operators and the switches to be tested make up the logical expression (see example, Figure 72).

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

SWITCHES

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

The processor:

1. Examines the status switches to determine whether all conditions specified in the logical expression are satisfied. If they are, the expression is true. If the logical condition is not met, the expression is false.
2. Sets the specified status switch to ON or OFF to reflect the true or false condition.
3. If a false condition exists and a label appears in the BOOL operand, the processor skips forward to the command or model statement containing a corresponding label in its label position.

To determine if a logical expression is true or false:

a. call all ON conditions true and all OFF conditions false.
b. let 1 = true and 0 = false.
c. calculate the logical value of the expression.

If the logical value of the expression is zero, the expression is false. If the logical value is one, the expression is true. For example, if switches 01, 02, 03 and 04 are on, the expression

@□01*□02@+@□03*□04@

is true because:

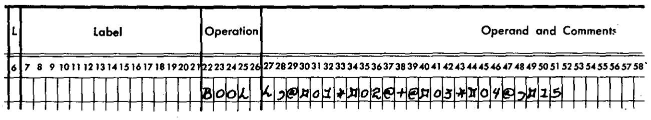
$$(\text{ON} * \text{ON}) + (\text{ON} * \text{ON}) =$$
$$(1 * 1) + (1 * 1) =$$
$$1 + 1 = 1$$

Examples: Figure 72 shows how the BOOL pseudo-macro can be used. The BOOL entry states:

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

The example shown in Figure 73 states:

1. If (both temporary switches 01 and 02) or (both

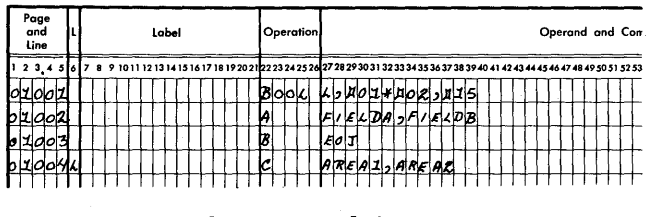


L	Label	Operation	Operand and Comments
		BOOL	L,@□01*□02@+@□03*□04@,□15

Figure 73. BOOL Pseudo-Macro

*	+	—
1 * 1 = 1	1 + 1 = 1	—1 = 0
1 * 0 = 0	1 + 0 = 1	—0 = 1
0 * 1 = 0	0 + 1 = 1	
0 * 0 = 0	0 + 0 = 0	

Figure 71. Table of Operators



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

Figure 72. Using the BOOL Pseudo-Macro

SWITCHES

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

Figure 74. True Conditions

temporary switches 03 and 04) are ON, the statement is true. Therefore, set temporary switch 15 ON.

2. However, if (either temporary switch 01 or 02) and (either temporary switch 03 and 04) is OFF, the statement is false. Therefore, set temporary switch 15 OFF and skip to the model statement whose label is L.

Figure 74 is a table showing all conditions that will cause the BOOL statement shown in Figure 73 to be true.

Figure 75 is a table showing all conditions that will cause the BOOL statement shown in Figure 73 to be false.

SWITCHES

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

CONDITIONS

FALSE

Figure 75. False Conditions

COMP — To Compare Two Fields

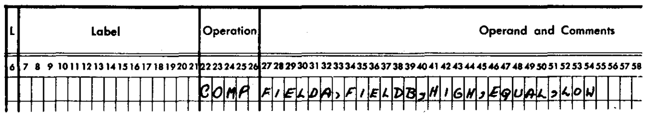
General Description: The COMP pseudo-macro compares an A-field to a B-field and sets permanent or temporary switches to indicate the result of the comparison.

The programmer:

1. Writes the name of the pseudo-macro (COMP) in the operation field.
2. Writes the operand field in the format shown in Figure 76. The first and second entries are the A- and B-fields. The A- and B-fields may be any of the parameters 01-99, sum boxes #05-#00, or literals.

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

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



L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
		COMP	FIELDA,FIELDB,HIGH,EQUAL,LOW

Figure 76. Format for COMP Pseudo-Macro

The processor:

1. Compares the A-field to the B-field.
2. Sets the status switches to the result of the compare:
   a. The first switch is set ON, if the value of the B-field is greater than that of the A-field.
   b. The second switch is set ON, if the B-field is equal to the A-field.
   c. The third switch is set ON, if the value of the B-field is less than that of the A-field.

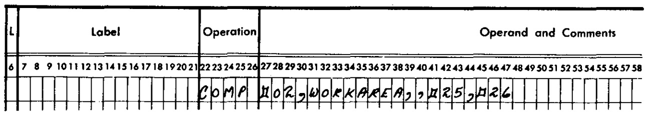
Examples: Figure 77 shows a COMP pseudo-macro which states:

1. Compare parameter 02 of the macro statement to WORKAREA.
2. If parameter 02 is WORKAREA, turn on temporary switch 25.
3. If parameter 02 is lower than WORKAREA, turn on temporary switch 26.

Figure 78 shows a COMP pseudo-macro which states:

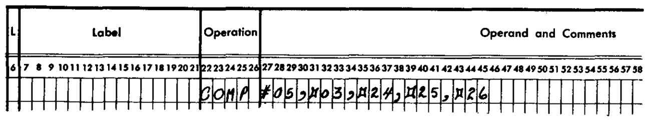
1. Compare the contents of sum box 05 to parameter 03 of the macro statement.
2. If the result is HIGH, set temporary switch 24.
3. If the result is EQUAL, set temporary switch 25.
4. If the result is LOW, set temporary switch 26.

NOTE: Standard 1410 collating sequence determines HIGH, EQUAL, or LOW conditions. Comparisons are controlled by the B-field in the 1410. Thus, the statement shown in Figure 79 will cause temporary switch 25 to be set ON if the low-order position of parameter 02 is an @ symbol (if parameter 02 is an alphamerical literal).



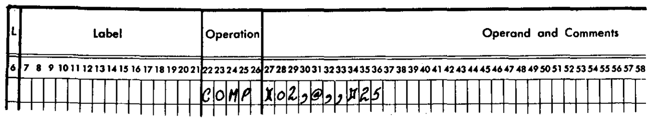
L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
		COMP	⌑02,WORKAREA,,⌑25,⌑26

Figure 77. COMP Pseudo-Macro



L	Label	Operation	Operand and Comments
6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21	22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
		COMP	#05,⌑03,⌑24,⌑25,⌑26

Figure 78. Comparing a Parameter to the Contents of a Sum Box



L	Label	Operation	Operand and Comments
		COMP	⌑02,@,,⌑25

Figure 79. Checking for an Alphamerical Literal

NOTE — To Produce a Message

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

The programmer:
1. Writes the name of the pseudo-macro (NOTE) in the operation field.
2. Writes the message in the operand field. The page and line number of the macro statement that is being processed will precede the printed message.

The processor prints the message and its accompanying identification numbers on the console printer.

Example: Figure 80 shows how the NOTE pseudo-macro can be used in combination with the BOOL pseudo-macro. The BOOL pseudo-macro tests to insure that parameters 01 and 02 are present in the macro-instruction. If either parameter is missing, the processor skips to the NOTE pseudo-macro and prints:
1. The page and line number of the macro-instruction.
2. PARAMETER ABSENT FROM MACRO.

MEND — End of Routine

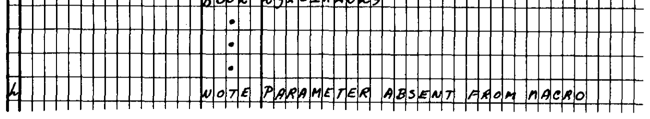
General Description: This pseudo-macro signals the end of generation for a macro-instruction. It may appear anywhere in a library routine.

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

The processor stops processing the macro-instruction when it encounters a MEND statement.

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

Example: Figure 81 shows a MEND pseudo-macro.



L	Label	Operation	Operand and Comments
		BOOL	L,⌑01*⌑02,
		•	
		•	
		•	
L		NOTE	PARAMETER ABSENT FROM MACRO

Figure 80. NOTE Pseudo-Macro

Pseudo-Macro Coding Example

Example: Figure 82 shows the library entry for a hypothetical macro called PRLIT. This library routine uses all of the five pseudo-macros. It illustrates the effect of the pseudo-macros on the processing of a macro-instruction. The meaning of each line in the library routine is:

Entry 1. If parameter 01 is prèsent, set temporary switch 50 ON and go to entry 3. If parameter 1 is missing, go to entry 2.

Entry 2. Print the note: OPERAND 01 ABSENT.

Entry 3. If permanent switch ten is off, go to entry 5. If permanent switch 10 is on, take entry 4.

Entry 4. ORG at the contents of sum box #05.

Entry 5. Put the contents of sum box #05 plus 100 in sum box #05.

Entry 6. Store the contents of the B-address register in an address equal to the address assigned to the internal label (□ 0K) + 5.

Entry 7. Move five zeros to the field whose symbolic address is parameter 03 of the macro-instruction.

Entry 8. Add the literal + 3 to the field specified by the parameter 03.

Entry 9. Branch to parameter 04.

Entry 10. If parameter 02 is a literal, the EQUAL switch (□ 51) is set ON.

Entry 11. If the EQUAL switch (temporary switch 51) is off, skip to entry 15. If the EQUAL switch is on, go to entry 12.

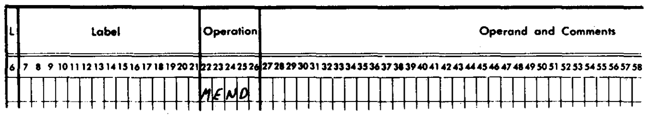
Entry 12. Move parameter 02 to parameter 01.

Entry 13. Subtract parameter 02 from parameter 06. (If parameter 06 is missing, this statement will be bypassed.)

Entry 14. Move parameter 03 to parameter 05.

Entry 15. On the typewriter print the field whose address is specified by parameter 05.

Entry 16. Branch to 0 if any of the I/O Channel status indicators is on.



L	Label	Operation	Operand and Comments
		MEND	

Figure 81. MEND Pseudo-Macro

Page and Line	L	Label	Operation	Operand and Comments
01001			BOOL	A,-⌑01,⌑50
01002			NOTE	OPERAND 01 ABSENT
01003	A		BOOL	A,#10
01004			ORG	#05
01005	A		MATH	#05,#05+100,,,
01006			SBR	⌑0K+5
01007			MLCA	@00000@,⌑03
01008			A	+3,⌑03
01009			B	⌑04
01010			COMP	⌑02,@,,⌑51,
01011			BOOL	B,⌑51,
01012			MLCA	⌑02,⌑01
01013			S	⌑02,⌑0F
01014			MLC	⌑03,⌑05
01015	B		WCP	⌑05
01016		⌑OK	BA1	0
01017			BOOL	C,-⌑51,
01018			MLCA	@⌑02@,⌑01
01019	C		MEND	

Figure 82. PRLIT Library Routine

Entry 17. If temporary switch 51 is on, skip to entry 19. If temporary switch 51 is off, go to entry 18.

Entry 18. Insert parameter 02 as a literal, and move it to the field represented by parameter 01.

Entry 19. End-of-library routine.

Assume that:

1. the macro shown in Figure 83 is encountered in the source program.
2. Permanent switch 10 is ON.
3. Sum box #05 contains 12345.

Macro Instruction

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

Assembled Symbolic Program Entry

```
        ORG  12345
        SBR  ⌑0K023+5
        MLCA @00000@,FIELD1
        A    +3,FIELD1
        B    EXIT1
        MLCA @42AB@,AREAA
        MLC  FIELD1,WORKAREA
        WCP  WORKAREA
⌑0K023  BA1  0
```

Figure 83. Using the PRLIT Routine

Call Routines

The 1410 *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 at least five characters in length, and each of these labels must have the same first five characters.

   CALL routines are stored as controlled by literal origin at the time and place where an 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 and INCLD are invalid in CALL routines. All other symbolic entries acceptable to *Autocoder*, except Literal Origin, Execute, and End, can be used. A CALL macro can be used:
   a. to allow one CALL routine to be used at some point in another CALL routine, or
   b. as a model statement in the library routine for a regular macro-instruction.

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 stores that routine elsewhere in the program. The CALL macro is part of the *Autocoder* processor.

The programmer:

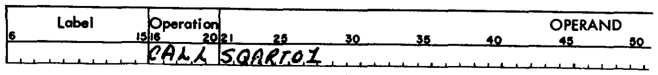
1. Writes the name of the macro (CALL) in the operation field.
2. Writes in the operand field the label of the library statement which is the desired entry point in the library routine. The five characters of this label must be the same as the five characters in the label field of the INSER statement that was used to enter the routine in the library tape (see *INSER).*
   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 SQART01 requires that the number whose square root is to be computed must be placed in a location labeled SQART02. The CALL macro is written as shown in Figure 84.
   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. 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 SUBRT01 (Figure 85). The addresses of DATA 1, DATA 2, and DATA 3 are needed by the CALL routine.

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

Call Macro



Label (6–15)	Operation (16–20)	OPERAND (21–50)
	CALL	SQART01

Assembled Symbolic Program Entry

```
B       SQART01
```

Figure 84. CALL Statement Specifying That Data Is in Specifically Labeled Areas of Storage

Call Macro

Label	Operation	OPERAND
	CALL	SUBRT01,DATA1,DATA2,DATA3

Assembled Symbolic Program Entry

```
B      SUBRT01
DCW    DATA1
       DATA2
       DATA3
```

Figure 85. CALL Statement for a Routine with Arbitrary Data-Storage Assignments

operand for this branch instruction is the entry point given in the operand field of the CALL macro as shown in Figures 84 and 85. 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 85. NOTE: 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 SQART01 which calculates the square root of a number in a work area (SQART02) and places the answer in another work area (SQART03). The programmer can design a library entry for the TAKSQ macro that will provide linkage to the CALL routine as shown in Figure 86.

When the object routine is executed, X + Z will be stored in SQART02. Then the program will branch to the CALL routine where the square root of X + Z will be calculated and the result stored in SQART03. The last instruction in the SQART01 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 shows 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 87.

Library Entry

L	Label	Operation	Operand and Comments
		ZA	⌑01,SQR02
		A	⌑02,SQR02
		CALL	SQR01
		ZA	SQR03,⌑03

Macro Instruction

Label	Operation	OPERAND
	TAKSQ	X,Z,RESULT

Assembled Symbolic Program Entry

```
TAKSQX,Z,RESULT
ZA     X,SQART02
A      Z,SQART02
CALL   SQART01
B      SQART01
ZA     SQART03,RESULT
```

Figure 86. CALL Statement within a Library Routine for a Macro-Instruction

Incld Macro

General Description: This macro is used to extract 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 88 shows an INCLD statement that causes a library routine named SUBRT01 to be incorporated in the object program.

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,

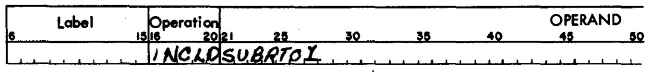
Source Program Entries

Label	Operation	OPERAND
	ZA	X,SQART02
	A	Z,SQART02
	CALL	SQART01
	ZA	SQART03

Assembled Symbolic Program Entry

```
ZA     X,SQART02
A      Z,SQART02
B      SQART01
ZA     SQART03,RESULT
```

Figure 87. Alternative Source Program Entries



Label 6–15	Operation 16–20	OPERAND 21–50
	INCLD	SUBRT01

Figure 88. INCLD Statement

END, or EX time by the processor. Thus, one exit from the main routine can be used to cause several library routines to be executed at object time.

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.

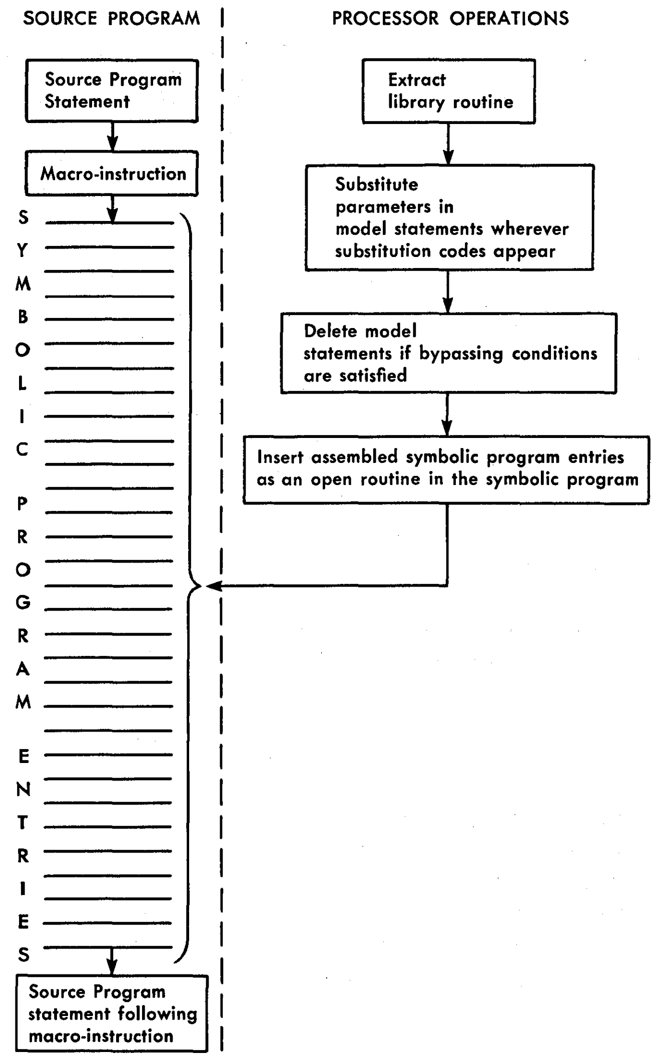
If CALL or INCLD are written *within* a library routine, only a single operand is permitted in the CALL or INCLD statement. This single operand is the name or entry point of the closed library routine. (See *Call Macro.)*

Macro Processing

Figures 89, 90, and 91 show the effect 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 89, 90, and 91 are later processed as though the user had himself, inserted all the entries in the source program. Symbolic entries are translated to machine-language instructions, constants cards are produced, etc.


SOURCE PROGRAM
PROCESSOR OPERATIONS
Source Program Statement
Macro-instruction
SYMBOLIC PROGRAM ENTRIES
Source Program statement following macro-instruction
Extract library routine
Substitute parameters in model statements wherever substitution codes appear
Delete model statements if bypassing conditions are satisfied
Insert assembled symbolic program entries as an open routine in the symbolic program

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 users source program.

Figure 89. Macro Processing

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 line number(s) of the model statement(s) to be deleted. If a whole routine is to be deleted, the operand field contains only an M or S. If more than one model statement of a continuous sequence are to be deleted, the first and last numbers must be written separated by commas.

The processor deletes the model statement or statements specified in the operand field.

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

Examples: Figure 92 is a DELET statement that will cause the whole CHECK library routine to be removed from the library.

Figure 93 is a DELET statement that will cause the first model statement to be deleted from the CHECK library routine.

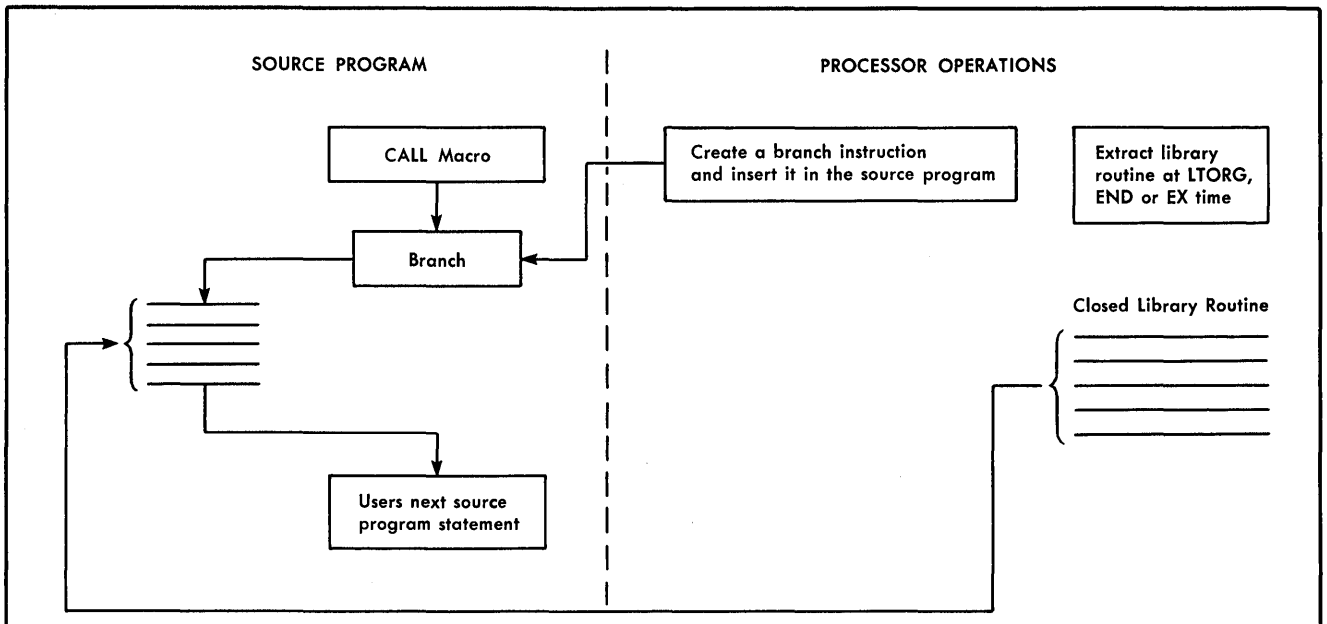
Figure 94 is a DELET statement that will cause model statements 2, 3, 4, and 5 to be deleted.

INSER — Insert

General Description: This entry can be used to insert whole library routines or part of a library routine in the library tape.

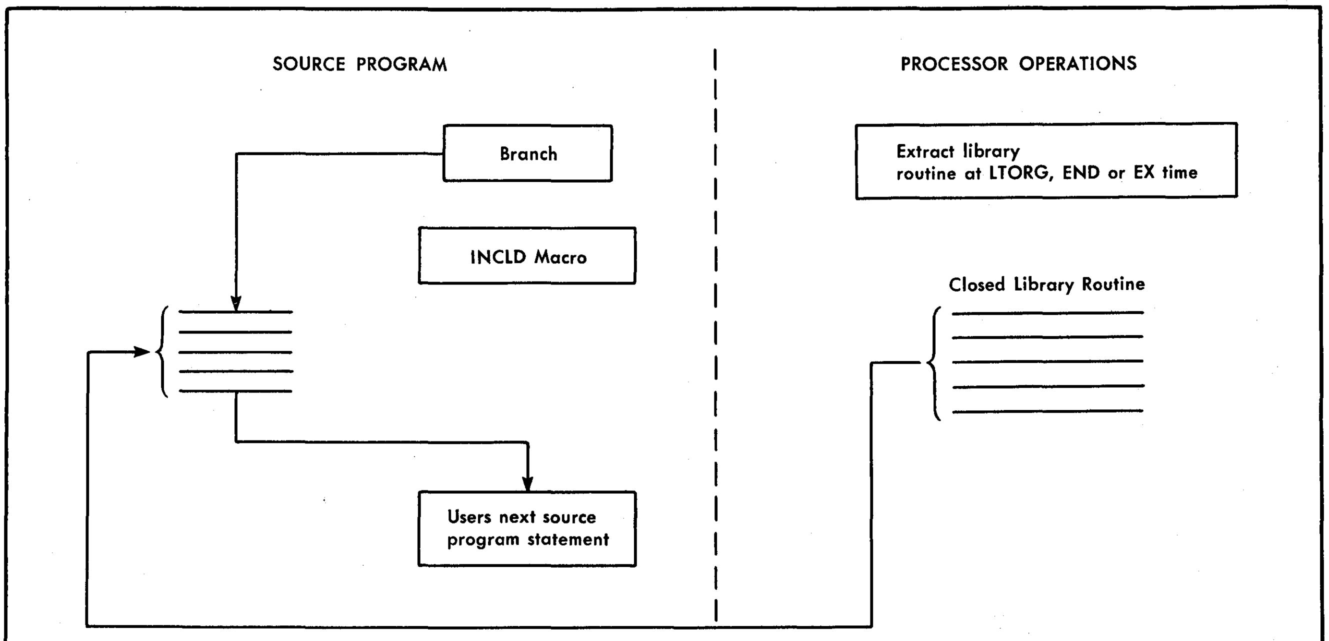
The programmer:

1. Writes the mnemonic operation code (INSER) in the operation field.


SOURCE PROGRAM
PROCESSOR OPERATIONS
CALL Macro
Create a branch instruction and insert it in the source program
Extract library routine at LTORG, END or EX time
Branch
Closed Library Routine
Users next source program statement

When the processor encounters a CALL macro, it creates an unconditional branch instruction to link the main program to the library routine. The branch instruction is placed in the symbolic program immediately following the CALL macro statement. Later, when the processor encounters a LTORG, END or EX statement in the source program, it extracts all library routines specified by CALL macros and stores them as closed routines.

Figure 90. CALL Processing


SOURCE PROGRAM
PROCESSOR OPERATIONS
Branch
Extract library routine at LTORG, END or EX time
INCLD Macro
Closed Library Routine
Users next source program statement

When the processor encounters an INCLD macro, it incorporates the specified library routines when an LT ORG, END, or EX statement is encountered in the user's source program. Note that the branch instruction that links the main routine to the closed library routine is provided by the programmer.

Figure 91. INCLD Processing

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 deletes model statements, if necessary, and inserts the model statement(s) in the library routine.

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

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

Figure 92. Deleting an Entire Library Routine

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

Figure 93. Deleting a Single Model Statement

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

Figure 94. Deleting Multiple Model Statements

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

Figure 95. Inserting an Entire Library Routine

Autocoder Statement

Label (6–15)	Operation (16–20)	OPERAND (21–50)
CHECK	INSERM,0	

Model Statement

L (6)	Label (7–21)	Operation (22–26)	Operand and Comments (27–58)
		C	⌑01,⌑02

Figure 96. Inserting a Single Model Statement

Examples: Figure 95 is an INSER statement that will cause a library routine named CHECK to be inserted in the library tape.

Figure 96 is an INSER statement that will cause new model statement 1 to be inserted in the COMPR library routine.

Figure 97 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 to be inserted in its place.

Figure 98 is an INSER statement that causes model statements 1 and 2 to be deleted and the model statements shown to be inserted in their places.

Autocoder Statement

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

Model Statement

L (6)	Label (7–21)	Operation (22–26)	Operand and Comments (27–58)
	⌑OA	B	PAR4

Figure 97. Substituting One Model Statement for Another

Model Statement

L (6)	Label (7–21)	Operation (22–26)	Operand and Comments (27–58)
	⌑OA	B	PAR4
		C	PAR1,PAR2
		BE	PAR3

Autocoder Statement

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

Figure 98. Substituting Multiple Model Statements

Index

IBM

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