File No. 1401/1460-22
Form C24-3319-0

IBM Systems Reference Library

Autocoder (on Tape) Language Specifications and Operating Procedures IBM 1401 and 1460

Program 1401-AU-037

This reference publication contains the language specifications and operating procedures for the Autocoder (on Tape) programming system. The IBM 1401 Autocoder processor program produces machine-language object programs for IBM 1401 and IBM 1460 from source programs written in the symbolic language of Autocoder.

The language specifications are divided into two sections. The first section contains the specifications of the symbolic language (mnemonics, labels, address types, and control operations) and the rules for writing the source program. The second section describes macro operations and macro instructions.

The operating instructions supplement the language specifications section of this publication. Described are the procedures to be performed by the operator when assembling an Autocoder program on an IBM 1401 or 1460 tape system. The phases of the Autocoder processor are explained and system halts and restarts are given.

For a list of associated publications and abstracts, see the IBM *1401 and 1460 Bibliography*, Form A24-1495.

Major Revision, November 1964

This publication, C24-3319-0, is a major revision of, and obsoletes, C24-1434-0, C24-3104-0, and Technical Newsletters N24-0212 and N24-0233. The main change is the consolidation of C24-1434-0 and C24-3104-0. Other changes include modifications to the address constants section and to the label description section of the *Specifications*.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the content of this publication to IBM Product Publications, Endicott, New York 13764.

© 1961, 1963 by International Business Machines Corporation

Contents

Autocoder (on Tape) Specifications 5
Machine Requirements 5

Programming with Autocoder 6
Symbolic Language 6
Coding Sheet 8
Address Types 10
Indexing 13
Declarative Operations 14
Imperative Operations 20
Processor Control Operations 21

The Macro System 27
Macro Operations 27
Macro Instructions 31
The System Tape 37
Additional Language Specifications 39

Autocoder (on Tape) Operating Procedures 41
Writing the System Tape 42
Pre-System Run 42
System Card Deck Format 42
Autocoder Listing Format 43
System Run 43
System Tape Format 43
Librarian Run 44

Program Assembly 47
Autocoder Phases 47
Autocoder Output 49
Reassembly Run 56
Patching the Object Program 57
Running the Object Program 58

Index 59

This Autocoder programming system for the IBM 1401 and IBM 1460 is called *Autocoder (on Tape)* because the 1401 processor program operates from magnetic-tape units.

The Autocoder processor program produces machine-language object programs from source programs written in the symbolic language of Autocoder. The Autocoder language includes the following significant features:

- Mnemonic Operation Codes—more easily remembered than the actual machine-language operation codes.
- Symbolic and Literal Operands—free the programmer of the burden of core-storage address assignment and reference—actual constants can be used without prior definition.
- Area-Definition Statements—allocate core storage for input/output areas (including multiple-record areas) and work areas and equate these to symbolic labels.
- Macro System—a basic set of frequently used subroutines supplied by IBM is easily added to by the user. These library routines can be tailored to fit a particular application and included in a program by the use of a single macro instruction.

Machine Requirements

The IBM 1401 Autocoder processor can assemble programs for all 1401 and 1460 systems. The processor does not include mnemonics for IBM 1311 and 1301 disk storage operations. Program that use the 1311 or 1301 can be written more easily for the Autocoders that operate from those disk units. Where necessary, however, instructions for those units can be assembled by this tape Autocoder. (See *Disk Input/Output Instructions.*)

The minimum machine configurations required to operate the 1401 Autocoder processor for program assembly are:

IBM 1401 Data Processing System

- 4,000 positions of core storage
- Four IBM 7330 or 729 Magnetic Tape Units. (A fifth magnetic tape unit can be used for delayed multiple program output.)
- IBM 1403 Printer, Model 2, or IBM 1404 Printer
- IBM 1402 Card Read-Punch
- The following special features:
  Advanced Programming
  High-Low-Equal Compare
  Sense Switches (Not necessary for original assembly from a source program card deck, but necessary for all other Autocoder operations.)

IBM 1460 Data Processing System

- 8,000 positions of core storage
- Four IBM 7330 or 729 Magnetic Tape Units. (A fifth magnetic tape unit can be used for delayed multiple program output.)
- IBM 1403 Printer, Model 2 or 3
- IBM 1402 Card Read-Punch
- The following special features:
  Indexing and Store-Address Register
  Sense Switches (Not necessary for original assembly from a source program card deck, but necessary for all other Autocoder operations.)

Programming with Autocoder

A programmer's job is divided into two phases:
1. *Defining* the problem to be solved.
2. *Coding* the source program for assembly by the Autocoder processor.

Start defining the program by outlining its requirements. Draw a block diagram of the procedural steps that are necessary to achieve the desired result. From this decide what data, constants, work areas, and instructions are needed to execute the program.

Constants are fixed data (such as a 10% discount or a serial number).

Work areas are locations within core storage where data can be manipulated (such as input and output areas, and accumulator fields).

After the program requirements are outlined, symbols, instead of actual machine addresses, can be used to refer to areas, data, and instructions.

The 1401 and the 1460 tape Autocoder is divided into two major categories: the symbolic language used by the programmer to write the source program, and the processor program that translates this symbolic language and assembles an actual machine-language object program.

Symbolic Language

The symbolic language of the Autocoder includes a standard set of mnemonic operation codes. They are easier to remember than the machine-language codes because they are usually abbreviations for actual instruction descriptions. For example:

Description	*Mnemonic*	*Machine-Language Code*
Multiply	M	@
Clear Word Mark	CW	□

Figure 1 shows a list of mnemonic operation codes for the IBM 1401 and the IBM 1460 tape Autocoder. Also included in the language are standard mnemonics for statements that define and allocate areas, enter constants, control the area in core storage where the object program will be assigned, etc. These mnemonics have no machine-language equivalent.

The names (symbols) given to data, instructions, and constants are also part of the symbolic program and are usually abbreviations for card fields, record names, and similar items that require frequent reference in the source program.

The Source Program

The source program consists of statements written in symbolic language. These statements contain the information that the processor must have to assemble the object program. This information is divided into four major categories:
- Area Definitions (Declarative Operations)
- Instructions (Imperative Operations)
- Processor Controls (Processor Control Operations)
- Macro Instructions (Macro Operations)

The declarative, imperative, and processor-control operations are described in this section. Macro operations are described in the section, *The Macro System.*

Area-Definition Statements

Area-definition statements reserve areas in core storage to store constants, or to work with data before it is punched, printed, or written on magnetic tape. Area-definition statements, in most cases, do not produce instructions to be executed as part of the object program. For these statements the processor program produces cards containing constants and their assigned machine addresses. These constant cards are loaded with the object program each time the program is used.

For example, a constant card containing the date to be printed on the heading line of each invoice is loaded into core storage. A word mark is placed over the high-order position of the date. The date can then be moved, during object-program execution, to a place in the print area in preparation for printing a heading line. To change the date, duplicate all columns in the constant card except the columns that contain the date itself. Then punch the new date in the card and insert it in the program deck in place of the outdated constant card.

Instruction Statements

Most of the statements in the source program are instructions that are used to read in data, process it, and write it out. The processor program translates them to machine-language instructions and causes the object program to be punched in cards or written on magnetic tape. The processor generates an additional sequence of instructions (called a *loader*) that loads the object program into the correct core-storage positions at program-load time.

Processor-Control Statements

The 1401/1460 tape Autocoder permits a limited amount of programmer control over the assembly process. For example, to locate a program in a particular

DECLARATIVE OPERATIONS

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

IMPERATIVE OPERATIONS

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

CONTROL OPERATIONS

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

† d-character must be placed in operand when coding in Autocoder.
*All disk I/O mnemonics are for IBM 1405 Disk Storage only.

Figure 1. IBM 1401/1460 Tape Autocoder Mnemonic Operation Codes

area of core storage, direct the processor program to start assigning core storage at a specific address by writing an ORG (see *ORG—Origin*) processor-control statement. These statements are used by the processor during assembly.

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

The Processor Program

The 1401 processor program analyzes the information it receives when the source-program statements are fed into the machine. After all the statements have been analyzed, the processor program automatically assembles the machine-language object program, punches it into cards, or writes it on magnetic tape. The punched output cards or program tape also contains the loader. Thus, the object program is called *self-loading*.

Coding Sheet

The Autocoder coding sheet (Figure 2) is free-form (the operand portion of each line is not subdivided into fields), thus allowing the programmer greater coding flexibility. The SPS coding sheet is fixed-form (the operand portion of each line is divided into specific fields).

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

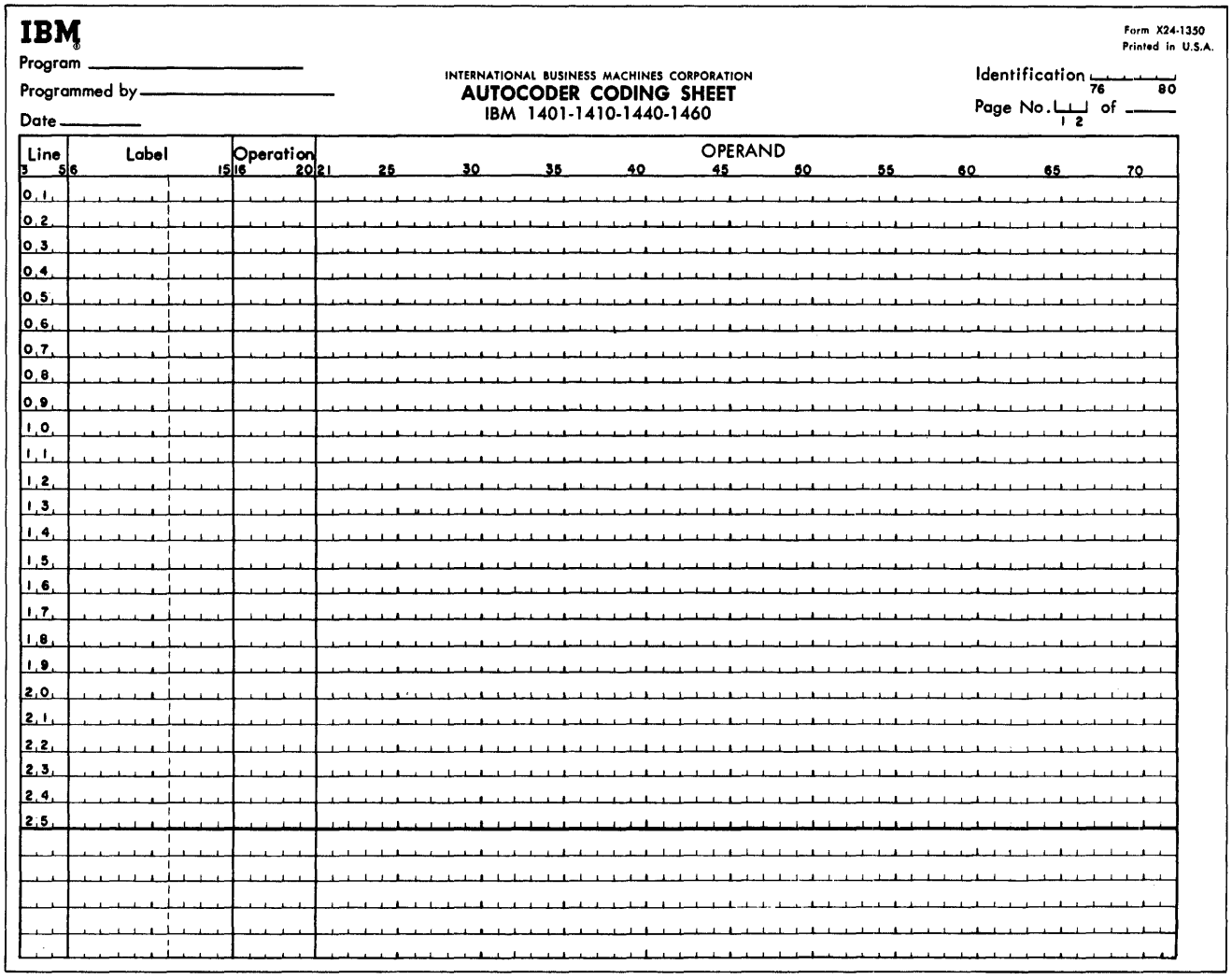


Figure 2. Autocoder Coding Sheet

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 alphameric characters may be used. Standard IBM 1400 series collating sequence should be followed when sequencing pages.

Line Number (Columns 3-5)

A three-character line number sequences entries on each coding sheet. The first 25 lines are prenumbered 01-25. The third position can be left blank (blank is the lowest character in the collating sequence). The five unnumbered lines at the bottom of each sheet can be used to continue line numbering or to make insertions between entries elsewhere on the sheet. The units position of the line number is used to indicate the sequence of inserts. Any alphameric character can be used, but standard collating sequence should be used. For example, if an insert is to be made between lines 02 and 03, it could be numbered 025. Line numbers do not necessarily have to be consecutive, but the deck should be in collating sequence, for sorting purposes.

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

Label (Columns 6-15)

A symbolic label can have as many as six alphameric characters, but the first character must be a letter (A through Z; it cannot be a blank). Special characters and blanks must not be used within a label. The label starts in column 6. Columns 12-15 are always blank.

Operation (Columns 16-20)

Mnemonic operation codes are written in the operation field starting in column 16. Figure 1 is a chart showing 1401 and 1460 tape Autocoder mnemonics.

Operand (Columns 21-72)

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

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

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

A A01 140

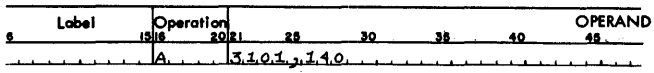


Figure 3. Autocoder Instruction with Actual Address

Figure 4 is an imperative instruction with two symbolic operands and a d-character. Although many of the augmented operation codes available for use with Autocoder eliminate the need to write the d-character in a symbolic instruction, sometimes the d-character must be specified by the programmer. If an instruction requires such a specified d-character, it is written following the A- and B-operands, and is separated from the remainder of the instruction by a comma. The assembled machine-language instruction is: B 392 498 2. It tests a location labeled SWITCH (498) and branches to ENTRYA (392) for the next instruction if SWITCH contains a 2.

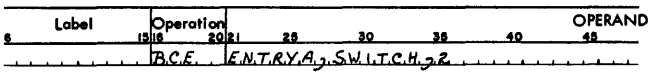


Figure 4. Autocoder Instruction with a d-character

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

Comments

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

Entire lines of information can be included anywhere in the program except within a complete DA entry, or between CTL and DIOCS cards during reassembly and regeneration of IOCS, by writing a comments line. This becomes a comments card when it is punched before assembly. This card can contain comments only and must have an identifying asterisk in column 6. Use columns 7-72 for the comment. The information in a

comments card appears in the symbolic-program listing produced by the processor during assembly, but it does not affect the object program in any way.

CALL or INCLD in columns 16-19 and 16-20, respectively, of a comments card will cause errors in assembly if these comments cards are macro model statements. Columns 16 through 18 of a comments card must not contain END.

Blank (Columns 73-75)

In the 1401 and the 1460 tape Autocoder, columns 73 through 75 are always blank.

Identification (Columns 76-80)

To identify a program or program overlay, assign it an identification number or description. Punch this number into each card in the source deck. The processor does not use this field.

Other Coding Sheet Areas

The areas labeled *Program, Programmed by,* and *Date* are for the user's convenience only. Their contents are never punched in the source deck cards.

Address Types

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

Blank

A blank operand field is valid:

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

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

Actual

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

Symbolic

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

Asterisk (*)

If an * appears as an operand in the source program, the processor will replace it in the object program with the actual core-storage address of the last character of the instruction in which it appears. For example, the instruction shown in Figure 5 is assigned core-storage locations 340-343. The assembled instruction is • 3 4 0.

Asterisk operands can have address adjustment and indexing.

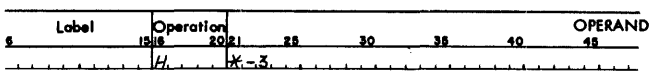


Figure 5. Asterisk Operand in an Autocoder Instruction

Literals

The IBM 1401/1460 tape Autocoder permits the user to put in the operand field of a source program statement the actual data to be operated on by an instruction. This data is called a *literal.* The processor allocates storage for literals and inserts their addresses in the operand or operands of the instructions in which they appear. The processor produces a DCW card that puts a word mark in the high-order position of a literal when it is stored at program load time. (In SPS, literals were not permitted. The actual data to be operated on had to be stored by DCW or DC statements.) Literals are permitted only in the operand field of an Autocoder statement and can be numeric or alphameric. A literal can be any length, provided the operand of the statement that contains the literal does not exceed 52 columns (a statement must be contained in one line of the coding sheet and must not extend beyond column 72). Literals cannot have address adjustment or indexing.

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

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

Figure 6. Literals

Numeric Literals

Numeric literals are written according to the following specifications:

1. A plus or minus sign must precede a numeric literal. The processor puts the sign over the units position of the number when it is assigned a storage location. To store an unsigned number, use an alphameric literal.

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

Figure 7 shows how a numeric literal can be used in an imperative instruction. Assume the literal (+10) is assigned storage locations of 584 and 585, and INDEX is assigned 682. The symbolic instruction will cause the processor to produce a machine-language instruction (A 585 682) that causes +10 to be added to the contents of INDEX.

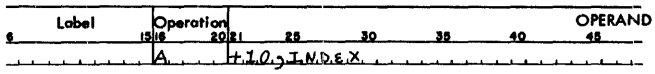


Figure 7. Numeric Literal

Alphameric Literals

Alphameric literals are written according to the following specifications:

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

Note: Only one alphameric literal may be written on one line of the coding sheet.

Figure 8 shows how an alphameric literal can be used in an imperative instruction. Assume that the literal JANUARY 28, 1961 is assigned a storage location of 906 and DATE is assigned 230. The machine language instruction (M 906 230) causes the literal JANUARY 28, 1961 to be moved to DATE.

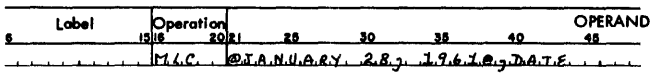


Figure 8. Alphameric Literal

Area-Defining Literals

With Autocoder, the programmer can instruct the processor to assign storage for a work area by using an area-defining literal. This literal defines the work area by specifying the name to be assigned to the work area and the number of core-storage positions needed. The programmer writes the area-defining literal in the operand field of any *one* source program instruction that uses the work area. All other instructions that use the work area require only the name of the area in the operand field.

A particular area-defining literal, or the name of the work area it defines, cannot be used in more than one program overlay.

To reserve storage for a working area by using an area-defining literal:

1. An area of 52 positions or less may be defined in any instruction that has, as an operand, the symbol which references it. The symbol can consist of as many as six alphameric characters, but the first character must be alphabetic. No special characters or blanks are allowed.
2. A # symbol (8-3 punch) must precede the number that specifies how many core-storage positions are needed for the work area. (Note the # symbol is represented in the Fortran character set as an = symbol.)

Figure 9 shows an imperative instruction with an area-defining literal. This entry causes the processor to allocate 6 storage locations for WKAREA. Six blanks will be loaded in storage at object-program load time by a DCW automatically produced by the processor. Assuming that AMOUNT is in storage location 796, and WKAREA

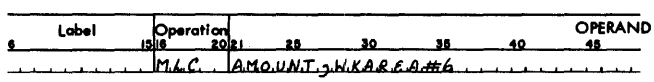
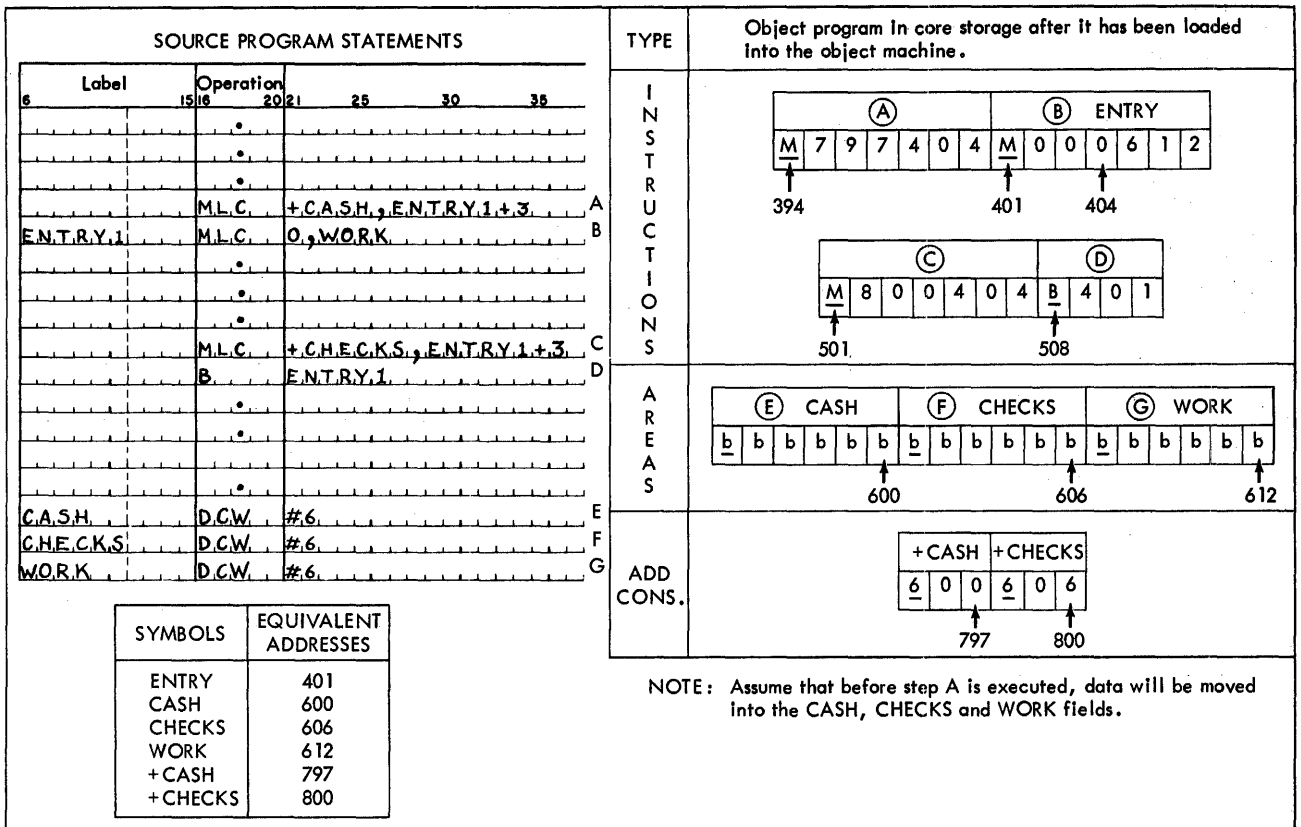


Figure 9. Area-defining Literal



SYMBOLS	EQUIVALENT ADDRESSES
ENTRY	401
CASH	600
CHECKS	606
WORK	612
+CASH	797
+CHECKS	800

NOTE: Assume that before step A is executed, data will be moved into the CASH, CHECKS and WORK fields.

PROGRAM STEP EXECUTED	OPERATION	CORE STORAGE BEFORE OPERATION	CORE STORAGE AFTER OPERATION
A	The address of CASH is moved to the A-address of B (ENTRY 1+3). B is thus modified.	B ENTRY 1: M 0 0 0 6 1 2 (401, 404)	B ENTRY 1: M 6 0 0 6 1 2 (401, 404)
B	The contents of CASH are moved to WORK.	E CASH 9 6 9 8 7 5 (600); G WORK 0 0 4 0 0 0 (612)	C CASH 9 6 9 8 7 5 (600); G WORK 9 6 9 8 7 5 (612)
C	The address of CHECKS is moved to the A-address of B (ENTRY 1+3). B is again modified.	B ENTRY 1: M 6 0 0 6 1 2 (401, 404)	B ENTRY 1: M 6 0 6 6 1 2 (401, 404)
D	Program branches back to execute B.	NO CHANGE	NO CHANGE
B_1	The contents of CHECKS are moved to WORK.	F CHECKS 6 0 7 8 9 2 (606); G WORK 9 6 9 8 7 5 (612)	F CHECKS 6 0 7 8 9 2 (606); G WORK 6 0 7 8 9 2 (612)

Figure 10. Address Constants

is in 596, the assembled machine-language instruction that moves AMOUNT to WKAREA is M 796 596. When using the loadable tape option, area-defining literals (#XX) of greater than 32 positions may not be correctly assembled. To insure correct results, use a DCW alphameric literal containing the desired number of blanks.

Address Constants

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

1. The symbol for an address constant can contain as many as six characters. The symbol must appear elsewhere in the program as a label.
2. A plus or minus sign must precede the symbol. If a plus sign is used, the address constant is the actual address that was assigned to the label by the processor. If a minus sign is used, the address constant is the 16,000's complement of the actual address.

When the processor encounters an address constant, it:

1. Assigns (in the object machine) a 3-position area that will contain the equivalent address of the symbol at object-program execution time.
2. Makes the address of the 3-position area equivalent to the symbol preceded by a plus or minus sign. For example, if CASH is the symbol whose address is needed as the address-constant, +CASH is the symbol that refers to the address of the equivalent address of CASH. If a minus sign precedes the symbol, for example, −CASH, the address constant is the 16,000's complement of the equivalent address of the symbol (CASH).
3. Generates a DCW card.

Note: Each time an address constant is encountered in a program or program section, it is assigned a core-storage address, and a DCW card is generated. If the address constant is used more than once in a source program, use a DCW statement to save core storage.

Figure 10 shows two address constants (+CASH and +CHECKS) used in a source program. It also shows the entries the processor makes in the object program, and the results when the instructions are executed in the object program. The programmer did not know which addresses would be assigned to CASH and CHECKS when he wrote the source-program statements. He did, however, write two instructions (A and C) that move these addresses into instruction B (ENTRY1). The address constants (+CASH and +CHECKS) caused the processor to store the addresses of CASH and CHECKS in the object machine, and to substitute the equivalent addresses of these constants in instructions A and C.

Character-adjusted and/or indexed address constants can be written symbolically. The address constant, not its equivalent address, is modified. Figure 11 shows an adjusted address constant. Assume that the equivalent addresses of ENTRY1 and +CASH are 401 and 797 respectively and that the address constant is 600. When the instruction (M 797 404) is executed, 12 will be added to the address constant, 600, and the resulting adjusted address constant, 612, will be moved to location 404.

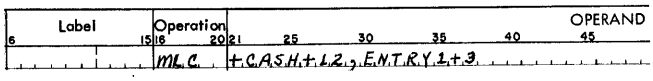


Figure 11. Address Constant with Address Adjustment

Indexing

If an object machine has the advanced-programming special feature (1401) or the indexing-and-store-address-register feature (1460), the source programmer can use the three 3-position index locations (registers) provided by the feature. The assigned core-storage addresses and index-register numbers are shown in Figure 12.

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

Figure 12. Index Locations and Associated Tag Bits

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

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

Note: If an index factor is to be used for address modification, the user should be sure that no zone bits appear in the units position if the system has only 4000 positions of core storage.

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

When the processor encounters an indexed operand, it puts tag bits over the tens position of the 3-character machine address assigned to the operand to specify which index register is to be used. The bit combinations and the registers they specify are shown in Figure 12.

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

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

Figure 13 shows an indexed imperative instruction that causes the contents of the location labeled TOTAL to be placed in an area labeled ACCUM as modified by the contents of index location 2. TOTAL is the label for location 3 1 0 1 and ACCUM is the label for location 1 4 0. The assembled machine-language instruction for this entry is: M A01 1M0. The M in the tens position of the B-address is a 4-punch with an 11-overpunch. The 11-overpunch is the B-bit tag for index location 2.

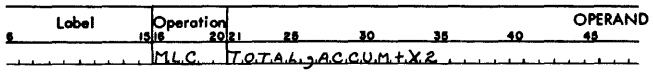


Figure 13. Autocoder Instruction with Symbolic Address and Indexing

Figure 14 shows an imperative instruction with address adjustment and indexing on a symbolic address. The processor will subtract 12 from the address which was assigned the label TOTAL. The effective address of the A-operand is the sum of TOTAL−12 plus the contents of index location 1 at program execution time. The assembled instruction (M ?Y9 140) will cause the contents of the effective address of TOTAL−12 +X1 to be placed in the location labeled ACCUM (assuming again that TOTAL is the label for location 3 1 0 1 and ACCUM is the label for location 1 4 0). The Y in the tens position of the A-address is an 8-punch with a zero overpunch. The zero punch is a tag for index location 1.

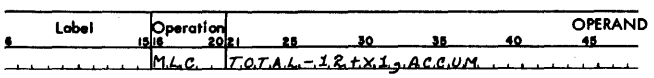


Figure 14. Autocoder Instruction with Address Adjustment and Indexing

Note: The address-adjustment factor cannot exceed ±999. Avoid using a negative address-adjustment factor that would result in an address less than zero. Because there is no *wrap-around* effect in address adjustment, the Autocoder processor will not assign the correct address.

Negative character adjustment in the A-operand is not correctly assembled, if the B-operand is a group-mark alphameric literal.

Declarative Operations

The IBM 1401 and IBM 1460 tape Autocoder provides six different declarative operations for reserving work areas and storing constants:

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

DCW—Define Constant with Word Mark

General Description: A DCW statement is used to enter a numeric, alphameric, or address constant with a word mark into a core-storage area.

The programmer:

1. Writes the operation code (DCW) in the operation field.

2. May write an actual address or a 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.

The processor:

1. Allocates a field in core storage that will be used to store the actual constant. If the DCW statement has a symbolic address in the label field, the processor assigns an address equal to the low-order position of this field.

2. Inserts the assigned address wherever the symbol in the label field appears in the operand of another symbolic program entry.

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

Numeric Constants

1. A numeric 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 numeric 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 numeric constant.
3. The maximum size of a numeric constant is 51 digits and a sign, or 52 digits with no sign.

Example: Figure 15 shows the number, +10, defined as a numeric constant. The address of the constant will be inserted in the object instruction whenever TEN appears in the operand field of another symbolic instruction.

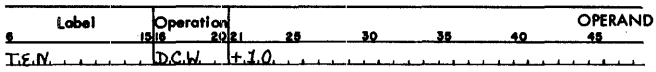


Figure 15. Numeric Constant Defined by a DCW Statement

Alphameric Constants

1. An alphameric constant must be preceded and followed by the @ symbol. Blanks and the @ symbol can appear within an alphameric constant, but the @ symbol cannot appear in a comment on the same line as an alphameric constant.
2. The alphameric constant can contain as many as 50 valid 1401 and 1460 characters.

Example: Figure 16 shows the alphameric constant, JANUARY 28, 1961, defined in a DCW statement. The address of the constant will be inserted in the object program instruction wherever DATE appears in the operand field of another symbolic program entry.

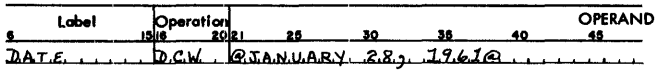


Figure 16. Alphameric Constant Defined by a DCW Statement

Blank Constants

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

Example: Figure 17 shows an 11-character blank field defined by a DCW statement. The address of this blank field will be inserted in an object program instruction whenever the symbol BLANK appears as the operand of another symbolic program entry.

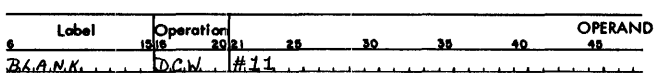


Figure 17. Blank Constant Defined by a DCW Statement

Address Constants

An address constant can be preceded by a plus or minus sign. If a plus sign or no sign is used, the constant is the actual machine language address of the field whose associated label is included in the operand. If a minus sign is used, the constant is the 16,000 complement of the actual machine address of that field.

Example: Figure 18 shows an address constant (the address of MANNO) defined by a DCW statement. The address of the address constant (MANNO) will be inserted in an object program instruction whenever SERIAL appears as the operand of another symbolic program entry.

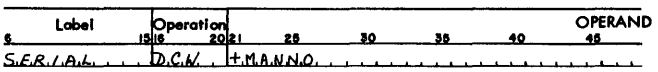


Figure 18. Address Constant Defined by a DCW Statement

An address constant that is defined by a DCW statement can be address-adjusted and indexed. The address adjustment and indexing refer to the address constant itself rather than to the address of the location of the address constant. If CASH is the symbolic address of a field, the equivalent address of CASH is indexed or address-adjusted rather than the equivalent address of +CASH.

Example: In Figure 19 the address constant (the equivalent address of CASH) is 600. Whenever TOTAL appears as the operand of another symbolic program entry, it will represent the equivalent address of a location that contains 604 (the adjusted address constant of CASH). See Figure 14.

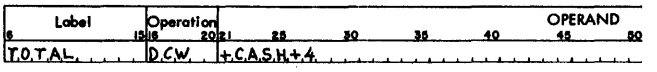


Figure 19. Address-Adjusted Address Constant Defined by a DCW Statement

DC—Define Constant (No Word Mark)

General Description: To load a constant without a word mark, write a DC statement like a DCW statement. The DC operation code is used in the operation field.

Example: Figure 20 shows TEN1 defined as a constant without a word mark.

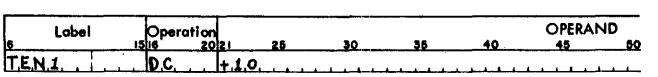


Figure 20. Constant Defined in a DC Statement

DS—Define Symbol

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

The programmer:

1. Writes the operation code (DS) in the operation field.

2. May write a symbolic address in the label field. Actual addresses cannot be used in the label field, and indexing is not permitted.

3. Writes a number in the operand field to indicate how many storage positions are to be bypassed.

The processor:

1. Assigns an actual address to the low-order position of the reserved area.

2. Inserts this address in the instruction wherever the symbol in the label field appears in the operand field of another symbolic program entry.

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

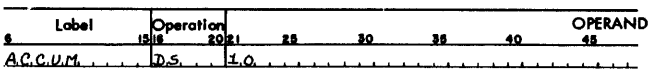


Figure 21. DS Statement

DSA—Define Symbol Address

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

The programmer:

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

2. May write in the label field, the symbol that will be used to make reference to the address constant.

3. Writes the actual address to be defined in the operand field. This address may be address-adjusted and indexed.

The processor:

1. Produces a constant containing the three-character machine address of the storage address written in the operand field.

2. Assigns this address constant an address in core storage and labels it using the symbol in the label field.

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

Example: To create and store an address constant for an actual address, the entry shown in Figure 22 is made.

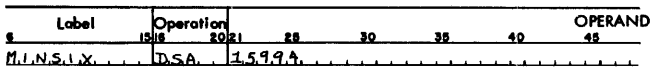


Figure 22. Defining the Address Constant of an Actual Address

Assume that the address assigned to the label (MINSIX) is 892. Storage locations 890, 891, and 892 will contain I 9 D (the three-character machine address of 15994). If index location 1 has been assigned the label INDEX 1, the instruction shown in Figure 23 will cause I 9 D to be moved to index location 1 (storage locations 087-089). The assembled machine language instruction for the statement shown in Figure 23 is M 892 089.

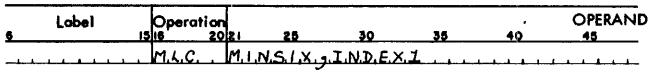


Figure 23. Moving the Address Constant to an Index Location

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

DA—Define Area

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

The programmer:

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

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

   b. May write an actual or symbolic address in the

label field. This address represents the high-order position of the entire area defined by the DA statement.

c. Indicates in the operand field the required size of the area in the form B X L. B is the number of identical areas to be defined, and L is the length of each area. For example, if four identical areas, each 100 characters long, are to be defined, the first entry in the operand field is 4 X 100 as shown in Figure 24. If only one area is to be defined, the first entry is 1 X 100.

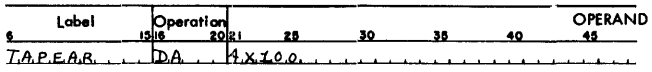


Figure 24. Four Areas Defined

Indexing: To index a DA entry, write a comma and the number of the index location (X1, X2, or X3) after the B X L indication in the operand field. When the DA HEADER specifies indexing, the equivalent addresses of all labels in subsequent DA entries will be indexed by the contents of the specified index location. The equivalent address of the entire defined area (B X L), represented by the label of the DA HEADER, is also indexed.

The tag bit that represents the specified index location will be inserted whenever the labels are used as operands in other symbolic program entries, *unless the operand is indexed.* For example, INAREA is defined by the DA HEADER shown in Figure 25. The second statement in Figure 25 is a defined field within INAREA. Thus the equivalent address of ACCUM has a tag bit (A-bit) over the tens position to indicate that it is to be indexed by the contents of index location 1.

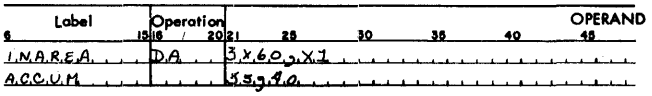


Figure 25. Indexing a DA Entry

However, a subsequent instruction in the program (Figure 26) indicates that ACCUM is to be indexed by the contents of index location 2. Because the instruction shown in Figure 26 is itself indexed, the processor will tag the equivalent address of ACCUM with a B-bit when it assembles the instruction for that statement only. Thus, the indexing in the instruction that uses the symbol ACCUM overrides the indexing prescribed by the DA HEADER statement. (Symbolic indexing is not permitted in a DA header statement.)

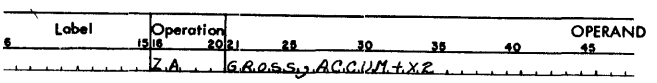


Figure 26. Overriding Previously Prescribed Indexing

To negate the effect of indexing on a field or subfield, put an X0 in the operand field of each instruction in which indexing is not wanted (Figure 27).

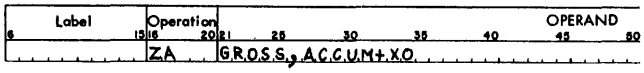


Figure 27. Negating the Effect of Indexing

Record Marks: Can be inserted to separate records in the defined area. The processor will cause a ≠ to be placed in storage immediately following each identically defined area if a ≠ follows the B X L entry in the operand field. B X L does not include an allowance for the record mark. For example, 2 X 100 will cause 200 positions to be reserved for the defined area, but 2 X 100, ≠ will cause 202 positions to be reserved as shown in Figure 28.

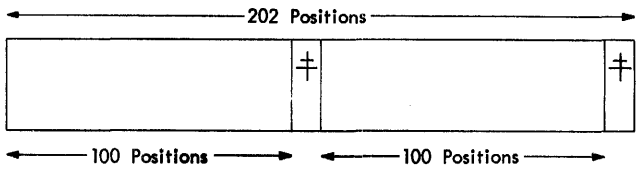


Figure 28. Record Marks

Group-Mark with Word-Marks: 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 as shown in Figure 29.

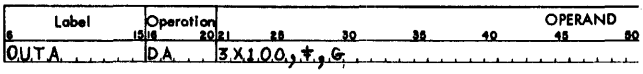


Figure 29. Group-Mark with Word-Mark

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

Subsequent DA Entries

The programmer:

1. Leaves the operation field blank.

2. May write a symbolic label in the label field. This label will have, as its equivalent address, the core-storage address of the field or subfield with which it is associated.

3. Specifies the relative location of a field within an area by writing two numbers in the operand field on

the same line as the label that identifies the field. The first location of the defined area is considered location 1. Write the high-order and low-order positions of the field beginning in column 21. Separate these two numbers by a comma.

In Figure 25 the ACCUM field is in relative positions 35-40. This means that the high-order position of the ACCUM field is to be associated with the 35th position of the defined area, and the low-order position, with the 40th position.

4. Specifies the location of a subfield (a field within a defined area) by writing, beginning in column 21, the number that represents the low-order position of the field whose label appears in the label field of the same line.

5. May list fields and subfields in any order in the DA entry. All positions within the defined area do not have to be included in the defined fields.

The processor:

1. Allocates an area in core storage equal to B X L plus positions for record marks and a group mark if they are specified in the heading line of the DA entry, and assigns actual addresses to the defined fields and subfields.

2. Inserts the assigned address of the high-order position of the entire defined area wherever the contents of the heading line label field appear as the operand of another symbolic program entry.

3. Inserts the assigned addresses of the low-order positions of fields and subfields in the place of symbols corresponding to the labels of the field-defining entries.

Result: At object-program load time:

1. If a , C appeared in the DA entry, the entire defined area is cleared.

2. A word mark is set in the high-order position of the entire defined area. If more than one area is defined (for example, 3 X 100), the high-order position of each area is identified by a word mark.

3. Word marks are set for field definition as noted previously.

4. 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 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 word-mark is to be placed in storage immediately following the third area.

The DA statement in Figure 30 defines three adjacent identical areas into which each block of three records will be read. It also defines the fields and subfields that are to receive the data listed. The notation 3 X 80 in the header line indicates that three consecutive areas of eighty locations each are to be reserved. The entire 240-location area can be referred to by its high-order label, RDAREA +X0. The G in the header line will cause a group-mark with word-mark to be placed in the 241st position. The reference to index location 2 in the header line indicates that the labels RDAREA, NAME, MANNO, DATE, GROSS, FICA, and MONTH, when referred to in symbolic instructions, will be indexed by index location 2.

Label	Operation	OPERAND
RDAREA	DA	3X80,X2,G
DATE		32,37
NAME		11,26
MANNO		4,8
GROSS		45,64
FICA		74,79
MONTH		35

Figure 30. DA Entry

The user can now, in his symbolic program, give an instruction to read data from tape into a storage area labeled RDAREA +X0. This causes a block of three data records to be placed in the 240 reserved core locations. As a result, the significant data is read into the appropriately labeled fields. This data can now be referred to via the labels DATE, MANNO, FICA, etc., and the user need not concern himself with actual machine addresses. In this example, the user begins by setting index locations 2 to zero. He then processes the significant data in the first record, increases index location 2 by eighty, and branches back to the first instruction of the particular routine. Because all labels defined by this DA statement are increased 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(s) with variable-length fields by defining all possible fields as subfields. In this case no word marks will be set in the area (except in the high-order position) but the programmer can control data transfer by setting word marks in the receiving fields.
2. If the length of the whole record(s) can also vary, the programmer should reserve an area equal to the largest possible record size.

EQU—Equate

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

The programmer:

1. Writes the operation code (EQU) in the operation field.

2. Writes a symbolic address for the new label in the label field.

3. Writes an actual or symbolic address in the operand field. This address can have indexing and address adjustment.

The processor:

1. Assigns to the label of the equate statement the same actual address that is assigned to the symbol in the operand field (with appropriate alteration if indexing and address adjustments are indicated).

2. Inserts this actual address wherever the label appears as the operand of another symbolic program entry.

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

Examples: Figure 31 shows the label INDIV equated to MANNO, which has been assigned storage location 1976. Whenever either MANNO or INDIV appear in a symbolic program, 1976 will be used as the actual address.

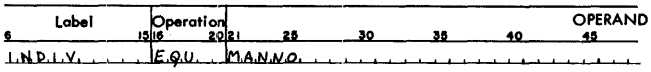


Figure 31. Equating Two Symbolic Addresses

Figure 32 shows an equate statement with address adjustment. If FICA is assigned location 890, WHTAX will be equated to FICA-10 (880). WHTAX now refers to a field whose units position is 880.

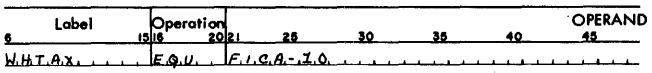


Figure 32. Address Adjustment in an EQU Statement

Figure 33 shows a label assigned to an actual address. Assume that an input card contains NETPAY in card columns 76-80. When this card is read into storage, the area locations 076-080 contain net pay. This field can be referred to as NETPAY if the EQU statement in Figure 33 is written in the source program.

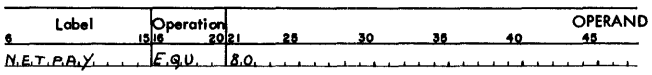


Figure 33. Assigning a Label to an Actual Address

Figure 34 shows how to index an operand in an EQU statement. With indexing, the symbol in the label field of the EQU statement is indexed by the same index location that is specified in the operand field of that EQU statement. However, if this symbol appears in the operand field of another symbolic program entry with another index code, the new index code overrides the index code in the EQU statement.

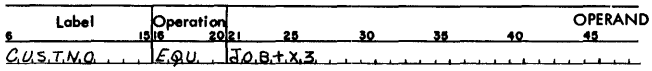


Figure 34. Indexing an EQU Statement

For example, in the statement shown in Figure 34 the equivalent address of JOB plus the contents of index location 3 is assigned to the label CUSTNO. Thus, if JOB+X3 is equal to 5H5, CUSTNO also has 5H5 as its equivalent address. However if CUSTNO+X1 or CUSTNO+X2 appears as the operand of another symbolic-program entry, the address inserted in its place will be 5Y5 or 5Q5, which specifies index location 1 or 2, respectively.

Figure 35 shows the symbol FIELDA equated to an asterisk address. The asterisk refers to the rightmost position of the last instruction or data whose location was assigned by the processor. Assume that this address is 698. FIELDA is now equal to 698.

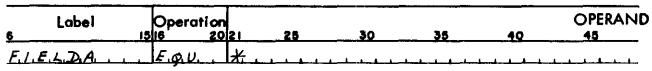


Figure 35. Equating with an * Operand

Figure 36 shows how a label can be assigned to an index location. Because the actual core-storage address of index location 1 in the IBM 1401 or the 1460 is 089, the EQU statement assigns the label INDEX1 to that index location. INDEX1 is now equal to 089. An index location so equated must still be coded X1, X2, or X3 when used to index an operand.

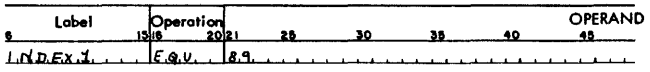


Figure 36. Assigning a Label to an Index Location

Figure 37 shows how a tape unit can be assigned a label. In this case, the programmer wishes to refer to tape 4 as INPUT, which is now equal to %U4.

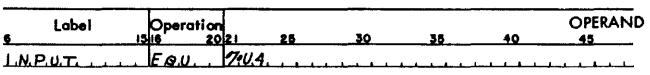


Figure 37. Assigning a Label to a Tape Unit

Imperative Operations

General Description: Autocoder imperative operations are direct commands to the object computer to act upon data, constants, auxiliary devices, or other instructions. These are the symbolic statements for the instructions to be executed in the object program. Most of the statements written in a source program will be imperative instructions. Although the Autocoder processor can assemble instructions with all the imperative operation code mnemonics that are shown in Figure 1, the programmer must keep in mind the particular special features and devices that will be included in the object machine that will be used to execute the program he is writing.

The programmer:

1. Writes the mnemonic operation code for the instruction in the operation field.

2. If the instruction is an entry point for a branch instruction elsewhere in the program or if the programmer wishes to make other reference to it, it must have a label. This label will be assigned an actual address equal to the address of the operation code of the assembled machine-language instruction. Thus, the programmer can use this label as the symbolic I-address of a branch instruction elsewhere in the program (see example, Figure 40).

3. Writes the symbolic address of the data, devices, or constants in the operand field. The first symbol will be used as the A- or I-address of the imperative instruction. If the instruction also requires a B-address, a comma is written following the first symbol and its address adjustment and/or indexing codes (if any); then the symbol for the B-address is written. If the instruction requires that a d-character be specified, a comma and the actual d-character follow the symbolic entries for the B-address or A/I-address if the B-address is not needed (see also *Address Types*).

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 1) all mnemonics that require that the d-character be included in the operand field are indicated by a +.

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

The programmer can:

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

b. Write the number of the tape unit in column 21 of the tape instruction. The assembled instruction for the symbolic entry shown in Figure 38 will cause a record to be written on tape unit 4 using the data beginning in a storage area labeled OUTPUT.

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

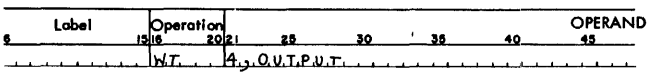


Figure 38. Write Tape

Compatibility with IBM 1410 Autocoder. To make the IBM 1401 and the IBM 1460 tape Autocoder language compatible with its IBM 1410 counterpart, five new mnemonic Op codes are provided that have the same function as five mnemonics presently available in SPS. When coding in Autocoder language, the programmer can use either mnemonic. These new

mnemonics are shown in Figure 39, together with their meanings and their SPS equivalents.

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

Figure 39. Alternate Move Mnemonics

The processor:

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

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

Examples: Figure 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. Assume that the address of READ is 596, and TEST is in 782. The assembled instruction is B 596 782 5.

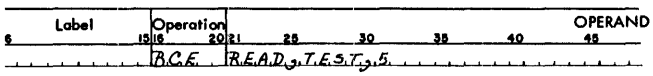


Figure 40. Branch if Character Equal

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. Assume that the address of OVFLO is 896. The assembled machine language instruction is B 896 Z.

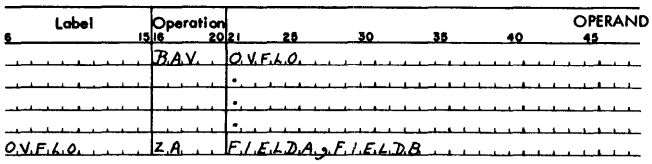


Figure 41. Branch if Arithmetic Overflow

Processor-Control Operations

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

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

JOB—Job

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

The programmer:

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

2. Writes in the operand field the indicative information to be printed in the heading line. This information may be any combination of valid 1401 and 1460 characters and appears in columns 21-72.

3. Writes in the identification field the information to be contained in the self-loading program deck or tape.

The processor:

1. Prints the information, the identification number from columns 76-80, and a page number from the JOB card on each page of the output listing. If there is no JOB card, the processor will generate one. In this case nothing will be printed in the heading line, except the page number.

2. Punches the identification number (cols. 76-80) in all condensed cards produced for the object program. If another JOB card (or cards) appear else-

where in the source program, the new identification number will be punched in subsequent condensed cards. This new JOB card will also cause the carriage to restore during listing, and the new information will appear in the heading line.

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

CTL—Control

General Description: The control statement is the second entry (card) in the source program deck. The user prepares this card to specify the size of the processing machine, the size of the object machine, the type of output he wishes, and the presence or absence of the Modify-Address feature. The modify address (MA) instruction is standard in IBM 1460 systems and in IBM 1401 systems with 8-, 12-, and 16-thousand positions of core storage. For an object machine *not* equipped with the MA feature, the Autocoder processor automatically assembles a routine to simulate the MODIFY-ADDRESS instruction.

The programmer:

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

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

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

Column 22 indicates the storage size of the object machine.

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

Column 23 indicates the type of Autocoder output desired.

Output	*Code*
Printed listing containing the symbolic source program and the machine-language object program.	Blank or 0
Printed listing and self-loading condensed program card deck.	1
Printed listing and self-loading program tape.	2
Printed listing, condensed card deck, and self-loading program tape.	3
Printed listing and one-instruction-per-card resequenced source deck.	4
Printed listing, condensed card deck and one-instruction-per-card resequenced source deck.	5
Printed listing, self-loading program tape, and one-instruction-per-card resequenced source deck.	6
All output options.	7
Error — list only	Any other code

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

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

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

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

If the CTL card is missing, the processor assumes that both the processing machine and the object machine have 4,000 positions of core storage. If the CTL card is included, lack of punching in column 21 and/or column 22 results in:

Column 21	*Column 22*	*Error message*	*Assembly*
blank	blank	Bad statement	No
blank	punched	Bad statement	No
punched	blank	Bad statement	Yes

If column 23 is left blank, or if the CTL card is missing, the processor provides a listing only.

ORG—Origin

General Description: An origin statement can be used by the programmer to specify a storage address at

which the processor should begin assigning locations to instructions, constants, and work areas.

The programmer:

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

2. Writes the symbolic, actual, blank, or asterisk address in the operand field. Symbolic or blank, or * addresses can have address adjustment (including X00) but indexing is *not* permitted in ORG statements.

3. If a symbolic label appears in the operand field of an ORG statement, it must appear in the label field elsewhere in the program sequence. It *need* not precede the ORG statement.

The processor:

1. Assigns addresses to instructions, constants, and to work areas as specified in the operand field of the ORG statement.

2. If there is no ORG statement preceding the first symbolic program entry, the processor automatically begins assigning storage locations at 333 (the first storage location following the fixed 1401 and 1460 read, punch, and print areas).

3. An ORG statement inserted at any point within the symbolic program causes the processor to assign subsequent addresses beginning at the address specified in the operand field of the new ORG statement.

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

Examples: Figure 42 shows an ORG statement with an actual address. The first symbolic program entry following this ORG statement will be assigned with storage location 500 as a reference point. (If the first entry is an instruction, the Op code position (I-address) of that instruction will be 500; if the first entry is a 5-character DCW, it will be assigned address 504, etc.)

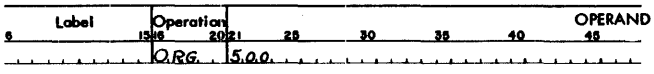


Figure 42. ORG Statement with an Actual Address

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 storage locations used to save this address. The processor will continue to assign addresses beginning at the actual address of START.

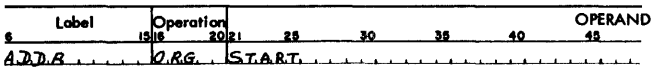


Figure 43. Saving the Address of the Last Storage Allocation

The programmer can insert another ORG statement later in the source program to direct the processor to begin assigning storage at ADDR (Figure 43).

If a symbolic label appears in the label field of an ORG or LTORG statement, it cannot be used in any other place except as the operand of another ORG or LTORG statement.

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

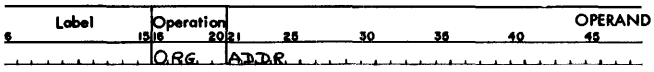


Figure 44. ORG Statement with a Symbolic Address

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 Step 3 *Processor*).

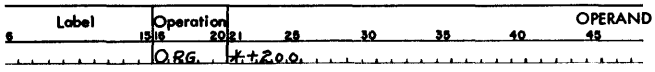


Figure 45. ORG Statement with an Asterisk Operand and Address Adjustment

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 525, the next instruction would have an address of 600.

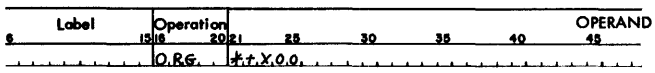


Figure 46. ORG Statement Advancing Address Assignment to the next Available Address that is a Multiple of 100

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

Figure 47 shows an ORG statement with a blank operand. When the processor encounters this statement, it begins assigning addresses to subsequent entries beginning with the first address (beyond 332) following the highest address assigned to other entries.

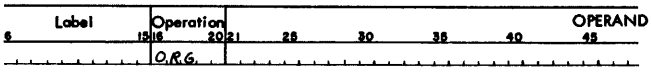


Figure 47. ORG Statement with a Blank Operand

A blank ORG statement that follows a DA statement will not be correctly assembled.

LTORG—Literal Origin

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

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

Example: Figure 48 shows how the programmer can direct the processor to begin assigning storage locations to literals and closed library routines.

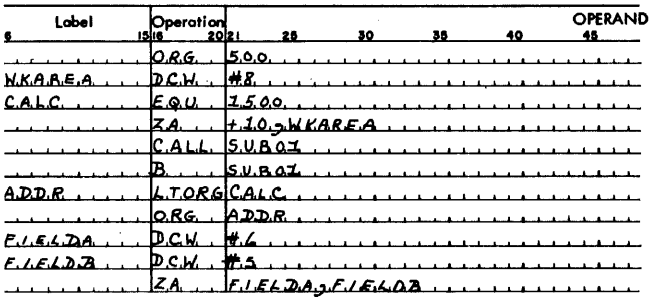


Figure 48. Using a LTORG Statement

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

EX—Execute

General Description: During the loading of the assembled machine-language program, the programmer may want to discontinue the loading process temporarily to execute a portion of the program just loaded. The EX statement is used for this purpose.

The programmer:

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

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

The processor:

1. Incorporates closed library routines, literals, and address constants in the program.

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

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.

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

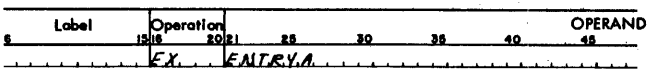


Figure 49. EX Statement

Figure 50 shows an example of this coding when the condensed card deck is used. The read area is cleared, word marks are set in 024, 056, 063 and 067; and a card is read with a branch to 056.

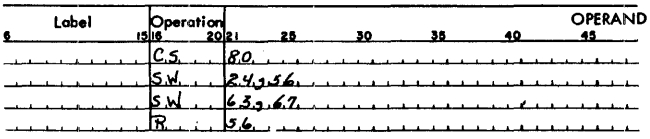


Figure 50. Re-Entry to the Load Routine

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

XFR—Transfer

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

END—End

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

The programmer:

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

The processor: Creates a CLEAR AND BRANCH instruction that is used as part of the loading data. The read area is cleared.

SFX—Suffix

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

A suffix statement with a blank operand can be used to stop the assignment of a suffix code.

The programmer:

1. Writes the mnemonic operation code (SFX) in the operation field.
2. Writes the character (which can be any valid 1401 and 1460 character) to be used for the suffix code in the operand field.

The processor:

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

Result: Each program section has unique labels.

Example: Figure 51 is an example of coding for a suffixing operation.

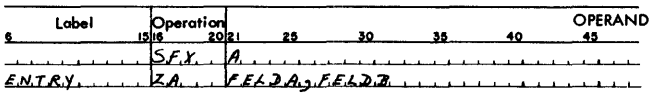


Figure 51. Specifying a SUFFIX Operation

ENT—Enter New Coding Mode

General Description: The 1401 and 1460 tape Autocoder processor accepts source programs coded in either free-form Autocoder language or in fixed-form SPS language. It is also possible to assemble a single program coded in a combination of the two languages. An ENT statement is used by the programmer to inform the processor that a change in coding form follows.

The programmer:

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

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

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

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

Example: Figures 52 and 53 are ENT statements to be used with Autocoder.

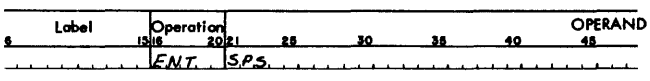


Figure 52. ENT Statement for Entering SPS mode

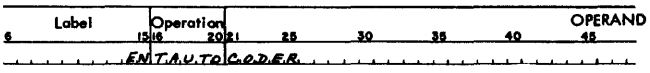


Figure 53. ENT Statement for Entering Autocoder Mode

ALTER—Alter

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

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

Additions

The programmer:

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

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

Example: The programmer wishes to insert two statements after the statement whose sequence number is 132. The three entries shown in Figure 54 are used.

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

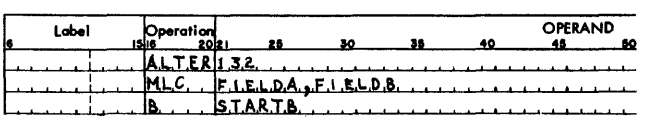


Figure 54. Adding Statements to an Assembled Object Program

Deletions

The programmer:

1. Writes the mnemonic operation code (ALTER) in the operation field of the ALTER statement.
2. Writes two numbers separated by commas in the operand field. The first of these numbers is the sequence number of the first statement to be deleted. The second number is the sequence number of the last statement to be deleted. If only one statement is to be deleted, only the sequence number is written twice in the operand field.

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

Example: If the programmer wishes to delete object program statements 192 through 203, he uses the entry shown in Figure 55.

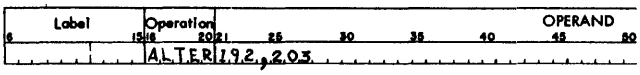


Figure 55. Deleting Statements from an Assembled Object Program

Substitutions

The programmer:

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

The processor:

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

Example: The entries shown in Figure 56 cause the processor to delete the statement whose alter number is 162 and add in its place the MLC and B instructions.

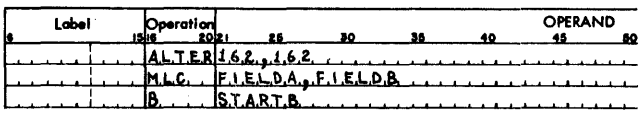


Figure 56. Substituting Statements in an Assembled Object Program

When reassembling with alterations, the instructions or constants inserted immediately following sequence number 103 or changes to sequence number 103, will not be assembled correctly. (Sequence number 103 is normally associated with the ORIGIN statement.)

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

Definition of Terms

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

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

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

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

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

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

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

Macro Operations

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

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

The Library Entry

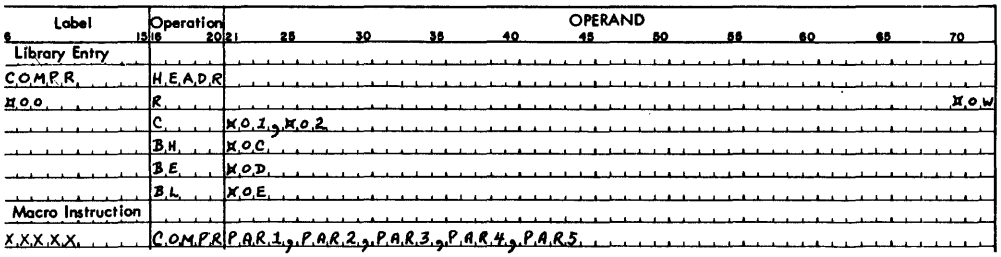
The library entry for the COMPR MACRO was created by writing a header statement and five model statements as shown in Figure 57.

HEADR—Header

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

The programmer:

1. Writes the operation code (HEADR) in the operation field.



Generated Symbolic Program Entries

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

Figure 57. Macro Operations

2. Writes the five-character label for the library routine in the label field. This label will be the same as the name that appears in the operation field of the associated macro instruction (except when either the CALL or INCLD macro is used). The first three characters must be unique (no two library entry labels may have the same first three characters).

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

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

Model Statements

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

The programmer:

1. Designs a general routine to perform a specific function when it is executed in the object program.

2. Writes the model statements as follows:

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

Example: Read a card (Figure 58).

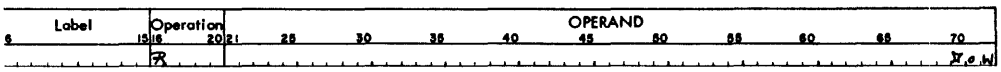


Figure 58. Model Statement for a Complete Instruction

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

Example: Insert parameters 01 and 02 specified by the COMPR macro instruction as shown in Figure 59.

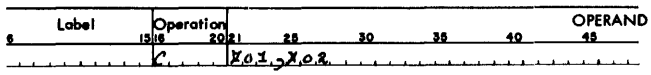


Figure 59. Model Statement Specifying that the First and Second Parameters be Present in the Associated Macro Instruction

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

Example: Insert parameter 03 in the following instruction if it is specified by the macro instruction. If parameter 03 does not appear in the macro instruc-

tion, the instruction shown in Figure 60 will be deleted from the object routine.

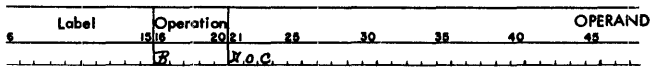


Figure 60. Model Statement for an Incomplete Instruction with Conditional Parameters

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

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

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

Example: Bypass the model statement shown in Figure 61 only if either parameter 04 or 05 is missing from the operand field of the macro instruction.

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

Example: Bypass the model statement shown in Figure 62 if either parameter 04 or 05 is present in the operand field of the macro instruction.

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

Example: Bypass the model statement shown in Figure 63 if parameter 04 is present or if parameter 05 is missing.

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

Labeling. If the model statement represents an instruction that is the entry point for a branch instruction elsewhere in the program, it must have a label. A □ 0 0 code in the first such model statement causes

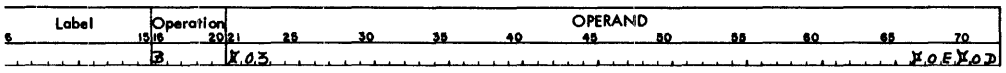


Figure 61. Condition Codes for Bypassing if Parameters are Missing from the Associated Macro Instruction

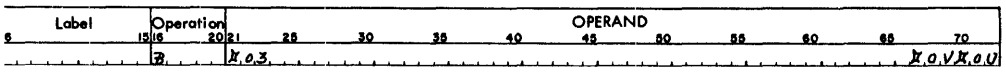


Figure 62. Condition Codes for Bypassing if Parameters are Present in the Associated Macro Instruction

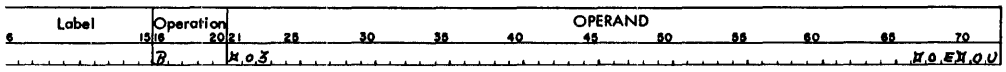
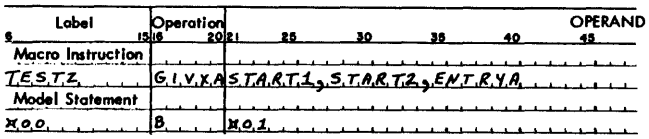


Figure 63. Condition Codes Combined

the contents of the label field of the macro instruction to be inserted in the label field of the generated symbolic entry as shown in Figure 64.



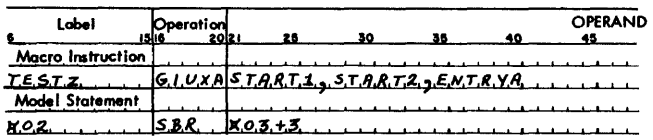
Generated Symbolic Program Entry

```
TESTZ    B      STRT1
```

Figure 64. Labeling

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

Example: Insert parameter 02 in the label field of the generated symbolic program entry as shown in Figure 65.



Generated Symbolic Program Entry

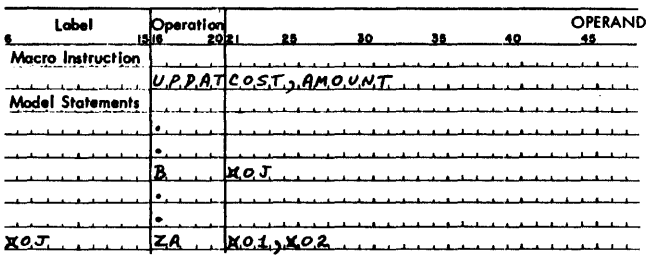
```
START2   SBR    ENTRYA+3
```

Figure 65. Additional External Label

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

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

Example: Use the generated symbolic address of (□ 0 J) as an operand for entry 3 and as the label for entry 6. UPDAT is the 23rd macro encountered in the source program (Figure 66).



Generated Symbolic Program Entries

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

Figure 66. Internal Labels

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 indexed, the leftmost indexed code in the assembled model statement takes precedence.

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

1. A model statement in the library routine for a macro instruction may not be another macro instruction, except the CALL, INCLD, or CHAIN macro (see *Call*).

2. Literal Origin, Ex and End statements cannot be used in library routines.

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

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

Figure 67 is a summary of the codes that can be used in the model statements of flexible library routines.

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

Figure 67. 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 an object program.

The programmer:

1. Writes, in the label field, the label that is to be substituted in the model statement that contains □ 0 0, if such a model statement appears in the library entry. If the □ 0 0 model statement is bypassed, the label is transferred to the next included statement.

2. Writes the name of the library routine in the operation field. This name must be the same five characters that appear in the label field of the header statement of the library entry.

3. Writes in the operand field the parameters that are to be used by the model statements required for the particular object routine desired as follows:

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

   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 unless the blanks appear between @ symbols. If parameters for a single macro instruction require more than one coding sheet line, the last parameter in each line must be followed immediately by a comma. The last parameter in a macro instruction must not be followed by a comma.

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

Figures 68, 69, and 70 show how parameters can be omitted. The hypothetical macro instruction called EXACT is used. EXACT can have as many as 9 parameters.

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

Result: The resulting program entries are merged with the source-program entries following the macro instruction. In the listing of the source and object programs (produced by the listing and condensed cards phase of Autocoder), the macro instruction is identified by a MACRO tag and the symbolic program entries generated by the processor are identified by GEN (Generated) tags.

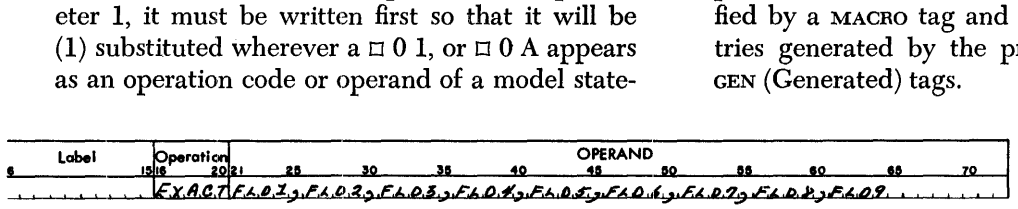


Figure 68. All Parameters are Present

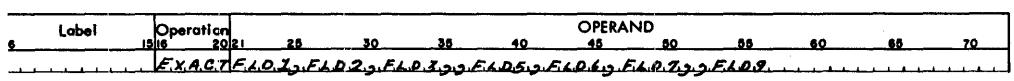


Figure 69. Parameters 4 and 8 are Missing

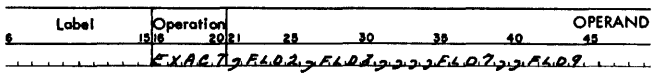


Figure 70. Parameters 1, 4, 5, 6, and 8 are Missing

Call Routines

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

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

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

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

   CALL routines are stored at the time and place where a Literal Origin, End, or Execute processor control statement is encountered. Duplicate symbols can occur if a CALL routine is used in more than one program overlay (if the same CALL routine is named in CALL macros that are separated by a Literal Origin or Execute statement). To prevent this possibility the Autocoder processor provides a Suffix (see *SFX*) operation. The programmer should use a suffix statement containing a new character in each program section.

2. The first instruction at each entry point in a CALL routine must store the contents of the B-address register (SBR) in an index location or in the last instruction executed in the CALL routine. This provides for re-entry at the proper place in the main routine after the CALL routine is executed.

3. All macro instruction operation codes except CALL, INCLD, and CHAIN are invalid in CALL routines. All other symbolic entries acceptable to Autocoder, except Literal, Origin, Execute, and End can be used. A CALL macro:

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

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

IBM-Supplied Macros

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

CALL Macro

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

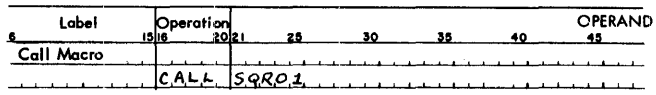
The programmer:

1. Writes the name of the macro (CALL) in the operation field.

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

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

the number whose square root is to be computed must be placed in a location labeled SQR02. The CALL macro is written as shown in Figure 71.



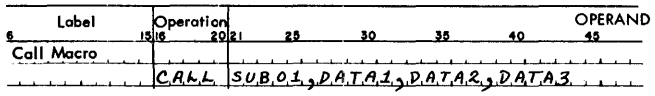
Generated Symbolic Program Entry

```
B     SQR01
```

Figure 71. CALL Statement Specifying That Data be in Specifically Labeled Areas of Storage

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

Example: Call a routine whose entry point is S U B 0 1 (Figure 72). The addresses of DATA 1, DATA 2, and DATA 3 are needed by the CALL routine.



Generated Symbolic Program Entries

```
B      SUB01
DCW    DATA1
       DATA2
       DATA3
```

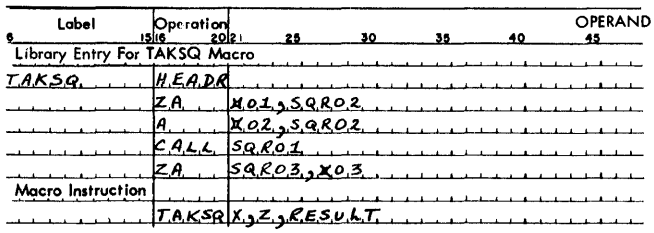
Figure 72. CALL Statement for a Routine with Arbitrary Data Storage Assignments

The processor:

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

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

Example: Assume that a library routine to compute the value of X + Z is associated with a regular macro instruction called TAKSQ. There is also a CALL routine in the library tape named SQR01, which calculates the square root of a number in a work area (SQR02) and places the answer in another work area (SQR03). The programmer can design a library entry for the TAKSQ macro that will provide linkage to the CALL routine as shown in Figure 73.

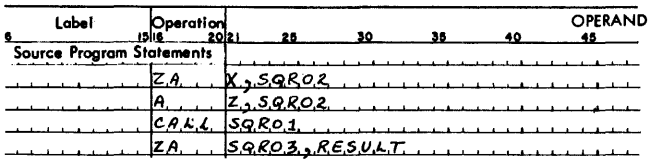


Generated Symbolic Program Entries

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

Figure 73. CALL Statement Within a Library Routine for a Macro Instruction

When the object routine is executed, X + Z will be stored in SQR02. Then the program will branch to the CALL routine where the square root of X + Z will be calculated and the result stored in SQR03. The last instruction in the SQR01 routine will cause an unconditional branch to the last instruction in the TAKSQ routine that puts the answer in an area labeled RESULT. This illustration is designed to show the combination of a regular macro and the CALL macro. The same result could be achieved by writing entries in the source program as shown in Figure 74.



Generated Symbolic Program Entries

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

Figure 74. Alternative Source Program Entries

INCLD Macro

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

Example: Figure 75 shows an INCLD statement that causes a library routine named SUB01 to be incorporated in the object program.

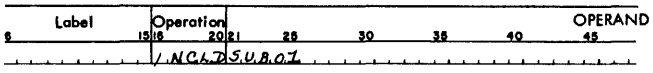


Figure 75. INCLD Statement

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

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.

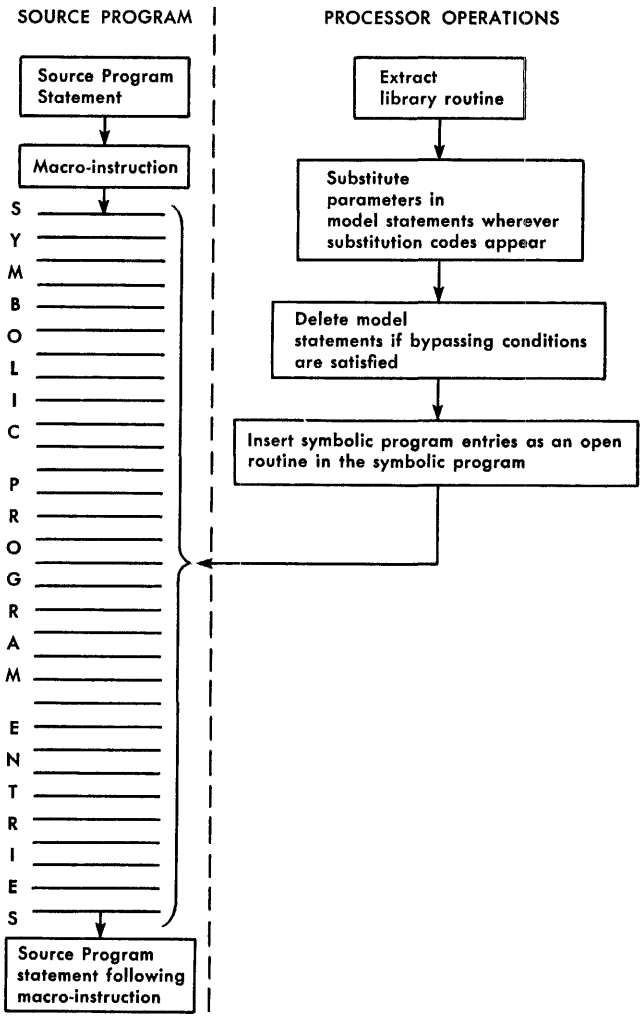
If CALL or INCLD are written *within* a library routine, only a single operand is permitted in the CALL or INCLD statement. This single operand is the name or entry point of the closed library routine. (See *Call Macro.*)

Macro Processing

Figures 76, 77, and 78 are diagrams showing the effects of the three different uses of library routines:

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

The symbolic programs that result from the processor actions described in Figures 76, 77, and 78 are later processed as though the user had inserted all the entries in the source program. (Symbolic entries are translated to machine-language instructions; constants cards are produced, etc.)



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

Figure 76. Macro Processing

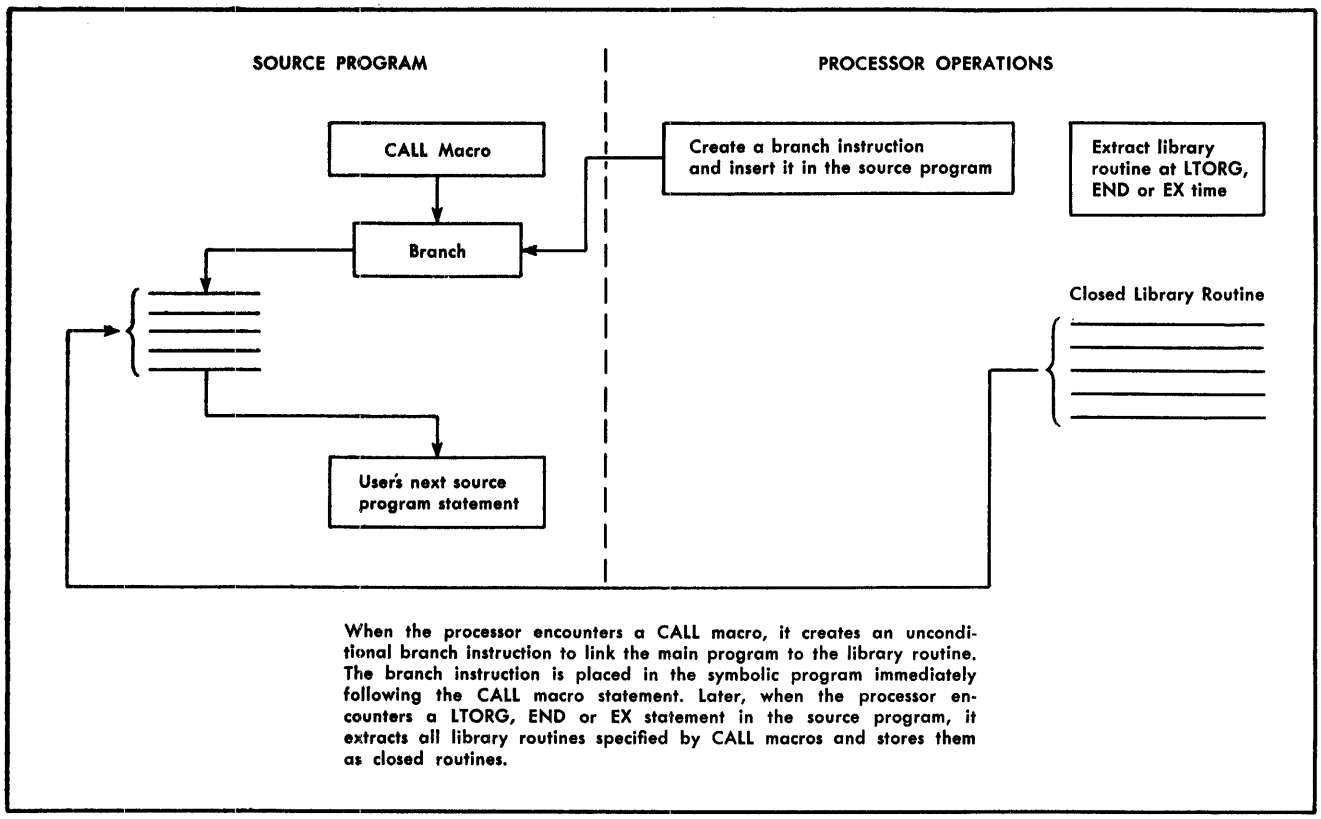


Figure 77. CALL Processing

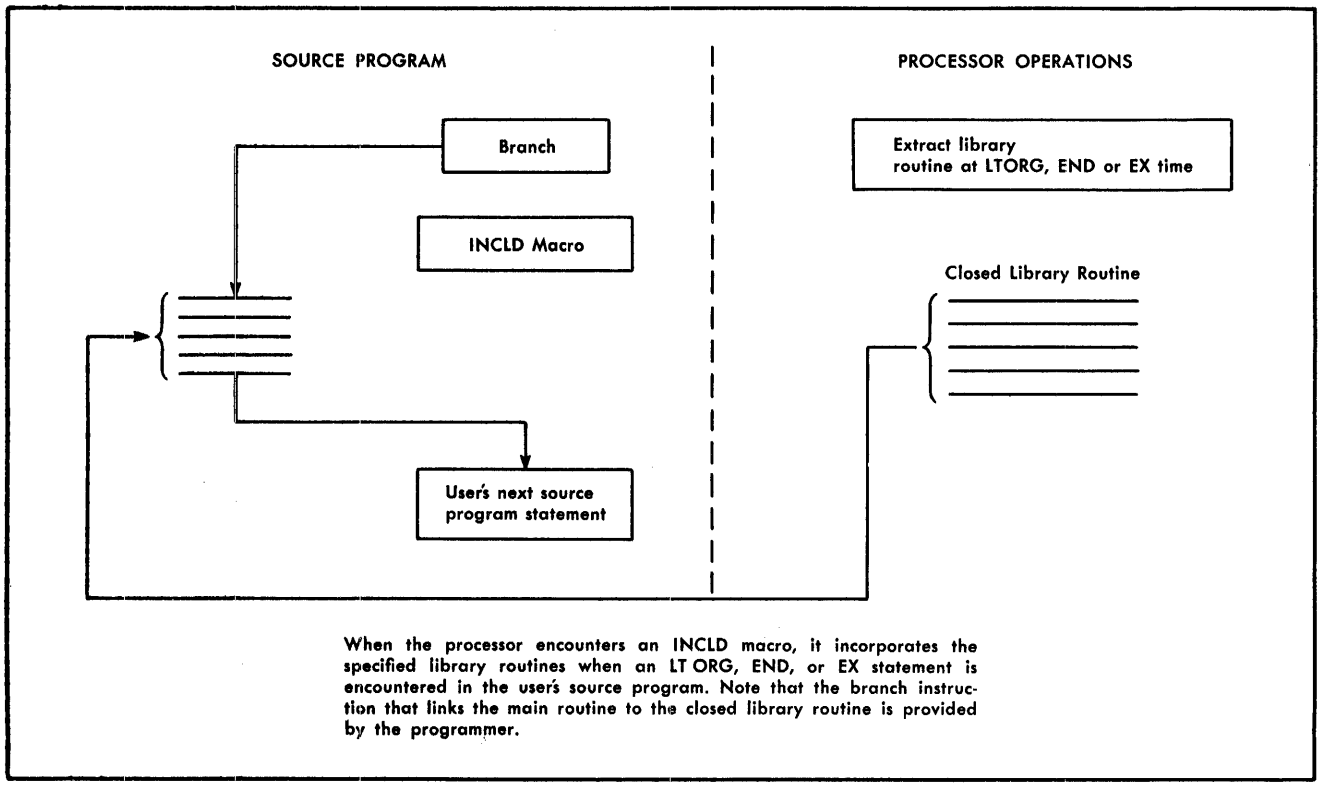


Figure 78. INCLD Processing

CHAIN Macro

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

The programmer:

1. Writes the instruction to be chained as usual.

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

The processor: Produces the desired number of additional operation codes.

Example: Figure 79 shows how an MLC statement can be chained five times.

Label	Operation	OPERAND
Source Program Entries		
	MLC	A,B,
	CHAIN	5

Generated Symbolic Program Entries

```
MLC
MLC
MLC
MLC
MLC
```

Figure 79. Chain Macro

OVLAY Macro—Card Overlay

General Description: This statement prepares storage and loads a new program section (overlay) from cards. The library routine for the OVLAY macro instruction is shown in Figure 80.

Label	Operation	OPERAND
OVLAY	HEADR	
□00	CS	80
	SW	24,56
	SW	63,67
	R	56
	DCW	#1

Figure 80. Library Routine for OVLAY Macro

The programmer: Writes the macro instruction as follows:

1. Writes the name of the macro (OVLAY) in the operation field.

2. Writes in the label field, the label to be inserted in the first statement in the library routine.

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

Example: At the end of a program section, the programmer places an OVLAY macro instruction in the source program as shown in Figure 81.

Label	Operation	OPERAND
Macro Instruction		
	OVLAY	

Generated Symbolic Program Entries

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

Figure 81. Using the OVLAY Macro

TOVLY Macro—Tape Overlay

General Description: The TOVLY macro prepares storage for and loads a new program section from magnetic tape. The library routine for the TOVLY macro instruction is shown in Figure 82.

Label	Operation	OPERAND
TOVLY	HEADR	
□0J	EQU	*+1
□00	CS	80
	RTW	1,1
	BER	*+5
	B	007
	BSP	1
	H	0,0
	B	□0J

Figure 82. Library Routine for TOVLAY Macro

The programmer: Writes the macro instruction as follows:

1. Writes the name of the macro (TOVLY) in the operation field.

2. Writes in the label field the label to be inserted in the first statement in the library routine.

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

Example: In the source program the programmer inserts the TOVLY macro as shown in Figure 83.

Label	Operation	OPERAND
Macro Instruction		
	TOVLY	

Generated Symbolic Program Entries

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

Figure 83. Tape Overlay

MA Macro—Modify Address

General Description: This library routine makes it possible to modify addresses with two addresses, or a single address when MA hardware is not available. The library entry is shown in Figure 84.

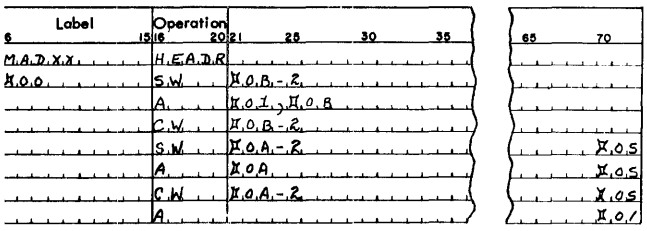


Figure 84. Library Entry for MA Macro

The programmer:

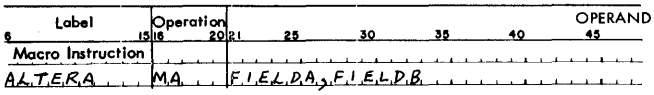
1. Writes the mnemonic operation code (MA) in the operation field.
2. May write a label in the label field.
3. Writes the macro instruction with one or two parameters.

The processor:

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

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

Examples: Figure 85 shows the MA macro instruction with parameters for both A- and B-addresses. The symbolic routine developed by the processor is also shown.



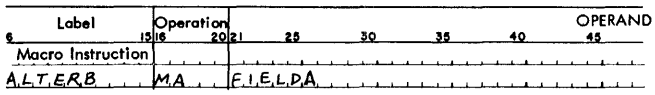
Generated Symbolic Program Entries

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

Figure 85. MA Macro with Two Parameters

Figure 86 shows the MA macro instruction with a parameter for the A-address only. The symbolic routine developed by the processor is also shown.

Note: An MA macro instruction with an asterisk in the B-operand will not be assembled correctly.



Generated Symbolic Program Entries

ALTERB	SW	FIELDA-2
	A	FIELDA
	CW	FIELDA-2

Figure 86. MA Macro with One Parameter

The System Tape

The Autocoder system tape contains the Autocoder processor and the library entries that can be extracted by macro instructions. All library routines must be stored on the system tape in alpha sequence. The IBM 1401 and 1460 standard collating sequence must be used.

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

DELET—Delete

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

The programmer:

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

The processor:

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

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

Examples: Figure 87 is a DELET statement that will cause the whole COMPR library routine to be removed from the library.

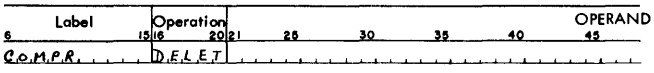


Figure 87. Deleting an Entire Library Routine

Figure 88 is a DELET statement that will cause the first model statement to be deleted from the COMPR library routine.

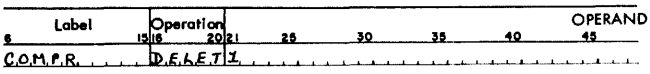


Figure 88. Deleting a Single Model Statement

Figure 89 is a DELET statement that will cause model statements 2, 3, 4, and 5 to be deleted.

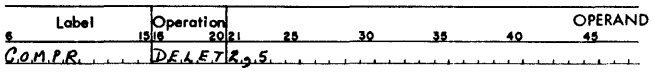


Figure 89. Deleting Multiple Model Statements

INSER—Insert

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

The programmer:

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

2. Writes the name of the library routine in the label field.

3. Writes the line number of the model statement after which the insertion is to be made. If two operands, separated by a comma, are written, the implied deletion will take place.

The processor:

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

2. Lists the action taken.

Result: The library tape contains the modified library routine.

Examples: Figure 90 is an INSER statement that will cause a library routine named COMPR to be inserted in the library tape.

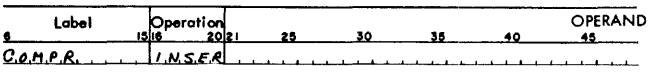


Figure 90. Inserting an Entire Library Routine

Figure 91 is an INSER statement that will cause new model statement 1 to be inserted in the COMPR library routine.

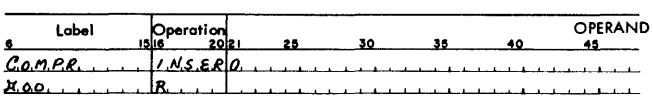


Figure 91. Inserting a Single Model Statement

Figure 92 is an INSER statement that will cause the first model statement that is presently in the library routine to be deleted and the model statement shown below to be inserted in its place.

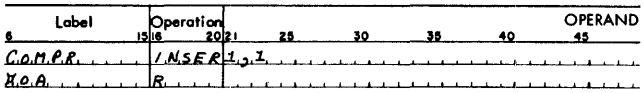


Figure 92. Substituting One Model Statement for Another

Figure 93 is an INSER statement that causes model statements 1 and 2 to be deleted and the model statements shown below to be inserted in their places.

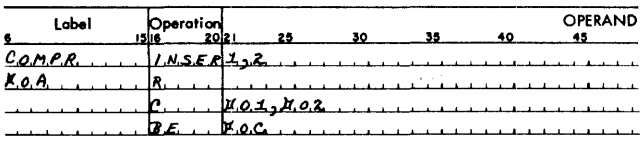


Figure 93. Substituting Multiple Model Statements

PRINT—Print Library Routine

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

The programmer:

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

2. Writes the name of the library routine in the label field.

The processor: Extracts and lists the library routine.

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

Example: The statement shown in Figure 94 causes the COMPR routine to be listed by the IBM 1403 printer.

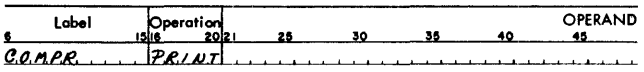


Figure 94. PRINT Statement

PUNCH—Print and Punch Library Routine

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

The programmer:

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

2. Writes the name of the library routine in the label field.

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

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

Example: The statement shown in Figure 95 causes the library routine called COMPR to be printed and punched.

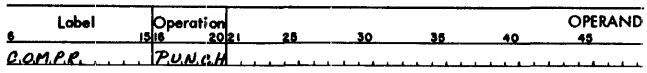


Figure 95. PUNCH Statement

Additional Language Specifications

Machine Language Coding

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

The programmer:

1. Writes in column 19 the actual machine language operation code for the instruction. Columns 16, 17, and 18 must be left blank.
2. Writes in column 20 the d-character in actual machine language. If no d-character is needed, column 20 must be left blank.
3. May write a label in the label field as described in *Imperative Operations,* Programmer Step 2.
4. Writes in the operand field a blank, actual, symbolic, or asterisk address, or a literal or address constant. The operand field must not contain the d-charter. The actual address of an input/output unit must be used unless the actual address has been equated to a symbol. For example,

Label	*Operation*	*Operand*	
	MR	%U1, INPUT	or
TAPE1	EQU	%U1	
	MR	TAPE1, INPUT	

are correct but,

	MR	1, INPUT	is incorrect

Disk Input/Output Instructions

The IBM 1401 and the IBM 1460 tape Autocoder includes mnemonic operation codes for IBM 1405 Disk Storage operations. When using these mnemonics, it is not necessary to specify the A-operand, and it is incorrect to use a comma to indicate that the A-operand is missing. Thus, the statement

Label	*Operation*	*Operand*
	RD	INPUTA

results in M %F1 xxx R, which reads a single sector without word marks from IBM 1405 Disk Storage into a core-storage area whose high-order address is xxx.

When coding programs that use the IBM 1311 Disk Storage Drive, or models 11, 12, 21, or 22 of the IBM 1301 Disk Storage unit, either of the following procedures can be used:

Label	*Operation*	*Operand*	
	MCW	%Fx, INPUTA, R	or
	MR	%Fx, INPUTA	

These same procedures can be used for the IBM 1405.

Auxiliary I/O Devices

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

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

Label	*Operation*	*Operand*	
	LCA	%TO, INPUTB, R	or
CONPR	EQU	%TO	
	LCA	CONPR, INPUTB, R	or
	LR	CONPR, INPUTB	or
	LR	%TO, INPUTB	

produce the actual machine language instruction

L %T0 xxx

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

Label	*Operation*	*Operand*	
	SS	L	or
	KL		

produce the actual machine language instruction K L.

3. For ENGAGE optical-character-reader statements:

Label	*Operation*	*Operand*	
	CU	%S2, E	or
OPTRD	EQU	%S2	
	CU	OPTRD, E	or
	UE	OPTRD	or
	UE	%S2	

produce the actual machine language instruction U %S2 E.

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

```
Label     Operation     Operand
          MCW           %D1, INPUTC, W          or
DTUNIT    EQU           %D1
          MCW           DTUNIT, INPUTC, W       or
               MW       DTUNIT, INPUTC          or
               MW       %D1, INPUTC
```

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

Processing Overlap

Special coding is required for all overlap operations because the A-address of the input/output instructions for these units contains the @ symbol. Autocoder recognizes the @ symbol as the leftmost and rightmost limits of an alphameric literal. To code overlap instructions for these units in Autocoder, we recommend that the programmer use the macro facility of Autocoder. A typical library routine and macro instruction to read a tape record in the overlap mode are:

```
Label     Operation     Operand
          RTOXX         3, INPUT
RTOXX     HEADR
          DCW           @M@U □01@
          DC            □02
          DC            @R@
```

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

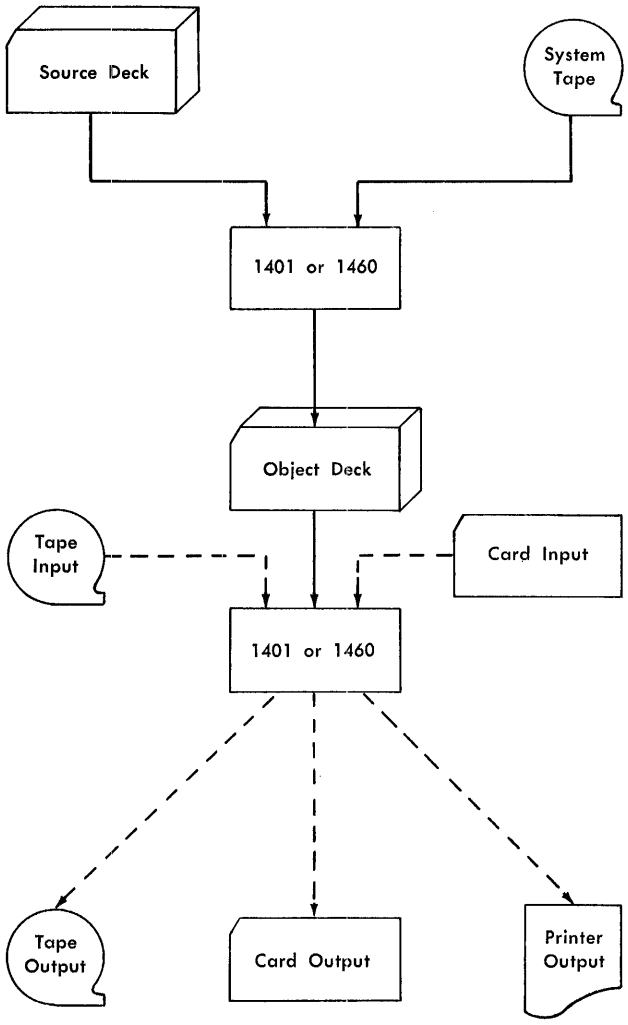


Figure 96. Tape Autocoder and Object Program Operations

The assembly of IBM 1401 and 1460 object programs from an *Autocoder* source program requires several distinct operations. First, a system card deck and listing are prepared from the *Autocoder* transmittal tape obtained from IBM. IBM supplies the 1401 *Autocoder* processor program on a reel of magnetic tape (called the *transmittal tape*). Requests for the *Autocoder* processor should be made through the local IBM sales office or sales representative. The *Autocoder* system tape is prepared from the card deck. Then, a librarian run may be performed to include user library routines (in cards) on the system tape. The assembly of a machine-language object program can then be performed, using the system tape and an *Autocoder* source program on magnetic tape or in punched-card form.

A listing of the assembled program with diagnostic messages is automatically provided by the processor. The following output options are also provided: new resequenced source deck, condensed object program card deck, loadable tape, and listing tape (for stacked multiple-program output).

In addition to program assembly, the processor allows printing and/or punching of all or part of the library routines on the system tape. This listing is useful when a new system tape containing additions, deletions, or modifications of the library routines is to be written.

Figure 96 illustrates the tape *Autocoder* and object program operations. Input/output operations are represented.

Writing the System Tape

To assemble a machine-language object program from an IBM 1401 *Autocoder* source program, the user must first prepare a system tape. The system tape is written in two or three steps:

1. A pre-system run that prints a listing of the *Autocoder* processor program and punches a system card deck.
2. A system run that generates the *Autocoder* system tape from the system card deck.
3. A library run that combines the tape written in step 2 with user library routines (in punched-card form) to produce a new system tape. Additional library runs can be performed to add, delete, or modify user routines in the library.

Pre-System Run

The 1401 *Autocoder* transmittal tape can be identified by an external label which reads: *1401 Autocoder-Listing and System Deck-Program #1401-AU-037.* The tape is high density, BCD mode, and contains a program at the beginning which will cause the *Autocoder* listing to be printed and the system deck to be punched into cards. *Autocoder* makes it necessary to generate a system deck to create the system tape, because the *Autocoder* system is maintained by IBM by the use of change cards to be inserted into the system deck (see *Change Cards).*

To retrieve the 1401 *Autocoder* Listing and the 1401 *Autocoder* System Deck from the *Autocoder* transmittal tape:

1. Mount the tape on Tape Unit 1 (high density).
2. Place paper in the printer. The listing uses approximately 210 sheets of paper (length per sheet is 11″).
3. Place at least 1500 blank cards in the punch.
4. Turn OFF the I/O check-stop switch.
5. Press the check-reset and start-reset keys.
6. Press the tape-load key.
7. A halt will occur at I-address 0365. To print the listing, press the start key. To bypass the listing, press the start-reset key, then the start key.
8. When the listing (or the bypass) is complete, a halt occurs at I-address 0505. To punch the system deck, press the start key. To rewind the tape instead, press the start-reset key, then the start key.
9. After the deck is punched, the message: 1401 AUTOCODER SYSTEM DECK PUNCHED is printed. The tape rewinds and a halt occurs at I-address 0514. The system deck is in pocket 4. To obtain an additional listing and/or system deck, return to step 5.

System Halts — Pre-System Run

The following additional halts may occur in the pre-system run:

I-Address	Reason	Restart Procedure
0497	Print error.	Press the start key to print the same data-line again.
0638	10 cumulative punch errors on one or more cards.	Press the start key to allow for 10 additional attempts.
0836	10 read errors, single-tape record.	1. Turn ON Sense Switch E and press the start key to retry the read operation an additional 10 times. 2. If the same halt occurs again, turn OFF Sense Switch E. Set the tape-select switch to D, and press the start key. 3. A halt will occur at I-address 0856. Scan storage for incorrect character(s) and correct it if possible. Set the tape-select switch back to N, set the I-address to 0805, and press the start key to process record. 4. If error is not detected in step 3, set the I-address to 0780 and press the start key to retry the read operation 10 times.

In all system halts, if the error is not corrected, the program should be restarted.

System Card Deck Format

The *Autocoder* system card deck is in a format which makes it possible to automatically generate a system tape. The cards are numbered sequentially, beginning with 0001 punched in columns 72-75. The deck is identified by punching in columns 76-80. Columns 76-77 contain the system program number for the processor, which is 37.

The *Autocoder* deck is divided into sections with the cards in each section identified by a number which is punched in columns 78-79. The first section of the deck, punched 11 in columns 78-79, contains a program that will generate the system tape. (This program itself is not written on the tape.) Cards on all succeeding sections in the system deck, with the exception of the last card (punched 99 in columns 78 and 79) are punched in the following format:

Column 78. The phase of the processor in which the card is contained (see *Autocoder Phases).*

Column 79. The section within each phase.

For example, the third section within Phase 7 is numbered 73 in columns 78-79.

A control card precedes each of sections 12 through 83 in the program. Its identification (columns 76-80) is that of the section it precedes. Columns 6-12 contain the word CONTROL. Columns 21-24 contain the high-order address of core storage (left-justified) where the section will be located during execution of the program section. Columns 28-33 contain PASS (1-8) representing the pass or phase in which the section is contained (see *Autocoder Phases).* The name of the section is punched on the control card beginning in column 38.

Change Cards

The *Autocoder* system is maintained with the use of change cards which are inserted into the system deck. The change cards are numbered sequentially in columns 72-75, beginning with C001. In addition, there is an 11 zone punch in column 80 of each card. When a modification is made to the *Autocoder* system, a set of change cards is sent to each user along with a modification letter containing a listing of the cards, an explanation of the changes, and instructions specifying where to insert the cards into the system deck. A new system tape must be generated when the system deck is modified (see *System Run*).

An *Autocoder* transmittal tape obtained from IBM contains all change cards up to and including the present modification level. When the system deck is punched, the change cards will be in their proper places. All modification letters to date are sent with each *Autocoder* transmittal tape.

Autocoder Listing Format

The *Autocoder* system tape listing is in the same format as the object program listing obtained after assembly (see *Assembly Listing).* Each program or section of the processor is printed beginning on a new page. Included in the listing, with identifying headings, are the system tape generation program and the passes of the processor.

System Run

Following the pre-system run, a system run is performed to obtain an *Autocoder* system tape.

1. Place the system deck in the card reader.
2. Mount a reel of tape (with file-protection ring) on tape unit 1.
3. Turn ON Sense Switch A.
4. Turn ON the I/O check-stop switch.
5. Press the check-reset, start-reset, and load keys. At the start of generation, the message: GENERATING 1401 AUTOCODER SYSTEM will be printed. All cards, with the exception of change cards (if they are present) are sequence-checked on columns 72-75.
6. After all cards have been processed, Tape 1 will rewind. The message: 1401 AUTOCODER SYSTEM GENERATED ON TAPE UNIT 1 will be printed, and the machine will halt at B-address 0142. At this point, file-protect the system tape.

System Halts — System Run

B-Address	Reason	Restart Procedure
0152	End of sequence checking (after a previous 0176 halt).	Check the system deck and put the cards in sequential order. Restart system run.
0161	Ten attempts to write Tape 1 correctly.	Replace the tape reel on Tape Unit 1. Restart system run.
0176	Sequence error in system card deck.	Press the start key to check the sequence of the balance of the deck.
0177	Missing control card in deck.	Check the deck and insert the necessary control card.

In all system halts, the program should be restarted if the error is not corrected.

System Tape Format

The 1401 *Autocoder* processor on tape consists of eight phases (Figure 97), each phase requiring a separate pass of the source program or partially assembled object program. The functions of these phases (or passes) are discussed in the *Autocoder Phases* section of this manual.

With the exception of section 12 of the first pass, which contains the *Autocoder* library, each section of the processor becomes a separate record on the system tape. As previously mentioned, the program (Section 11) which generates the system tape is not itself written on the tape. Therefore, section 12 becomes the first record on the system tape. An inter-record gap follows section 12. Then the following three built-in library routines are written in the form of card images: Modify Address — MA Macro; Card Overlay — OVLAY Macro; and Tape Overlay — TOVLY Macro. (Each instruction of each library routine is a separate record on tape.) Following the library is a tape mark, after which comes the balance of the system in individual tape records by section. A final tape mark comes at the end of Pass 8.

Record Numbers	Name	Card Identification (78-79)
1	Pass 1. Select Program	12
2-25	Library (MADXX, OVLAY, TOVLY)	12
—	Tape Mark (7-8)	—
26-29	Pass 1. Librarian	13-16
30-37	Pass 2. Macro Phase	21-28
38-42	Pass 3. Translator Phase	31-35
43-48	Pass 4. Relative Addressing Phase	41-46
49-50	Pass 5. Label Phase	51-52
51-52	Pass 6. Operand Phase	61-62
53-57	Pass 7. List, Condense Phase	71-75
58-60	Pass 8. Loadable Tape Phase	81-83
—	Tape Mark (7-8)	—

Figure 97. System Tape Layout

Librarian Run

After the system run, the *Autocoder* system tape is ready for program assembly. However, the user may wish to insert his own routines in the system library before he assembles an object program. This is accomplished by a librarian run, which involves Pass 1 of the processor.

Pass 1 of the processor includes a selection program and a librarian program. The selection program, which is at the beginning of Pass 1, analyzes sense-switch settings to determine whether a librarian run or an assembly run is to be performed. (If an assembly run is specified, the remainder of Pass 1 is bypassed and processing begins with Pass 2.)

The librarian program has three functions

1. Updating the system library
2. Copying the system tape
3. Displaying the system library

Updating the System Library

At the completion of the system run, the system library contains three routines: MADXX — Modify Address, OVLAY — Card Overlay, and TOVLY — Tape Overlay.

Subroutines in the library are on tape in the form of card images (each statement in the routine is an 80-character tape record). The first "card" of each routine contains the operation code HEADR and a label which is the name of the library routine. The HEADR statement of a routine is referenced as statement 0; the remaining (model) statements are referenced as 1 through n.

The name of a library routine must be five characters in length. The arrangement of the first three characters must be unique for each routine. Some names are not available to the user for naming his routines because they are used or are reserved for use by *Autocoder:*

CAL	MAD
CIIA	OPE
CLO	OVL
DCL	PUT
DIO	RDL
DTF	REL
FEO	SPA
GET	STA
INC	TOV

All library routines are stored on the tape in 1401 collating sequence by routine name; therefore, all cards used to update the library must be in the same order.

All routines to be inserted into the system library must begin with a HEADR statement containing the name of the routine in the label field. Each routine is preceded by an INSER statement and the routine(s) are placed in the card reader in collating sequence by routine name.

An updating librarian run may also be used to insert parts of routines and replace or delete entire routines or parts of user routines from an existing system library. Instructions to modify the library, using INSER and DELET statements, are given in the *Specifications* section of this publication. Cards specifying changes must be in collating sequence by library routine name. Only the library may be altered during an updating librarian run.

To update the system library:

1. Mount the old system tape on Tape Unit 1.
2. Ready a tape (with file-protection ring) on Tape Unit 6.
3. Turn ON (up) Sense Switches A and F.
4. Turn ON the I/O check-stop switch.
5. Place the deck of routines or changes, in collating sequence by name, in the card reader.
6. Press the check-reset and start-reset keys.
7. Press the tape-load key.

8. At the end of the run, the message: 1401 AUTO-CODER SYSTEM COPIED ON TAPE UNIT 6 will be printed. Tape reels on Units 1 and 6 will be rewound. The machine will halt at B-address 0122. Tape Unit 6 contains the updated system tape.
9. At this point, a copy run may be initiated by interchanging the tape-unit-select dial settings of Tape Unit 6 and Tape Unit 1 and obeying the rules for copying. (See *Copying the System Tape.)*

MESSAGES DURING UPDATING PROGRAM

A diagnostic listing is printed as changes are being made to the system library.

Error messages may occur during an updating run. Except for the occurrence of a card sequence error (which causes a programmed machine halt), the error message is printed, and the machine bypasses the card in error and continues processing.

MESSAGE	REASON
SUBROUTINE UNKNOWN	Routine to be modified is not in the system library.
BAD STATEMENT	An INSER or DELET card has two statement numbers in reverse order in the operand field.
STATEMENT DOES NOT EXIST	An INSER or DELET card references a statement not in the routine. For example, an instruction is given to delete statement 15 in a routine which contains fewer than 15 statements.
END OF LIBRARY REACHED	The end of the library is reached before a routine is found. This message will sometimes appear with a SUBROUTINE UNKNOWN error message.
INPUT CARDS OUT OF SEQUENCE — START OVER	The input cards were not in collating sequence by routine name. The machine halts at this condition. See *System Halts — Librarian Run.*

Copying the System Tape

The librarian provides the ability to copy the system tapes as many times as desired.

Besides copying the system, the librarian copy program allows space for reflective spots to be placed between copies of the system on the same tape. When this arrangement is used to copy the system many times on the same tape, if one system becomes unusable, the tape may be cut beyond the first reflective spot and the next system used.

To copy the system tape:

1. Mount the current system tape on Tape Unit 1.
2. Ready a tape (with file-protection ring) on Tape Unit 6.
3. Turn ON (up) Sense Switches C and F. Turn ON I/O check-stop switch.
4. Press the check-reset and start-reset keys.
5. Press the tape-load key.
6. At the end of the run, the message: 1401 AUTO-CODER SYSTEM COPIED ON TAPE UNIT 6 will be printed. In addition, a series of skip and blank tape instructions will have been executed to allow space for reflective spots (load points) to be placed by the user between copies of the system. After the skipping and blanking has taken place, the program will halt at B-address 0122. At this point the user may unload the tape (without rewinding), place a reflective spot at that point, and reload the tape, making sure that the new reflective spot is on the take-up reel. Press the start key to copy the system again.

Displaying the Library

Because additions, deletions, and modifications of user library routines can be made by an updating run, it is desirable to have a list of the library routines on the system tape. Also, when generating a new system tape from a modified card deck (see *Change Cards)* or when inserting selected library routines on a different system tape, it is necessary to have the routines in punched-card form.

The librarian makes it possible to obtain a listing of the library routine headers only or a complete or partial listing of the routines themselves. The entire library or part of the library can be punched into cards, with each HEADR card preceded by an appropriate INSER card.

A. To print the entire library:

1. Mount the system tape on Tape Unit 1.
2. Turn ON Sense Switches B, E, and F.
3. Turn ON I/O check-stop switch.
4. Press the tape-load key.
5. After all of the library routines have been printed, the message: END OF LIBRARY is printed. Tape 1 rewinds and the machine halts at B-address 0155.

B. To print a list of the library routine headers only:

1. Perform the preceding steps with Sense Switches B, E, D, and F ON.

C. To punch the entire library, together with appropriate INSER statements for each routine, into cards in *Autocoder* format:

1. Perform instruction steps (A) with Sense Switches B, E, F and G ON.
2. Besides punching, the entire library will also be printed.

D. To print or punch and print selected library routines:

1. Mount the system tape on Tape Unit 1.
2. Turn ON Sense Switches A, B, and F.
3. Turn ON I/O check-stop switch.
4. Press the check-reset and start-reset keys.
5. Place the appropriate PRINT and/or PUNCH cards in the card reader. These cards, described in the *Specifications* section of this publication, specify the routines to be printed or punched and printed.
6. Press the tape-load key.
7. At the end of the operation, the message: END OF LIBRARY is printed. Tape 1 rewinds and the machine halts at B-address 0155.

System Halts— Librarian Run

Figure 98 is a listing, by librarian function, of the halts that can occur during the librarian run. The information given for each halt consists of:

1. the librarian routine(s) in which the halt may occur (U = Updating, C = Copying, D = Displaying).
2. the B-address that can be displayed on the 1401 console when the halt occurs.
3. the message associated with the halt and/or the reason for the halt.
4. the procedures to be followed by the operator. Restart procedures for tape-read or write-error halts are given in *Tape Redundancy Procedures.*

Librarian Function	B-Address	Message and/or Reason	Procedure
U,C,D	0111	Second tape read redundancy halt.	Note the contents of the I-address register. Follow the tape read redundancy procedure in the text, starting with step 3.
U,C	0122	1401 AUTOCODER SYSTEM COPIED ON TAPE UNIT 6 - Message printed at the end-of-job halt for the Update and Copy Program.	
U	0133	Input cards out of sequence. A message accompanies this halt. (See Messages During Updating Program.)	Check input cards and put them in collating sequence by routine name. Restart Updating Program.
D	0144	Input cards do not contain a correct PRINT or PUNCH operation code.	Correct the cards and restart the Display Program.
D	0155	END OF LIBRARY — Message printed at the end-of-job halt for the Display Program.	
U,C,D	0166	Write redundancy, Tape 6.	Follow the procedure given in the text for tape write redundancy.
U,C,D	0191	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart the librarian run with another system tape.
U,C,D	0199	Read redundancy, Tape 1. Occurs in the Selection Program when the tape-load key is pressed.	Rewind system tape and press tape-load key again. If halt re-occurs, try another system tape or tape unit, restarting librarian run.

Figure 98. System Halts and/or Messages — Librarian Run

Program Assembly

After the system run (or librarian run, if the library has been modified), the *Autocoder* system is ready for program assembly. During an assembly run, the *Autocoder* processor produces a 1401 object program in condensed card format from a source program written in *Autocoder*. A listing of this program along with diagnostic and assembly messages is produced automatically by the processor. Other output options are selected by the user with the use of a CTL card in the source program deck.

To assemble an object program, the following steps are required:

If the source program is on cards,

1. Mount the system tape on Tape Unit 1. This tape must be rewound before starting, but the other tapes (4, 5, and 6) are rewound by the processor.
2. Place the source program card deck in the card reader. The first card in the deck should be the JOB card (used to provide identification); the second card must be a CTL card to specify

   a. output options desired by the user, and

   b. processor and object machine configurations.

   The last card in the deck must be an END card used by the processor as a signal that all source program entries have been read.
3. Mount working tapes on Tape Units 4, 5 and 6.
4. Mount a fifth tape on Tape Unit 3 if a listing tape is to be generated.
5. Turn ON Sense Switch A. All other sense switches must be OFF. (For machines without sense switches, this step may be bypassed. However, only initial assemblies using card input described here can be performed without sense switches.)
6. Turn ON I/O check-stop switch.
7. Press the check-reset, start-reset, and tape-load keys.
8. The message: PASS 3 COMPLETED is printed at the end of Pass 3 to indicate that Tape Unit 4 now contains the source program in free form for a re-assembly run (see *Reassembly Run).*
9. At the end of assembly when the listing and other requested output options are completed, the machine halts at B-address 0880 after the following messages are printed:

   END OF ASSEMBLY
   IF EXTRA OUTPUT DESIRED, SET SENSE SWITCH F ON, AND
   B ON FOR CONDENSED CARDS
   C ON FOR LOADABLE TAPE 6
   D ON FOR LISTING TAPE 3
   E ON TO SUPPRESS LISTING
   G ON FOR NEW SOURCE DECK
   AND PRESS START
   IF NO EXTRA OUTPUT DESIRED, PRESS START.

   If no extra output is requested or if the processor machine does not contain sense switches (applies to assembly from card input only) press the start key.
10. All tapes (except the listing tape, if it is used) are rewound by the processor to be ready for the next assembly. The machine halts at B-address 0889 after the following messages are printed:

    END OF JOB
    INPUT FOR REASSEMBLY ON TAPE UNIT 4

    If loadable tape has been produced, the message: LOADABLE TAPE ON TAPE UNIT 6 will be printed.

If the source program is on tape, it must be in 80-character records (exact image of the source program deck).

1. Mount the system tape on Tape Unit 1. This tape must be rewound before starting, but the other tapes (4, 5, and 6) are rewound by the processor.
2. Mount the source program tape on Tape Unit 4.
3. Mount working tapes on Tape Units 5 and 6.
4. Mount a fifth tape on Tape Unit 3 if a listing tape is to be generated.
5. Turn ON Sense Switches A and C. Follow the preceding steps 6 through 10.

Autocoder Phases

The 1401 *Autocoder* is an eight-phase processor, each phase of which requires a separate pass of the input:

Pass 1. Selection Program and Librarian. This phase is discussed in *Librarian Run.* The selection program at the beginning of the system tape determines whether an assembly or librarian run is needed. The librarian program maintains the *Autocoder* library and copies the system tape. It is bypassed in an assembly run.

Pass 2. Input, IOCS Processor, and Macro Phase. This is the first assembly phase. During it, the processor:

1. Reads source program from cards or tape (Tape 4).
2. Processes all macro-instructions, including IOCS instructions, needed from the library on the system tape.
3. Processes ALTER statements.
4. Writes symbolic statement records (including generated macro statements) on Tape 6.

Pass 3. Translator Phase. During this phase, the processor:

1. Reads statement records from Tape 6.
2. Translates any fixed form (SPS) information present in the program to free-form (see *Conversion of SPS Statements).*
3. Translates mnemonic operation codes to machine language, and checks for validity.
4. Assigns sequence numbers to free-form records.
5. Writes 86-character free-form statement records on Tape 4. After this pass, Tape 4 is in condition for a re-assembly run.

Pass 4. Relative Addressing Phase. During this phase, the processor:

1. Reads statement records from Tape 4.
2. Assigns relative addresses to all data to be loaded into storage at object time.
3. Converts all literals, including duplicates, to DCW's and merges them into the source program when a LTORG, EX, or END is encountered.
4. Converts free-form statements to fixed-form.
5. Generates DC and DCW statements when a DA specifies:
   a. record marks are to be placed between records; or
   b. a group mark is to be placed after the area; or
   c. the area is to be cleared at the time the object program is loaded.
6. Counts number of labels and stores total for Pass 5.
7. Writes blocked statement records on Tape 5 (see *Blocking).*

Pass 5. Label Phase. During this phase, the processor:

1. Reads blocked records from Tape 5.
2. Assigns actual addresses to instructions and constants.
3. Generates a table of labels in storage, each entry containing the label and its true address.
4. Partially processes ORG, LTORG, and EQU statements.
5. Eliminates duplicate literals from the object program.
6. Writes blocked statement records on Tape 6.

Pass 6. Operand Phase. During this phase, the processor:

1. Processes all operands, looking up symbolic operands on the table.
2. Assigns addresses to partially processed ORG, LTORG, and EQU statements. If the symbol which defines an ORG, LTORG, or EQU statement appears after the statement, the processor must execute at least one more iteration. (Rearrangement of the source deck may reduce the number of iterations of this type.)
3. Lists, at the end of the phase, the entire symbol table and all unreferenced labels.
4. Writes blocked assembled program statement records on Tape 5.
5. If the total number of labels in the program exceeds the maximum number that can be processed in one iteration of Passes 5 and 6 (see *Symbol Table)* or if there are unprocessed ORG, LTORG, or EQU statements, additional iterations are required and control is transferred to Pass 5.

Pass 7. Listing and Condensed Cards Phase. During this phase, the processor:

1. Generates from Tapes 4 and 5 a combined listing of the source and object program along with source program error messages to the right of the statements in error.

If requested by the CTL card, the processor:

2. Punches a condensed self-loading object program deck.
3. Produces, if a fifth tape is available, Tape 3 containing the listing and condensed cards.

Pass 8. Loadable Tape and New Source Deck Phase. During this phase, the processor (if requested by the control card):

1. Produces, from Tape 5, a loadable tape (Tape 6) containing the assembled program.
2. Punches, from Tape 4, a new resequenced source program deck.

At the end of Phase 8 a message is printed indicating the user's options to select additional output options by sense-switch control. If no additional output is requested, pressing the start key causes the processor to rewind all tapes (except the listing tape 3, if it is used) and to print end-of-job messages.

BLOCKING

Statement records are processed one-per-block (unblocked) in passes 1-3. Pass 4 does the initial blocking, according to the following format.

Machine Size	Block Length	Record Length
4,000	1	80
8,000	5	80
12,000	10	80
16,000	10	80

In passes 5-8, the statements are processed in blocked format.

SYMBOL TABLE

The number of labels that can be processed during one iteration of Passes 5 and 6 depends upon the size of the processor machine. If unprocessable symbolic origins or equates are encountered, the maximum number of labels may not be processed.

Machine Size	Maximum Number of Labels
4,000	150
8,000	510
12,000	870
16,000	1,270

ALLOWABLE BLANKS IN AUTOCODER STATEMENTS

When coding in *Autocoder*, any number of blanks may be used between the A- and B-operands if the comma (,) immediately follows the A-operand. One blank only is tolerated between the B-operand and the d-character, if the comma (,) immediately follows the B-operand.

CONVERSION OF SPS STATEMENTS

Source program statements in SPS are processed by *Autocoder* if they are preceded by an Enter (ENT) SPS statement and succeeded by an Enter (ENT) AUTOCODER statement. These fixed-form statements are converted to *Autocoder* format in Pass 3 of the assembly run.

DS statements in SPS, which are used to assign labels to a symbolic or actual address, are given the *Autocoder* mnemonic operation code EQU. DCW, DC, and DSA statements in SPS, which have labels and actual addresses, are expanded into two statements in *Autocoder* (Figure 99). Five-character branch instructions are converted to unique mnemonic operation codes where these codes exist.

Autocoder Output

The 1401 *Autocoder* system provides six types of output.

1. Diagnostic messages during assembly.
2. Listing of source program and assembled program with error codes.

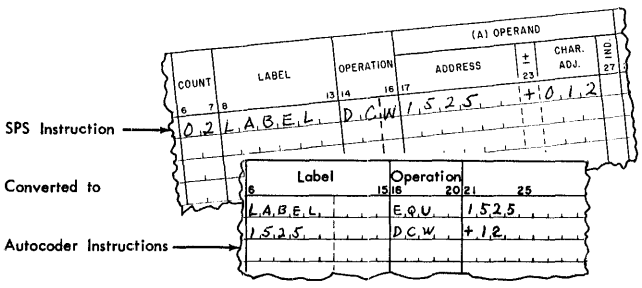


Figure 99. SPS Conversion Example

3. Condensed assembled program card deck.
4. Loadable tape containing assembled program.
5. New resequenced source deck in *Autocoder* language format.
6. Tape containing the assembly listing and the assembled program in condensed card format (only if the fifth tape is available).

The assembly messages and listing with error codes (1 and 2) are produced automatically by the processor during every assembly run.

The diagnostic assembly messages are explained in *System Halts* and/or *Messages-Program Assembly*.

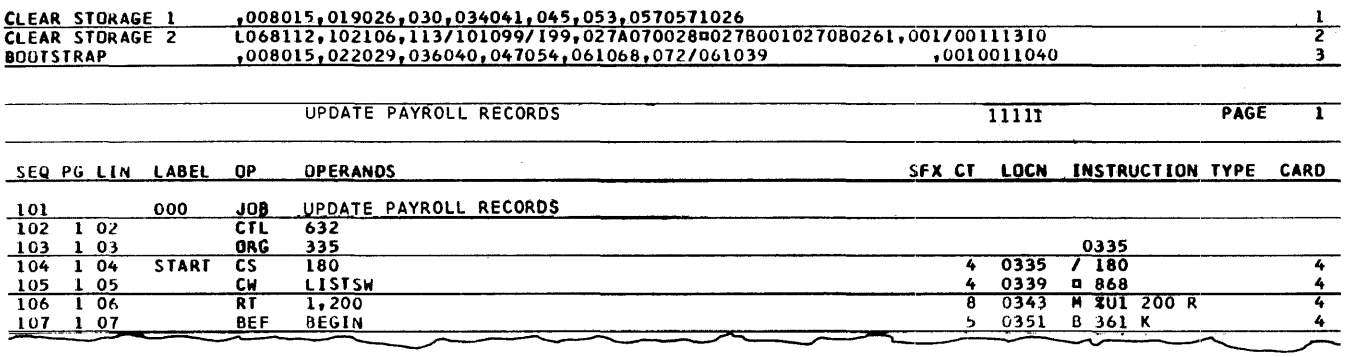
The other output options must be requested on the CTL card in the source program deck or, for extra output, as sense-switch options after assembly. Machines without sense switches can only obtain output requested on the control card.

Assembly Listing (with Error Codes)

On the top of the first page of the listing are printed the images of the clear storage and bootstrap cards generated by the processor for loading the object program. At the extreme left are the legends CLEAR STORAGE 1, CLEAR STORAGE 2, and BOOTSTRAP to identify the cards. At the extreme right are numbers 1, 2, and 3 to indicate these are the first three cards of the condensed deck.

Below the clear storage and bootstrap card images on the first page of the listing, and at the top on every succeeding page are two heading lines followed by the detailed source-object program listing. The first heading line contains the page number on the right, and the contents of the operand and identification fields of the JOB card on the left. The second heading line identifies each column of information in the detail section of the listing (Figure 100).

Heading	Source Program Card Columns	Meaning
SEQ		The instruction sequence number, starting with 101 (used for reassembly changes — see *Reassembly Run*. Sequence numbers, but no page and line numbers, appear for statements generated by a macro-instruction.
PG	1-2	Page number of the source program statement.
LIN	3-5	Line number of the source program statement. For literals generated every time they are used, literal-address constants, and area-defining literals, the sequence number of the source



```
CLEAR STORAGE 1      ,008015,019026,030,034041,045,053,0570571026                                    1
CLEAR STORAGE 2      L068112,102106,113/101099/I99,027A070028¤027B0010270B0261,001/00111310           2
BOOTSTRAP            ,008015,022029,036040,047054,061068,072/061039          ,0010011040             3

                          UPDATE PAYROLL RECORDS                          11111                PAGE   1

SEQ PG LIN  LABEL  OP    OPERANDS                                  SFX CT  LOCN  INSTRUCTION TYPE  CARD

101         000    JOB   UPDATE PAYROLL RECORDS
102  1 02          CTL   632
103  1 03          ORG   335                                                     0335
104  1 04   START  CS    180                                          4  0335  / 180                4
105  1 05          CW    LISTSW                                       4  0339  ¤ 868                4
106  1 06          RT    1,200                                        8  0343  M %U1 200 R          4
107  1 07          BEF   BEGIN                                        5  0351  B 361 K              4
```

Figure 100. Autocoder Assembly Listing

Heading	Source Program Card Columns	Meaning
		statement is printed on the line of the generated statement in the PG LIN area.
LABEL	6-11	Source Program Statement.
OP	16-20	Source Program Statement.
OPERANDS	21-72	Source Program Statement.
SFX		Suffix (SFX) character, if any.
CT		Count. The number of characters to be loaded into core storage at object time.
LOCN		The location at which the assembled instruction or data field will be loaded. (High-order position is given for instructions and DA header; low-order position is given for DCW, DC, DS, EQU, DSA, DA fields and subfields; and the label address, if any, is given for ORG and LTORG statements.)
INSTRUCTION		The assembled instruction. For ORG and LTORG statements this field contains the address of the origin. It contains the low-order position of the area generated by a DA statement.
TYPE		This field contains an abbreviation indicating the type of an *Autocoder* or IOCS-generated statement. It is blank for non-generated statements. ADCON. Address Constant Literal AREA. Area-defining literal FIELD. DA field GEN. Macro-generated statement GENIO. Generated IOCS statement GMARK. DA Group Mark IOCS. DIOCS and DTF statements LIT. Alphabetic or Numeric literal MACRO. Macro Statement RMARK. DA Record Mark SBFLD. DA Subfield
CARD		The condensed card number on which the data appears. This legend is always printed, even when a condensed card deck is not requested as output.

The remaining field of the printed listing, the error code field, does not have a heading. This field is used to print an indication if the statement on the same line has an error.

Error Code	Meaning
ADDR	Address. The data would be loaded into the read area (locations 001-080). This might result in an error when loading the object program.
LABEL	The label is doubly defined (i.e., another statement defines the same label).
MACRO ERROR	The statement indicates an incorrect macro instruction.
NO BXL	The length of a DA was not specified correctly. If this error occurs, BXL is assumed to be 1X1.
OP	Invalid mnemonic operation code or blank operation code following an imperative operation.
OVERCALL	The maximum number of CALL statements allowed for this overlay (58) has been exceeded.
SYM	Unprocessable operand, usually an undefined symbolic operand. In the assembled instruction, the address appears as # # #.
UNDEF ORG	Undefined symbolic ORG or LTORG.
BAD STATEMENT	The statement on the same line was either unprocessable or may have been processed incorrectly because of an input error.

Statements which are marked BAD STATEMENT but are processed anyway include:

1. Those whose last operand is not followed by two blanks.

2. DA fields or subfields whose parameters exceed the record length size specified in the header.
3. Instructions whose A-operand is followed by a comma, but which contain no B-operand.
4. Symbolic addresses greater than six characters. In this case, the operand in error is marked undefined. A common cause of this error is a missing comma between operands. This case will cause the processor to treat the entire field as the A-operand.
5. A CTL card punched in column 21, but unpunched in column 22.

Statements which contain the error message and are unprocessed include:

1. Alphameric literals or constants that have no ending @ sign.
2. A control card with illegal codes.
3. An area-defining DCW (for example, DCW #53) whose length exceeds 52.
4. An operand which apparently extends beyond column 72.
5. A constant whose length is zero (for example, DCW @@).
6. A DA field whose low-order location is specified as being a lower value than its high-order limit (for example, 20, 19).

Condensed Assembled Program Deck

The first three cards of the object program deck are generated by the processor. Two clear core storage cards clear all of core storage, and a bootstrap card sets word marks in the read area before the object program is loaded.

The remainder of the cards contain assembled program instructions and load instructions in condensed card format. There may be as many as seven instructions or constants on each card. The card format is:

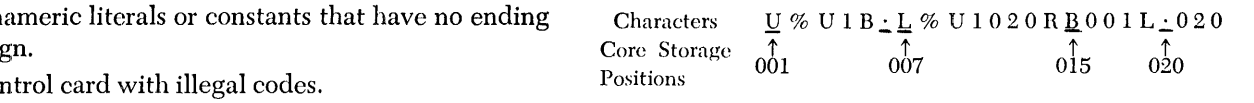
CARD COLUMNS	CONTENTS
1-39	The data (instructions or constants) to be loaded into storage.
40-46	Load instruction; instructions necessary to load the data into storage.
47-67	Three 7-character set-word-mark instructions (or one clear-word-mark and two set-word-mark instructions for cards containing DC's). These instructions set the word marks that define the separate fields in the block of storage being loaded.
68-71	1040. This is an instruction which will cause the 1401 to read a card and branch to location 040.
72-75	Card Number. The card deck, beginning with the first clear storage card, is numbered sequentially in these columns.
76-80	Identification. The identification in columns 76-80 of the JOB card appears in all cards in the condensed deck. Each new JOB card resets the identification of the condensed deck.

The assembled deck is selected to the NP stacker.

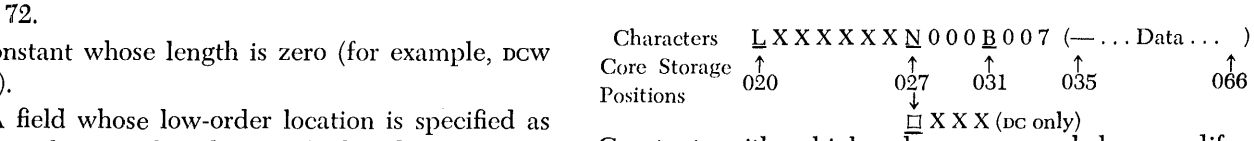
Loadable Tape

When requested, the processor writes the object program on Tape 6 in a format which makes it possible to load the program from tape using the tape-load key. The four high-order characters of the identification in the JOB card are placed in every tape record. The loader and tape area are in locations 001-080. Positions 076-079 will contain the four-character identification in each record. Position 080 contains a group mark with a word mark.

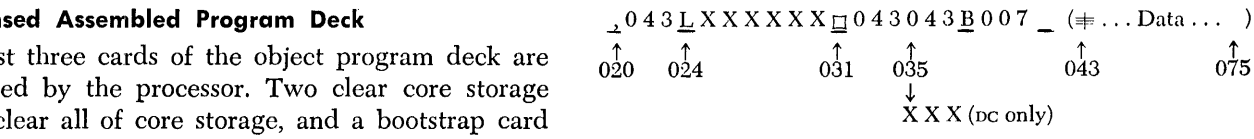
The first tape record is a clear storage routine. The second record is a bootstrap record, which, when read, appears as follows (omitting identification):



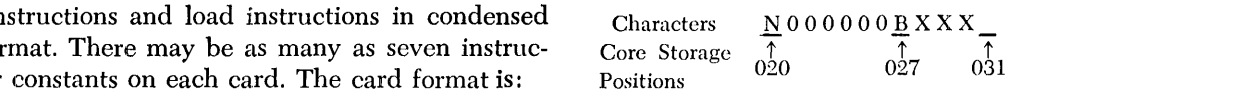
Subsequent records are read into location 020 and contain the data (one instruction or constant per record) to be loaded. The format is as follows:



Constants with a high-order group mark have a different format:

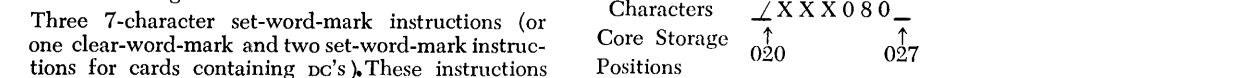


An assembled execute statement record (EX, XFR) is as follows:



Following the execute record is a new bootstrap record which begins in location 001.

An assembled END statement record is as follows:



Resequenced Source Deck

If requested, the processor will punch a new resequenced source deck (selected to stacker 4). All statements are in *Autocoder* format (free-form) and the sequencing starts with 0101 in columns 1-4 of the first source program card. If another assembly is made from the resequenced source deck, the page and line number on the listing is in sequential order, agreeing

with the cards. On the listing there are no page and line numbers assigned to statements generated by a macro instruction, but these statements have sequence numbers.

Listing Tape

If there is a fifth tape available (Tape 3), it is possible to obtain the assembly listing and condensed cards on tape for postponed output. Because this tape is not rewound before or after program assembly, it is possible to stack output from many assemblies.

Additional listings can be printed from the tape by using the IBM 1401 Tape-to-Printer Utility Program (1401-UT-026) with the following control card:

Card Columns	Contents
1-10	0133001132
11-20	blank
21-23	111
24-53	blank
54-66	0100021320011
67-79	blank
80	1

The condensed cards can be punched from this tape using the IBM Tape-to-Card Utility Program (1401-UT-028) with the following control card:

Card Columns	Contents
1-7	0081001
8-16	blank
17-30	1&120001&20001
31-41	blank
42	3
43-58	blank
59-70	0180002201
71-80	blank

When retrieving the listing or condensed cards, mount the listing tape on Tape Unit 3. For additional information about the 1401 Tape Utility programs, refer to the SRL publication, *IBM 1401/1460 Bibliography*, Form A24-1495, which lists the publication for these programs.

System Halts and/or Messages — Program Assembly

Figure 101 constitutes a listing, by phase, of the halts and/or messages that can occur during the assembly run. The information given for each halt consists of:

1. the B-address that can be displayed on the 1401 console when the halt occurs.
2. the message associated with the halt and/or the reason for the halt. (A printed message can occur without a machine halt.)
3. the procedures to be followed by the operator.

TAPE-REDUNDANCY PROCEDURES

There are standard restart procedures for tape read or write error halts which may occur during program assembly (and/or the librarian run). When a tape-redundancy halt occurs, pressing the B-address register key will display the contents of the B-address register in the storage-address display lights.

The thousands and hundreds positions of the storage-address lights represents the pass in which the redundancy occurred. The tens position will note whether it is a read or a write redundancy (6 will be displayed for a write redundancy; 9 for a read redundancy). The units position shows the tape unit on which the redundancy occurred.

For example, if 0466 is displayed from the B-address register, there is a write redundancy (6) on tape 6 (6) in Pass 4 (04). NOTE: Exceptions to this B-address convention for tape-redundancy halts occur under special conditions (see Figures 98 and 101).

Read Redundancy. If an error occurs during a tape read operation, the processor attempts to read the tape an additional ten times. If still unsuccessful, the machine halts at one of the B-addresses for read-redundancy errors given in Figure 98 or 101. After the halt occurs.

1. Turn ON Sense Switch E and press the start key to retry the read operation an additional 10 times.
2. If the same halt occurs again, turn OFF Sense Switch E. Set tape-select switch to D, and press the start key.
3. A halt will occur at the B-address given for the second tape read redundancy (Figure 98 or 101). Find the contents of the I-address register at this halt. Scan storage for incorrect characters and correct if possible. Set the tape-select switch back to N, set the I-address to the contents when the halt occurred, and press the start key to process record.
4. If the error was not corrected, restart the assembly run (or librarian run). After Pass 3 has been successfully completed, a reassembly run without alterations can be performed instead of restarting the assembly run from the beginning, unless the error occurred on Tape 4.

Write Redundancy. If an error occurs during a tape-write operation, the processor attempts to rewrite the record on successive portions of tape, skipping and blanking tape between writes. If in any one pass of the processor the redundancy procedure accumulates fifty skips (in either one, or a series of tape-write operations), the machine halts at one of the B-addresses

Pass	B-Address	Message and/or Reason	Procedure
1	0111	Second tape read redundancy halt.	Note the contents of the I-address register. Follow tape read redundancy procedure, given in the text, starting with step 3.
1	0165	Write redundancy Tape 5.	Follow the procedure given in the text for tape write redundancy.
1	0166	Write redundancy, Tape 6.	Same as the preceding item.
1	0191	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart with another system tape.
1	0195	Read redundancy, Tape 5.	Follow the tape read redundancy procedure.
1	0199	Read redundancy, Tape 1. Occurs in Selection Program when tape-load key is pressed.	Rewind system tape and press tape-load key again. If halt re-occurs, try another system tape or tape unit.
2	0201	Second tape read redundancy halt.	Note the contents of the I-address register. Follow the tape read redundancy procedure, starting with step 3.
2	0265	Write redundancy, Tape 5.	Follow tape write redundancy procedure.
2	0266	Write redundancy, Tape 6.	Same as the preceding item.
2	0288	Read redundancy, Tape 1, in overlay program segment.	Press the start key to retry the read operation once If unsuccessful after ten or so attempts, restart the assembly run with another system tape.
2	0291	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart the assembly run with another system tape.
2	0294	Read redundancy, Tape 4.	Follow the tape read redundancy procedure.
2	0295	Read redundancy, Tape 5.	Same as the preceding item.
3	0301	Indicates that the record being processed is coded by the processor with an invalid statement type	Restart assembly run.
3	0302	Second tape read redundancy halt.	Note the contents of the I-address register. Follow the tape read redundancy procedure, starting with step 3.
3	0364	Write redundancy, Tape 4.	Follow the tape write redundancy procedure.
3	0391	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart the assembly run with another system tape.
3	0395	Read redundancy, Tape 5.	Follow tape read redundancy procedure.
3	0396	Read redundancy, Tape 6.	Same as the preceding item.
3	No Halt.	PROCESSING AS FIXED FORM RECORD — Message printed when the processor encounters a statement which is not in acceptable Autocoder free-form format and there has not been a preceding Enter (ENT) SPS statement. The statement is processed and may or may not be assembled correctly.	Check the source deck later and insert correct ENT card(s).
3	No Halt.	INCORRECT PROCESSOR MACHINE SIZE SPECIFIED — Message printed if the size of the processor machine specified in column 21 of the CTL card is larger than the machine actually used. In this case the processor assumes a 4k machine.	Correct the CTL card if performing another assembly.

Figure 101. System Halts and/or Messages — Program Assembly (part 1 of 3)

Pass	B-Address	Message and/or Reason	Procedure
3	No Halt.	ACTUAL OP CODES PRESENT IN FIXED FORM IMAGES — Message printed when the processor is handling SPS fixed-form statements and encounters an actual operation code (which was punched in column 16 of the source program card). Because SPS allows the coding of actual machine op codes, the statement may be valid and the processing of this statement may be correct. However, this message warns the operator in case the punch in column 16 is the first character in an Autocoder mnemonic operation code and an Enter (ENT) Autocoder card is missing.	Check the source deck and insert correct ENT card(s) if necessary.
3	No Halt.	PASS 3 COMPLETED — Message printed at the end of the pass to indicate that Tape Unit 4 now contains the source program in free-form in condition for a reassembly run. (See Reassembly Run).	
4	0401	Second tape read redundancy halt.	Note the contents of the I-address register. Follow the tape read redundancy procedure, starting with step 3.
4	0402	Indicates that the record being processed is coded by the processor with an invalid statement type.	Restart assembly run.
4	0465	Write redundancy, Tape 5.	Follow the tape write redundancy procedure.
4	0491	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart assembly run with another system tape.
4	0494	Read redundancy, Tape 4.	Follow the tape read redundancy procedure.
5	0511	Second tape read redundancy halt.	Note the contents of the I-address register. Follow the tape read redundancy procedure, starting with step 3.
5	0566	Write redundancy, Tape 6.	Follow the tape write redundancy procedure.
5	0591	Read redundancy, Tape 1.	Follow step 1 only of the read redundancy procedure. If halt re-occurs, restart the assembly run with another system tape.
5	0595	Read redundancy, Tape 5.	Follow the tape read redundancy procedure.
6	0611	Second tape read redundancy halt.	Note contents of the I-address register. Follow the tape read redundancy procedure starting with step 3.
6	0612	Second tape read redundancy halt in overlay which prints symbol table	Same as the preceding item.
6	0665	Write redundancy, Tape 5.	Follow the tape write redundancy procedure.
6	0691	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart assembly run with another system tape.
6	0696	Read redundancy, Tape 6.	Follow the tape read redundancy procedure.
7	0711	Second tape read redundancy halt during initialization phase of Pass 7.	Note the contents of the I-address register. Follow the tape read redundancy procedure, starting with step 3.
7	0712	Second tape read redundancy halt in the main-line section of Pass 7.	Same as the preceding item.
7	0763	Write redundancy, Tape 3.	Follow the tape write redundancy procedure.
7	0766	Write redundancy, Tape 6.	Same as the preceding item.

Figure 101. System Halts and/or Messages — Program Assembly (part 2 of 3)

Pass	B-Address	Message and/or Reason	Procedure
7	0770	Indicates that a record being processed is coded by the processor with an invalid statement type.	Restart assembly run.
7	0777	Occurs when sequence numbers on input Tapes 4 and 5 do not match.	Restart assembly run.
7	0791	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs,restart assembly run with another system tape.
7	0794	Read redundancy, Tape 4.	Follow the tape read redundancy procedure.
7	0795	Read redundancy, Tape 5.	Same as the preceding item.
7	No Halt.	NO CONTROL CARD — Message printed in place of the missing statement on the listing if a control (CTL) card has not been placed in the source deck. The processor assumes that both the processing and object machine are 4k (without Modify Address) without the Read-Punch Release Special Feature. Only a assembly listing with diagnostic messages will be provided by the processor.	Other output options may be specified by the programmer at the end of assembly. (See Program Assembly.)
7	No Halt.	OBJECT CORE EXCEEDED — Message printed on the last page of the listing if the object program exceeds the core size specified in column 22 of the CTL card.	
8	0811	Second tape read redundancy halt.	Note the contents of the I-address register. Follow the tape read redundancy procedure, starting with step 3.
8	0812	Second tape read redundancy halt after attempting to read Pass 7 when additional output is requested.	Same as the preceding item.
8	0866	Write redundancy, Tape 6.	Follow the tape write redundancy procedure.
8	0880	END OF ASSEMBLY — Message printed with extra output options and instructions after completion of assembly. (See Program Assembly.)	Select extra output options, if desired, and press the start key. If no extra output is desired, press the start key.
8	0889	END OF JOB — This and other end-of-job messages are printed and the tapes (except the listing tape, if used) are rewound by the processor.	
8	0891	Read redundancy, Tape 1.	Follow step 1 only of the tape read redundancy procedure. If halt re-occurs, restart the assembly run with another system tape.
8	0894	Read redundancy, Tape 4.	Follow the tape read redundancy procedure.
8	0895	Read redundancy, Tape 5.	Same as the preceding item.

Figure 101. System Halts and/or Messages — Program Assembly (part 3 of 3)

for write-redundancy errors given in Figure 98 or 101. After the halt occurs,

1. Press the start key to retry write operation an additional 50 times.
2. If the machine halts at the same address, replace the tape in error and restart the assembly run (or librarian run). (After Pass 3 has been successfully completed, a reassembly run without alterations can be performed instead of restarting the assembly run from the beginning.)

RETRIEVING ASSEMBLY OUTPUT

If the processor is interrupted while generating output in Pass 7 or 8 (for example, while printing assembly listing, punching condensed deck, writing loadable tape, etc.) the output passes can be retrieved without having to restart the assembly run.

1. Rewind the system tape on Tape Unit 1.
2. Turn ON Sense Switches F and G.
3. Press the tape-load key. A halt will occur at B-address 0880 (end-of-assembly halt) after the extra output option messages are printed.
4. Select desired output by setting sense switches and press the start key.

Reassembly Run

If Pass 3 of the original assembly has been successfully completed, a considerable amount of time may be saved when reassembling an object program (with or without alterations) by using tape 4, which contains (after Pass 3) the source program statements in free-form with assigned sequence numbers.

Reassembly Without Alterations. If a system halt that requires restarting the assembly run occurs after Pass 3 of the assembly, the operator can save time by reassembling using Tape 4 instead of the original source program on cards (or tape).

Reassembly With Alterations. After the original assembly has been completed, it is possible to alter and reassemble the program by using ALTER cards and Tape 4. During assembly, each statement that can be altered by an ALTER entry is assigned a sequence number which is listed in the first column of the assembly listing. These sequence numbers are used in ALTER entries to add, delete, or substitute instructions in proper places in the source program.

This method permits alteration of the program without recompiling the macro statements which were processed in the initial assembly, unless the macro statements are also being altered. Note that:

1. Only those statements with sequence numbers can be altered.
2. Alteration to a CALL or INCLD macro statement does not automatically alter the referenced subroutine.
3. If a macro-statement is altered, the generated instructions will be recompiled.
4. If IOCS macro-instructions are altered, the IOCS routines will be recompiled only if Sense-Switch G is ON. (See subsequent reassembly instructions.)
5. If a macro-generated statement (IOCS or *Autocoder)* is altered, the entire macro routine is not reprocessed.
6. If LTORG or EX statements are added or deleted, closed subroutines are not rearranged due to this change.

To reassemble the object program:

1. Mount the system tape on Tape Unit 1. This tape must be rewound before starting, but the other tapes (4, 5, and 6) are rewound by the processor.
2. Ready the Tape 4 output from the original assembly on the same tape unit.
3. Mount working tapes on Tape Units 5 and 6.
4. Mount a fifth tape on Tape Unit 3 if a listing tape is to be generated.

If reassembling without alterations:

5. Turn ON Sense Switches A, B, and C.
6. Press the check-reset, start-reset, and tape-load keys. After this, follow steps 5 through 8 of the assembly run. (See *Program Assembly.*)

If reassembling with alterations:

5. Place the ALTER cards (in sequential order by statement numbers) in the card reader.
6. Turn ON Sense Switches A and B. If IOCS statements are to be regenerated also, turn ON Sense Switch G.
7. Turn ON the I/O check-stop switch.
8. Press the check-reset, start reset, and tape-load keys. After this, follow steps 8 through 10 of the assembly run. (See *Program Assembly.)*

Patching the Object Program

Correcting or revising an assembled program is accomplished through a procedure known as *patching*. Patching makes it possible for the programmer to change the condensed object-program deck without reassembling.

Two methods are used in patching:

1. The user duplicates the condensed card, substituting the correct information where needed. The corrected card is then placed in the proper location within the condensed deck before loading the object program. (The condensed deck is numbered sequentially in columns 72-75, and the card number for all data appears in the listing on the same line as the data.) This method is often used to substitute correct equal-length information (for example, addresses, d-characters) on a condensed program card.
2. The correct information is loaded into storage after the original object program has been loaded, overlaying part of the original object program. The user punches patch card(s) and places them just before the assembled END, XFR, or EX card in the object program or program segment to which the patch applies. (Check the listing for card number of the END, XFR, or EX assembled instruction.)

A patch card is punched in the following format:

Card Columns	Contents
1-39	The data, machine-language instruction(s) or constant(s), to be loaded into storage. The information must be left-justified in this field.
40-46	A load instruction which loads the above data into storage with a high-order word mark.
47-53	If the data should not have a high-order word mark, this field contains a seven-character clear-word-mark instruction. If the high-order word mark is to be left in storage, this field contains: 1. A set-word-mark instruction. If two or more instructions have been loaded, into storage, a word mark must be set for each instruction, or: 2. A NOP instruction (N000000), if additional word marks are not needed.
54-60 61-67	These fields contain set-word-mark or NOP instructions. (See preceding paragraph.)
68-71	1040. An instruction which causes the 1401 to read a card and branch to location 040, which is the address of the next load instruction or an execute instruction.
72-75	Card number
76-80	Program identification

EXAMPLE: Suppose that a programmer wishes to insert a MOVE (M 523 201) instruction in his assembled program after a 7-character add instruction (A 430 523) whose high-order location is 629. This patch involves changing the add instruction to an unconditional branch to an area which will contain instructions to add, move, and branch back to the next instruction in the program. Suppose the high order of this patch area is 800.

If the second method of patching is used, two patch cards are needed (Figure 102):

The three NOP instructions, which are loaded into positions 633, 634, and 818 of core storage, are used so that word marks are set after the unconditional branches.

After the patch cards are loaded, core-storage locations 629-635 will contain:

```
                          Next
B 8 0 0 N N 3 Instr
↑           ↑ ↑ ↑ ↑
629         633 634 635 636
```

The N3 becomes a two-character NOP instruction.

The patch area (core-storage locations 800-818) will contain

```
A 4 3 0 5 2 3 M 5 2 3 2 0 1 B 6 3 6
↑           ↑           ↑
800         807         814
```

The two patch cards are placed in the object deck (before the assembled END, XFR, or EX card) before loading the object program.

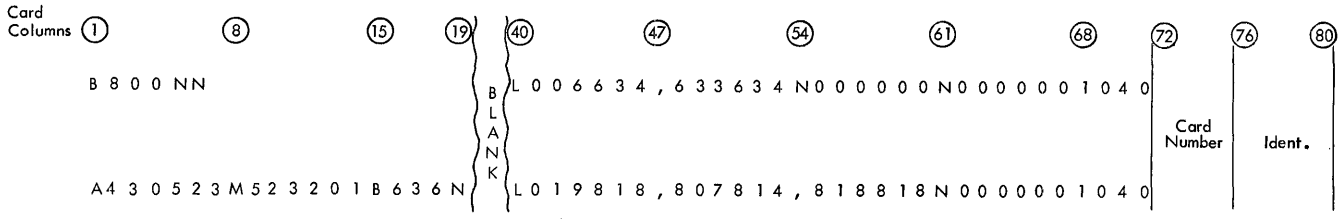


Figure 102. Patch Card Sample

Running the Object Program

To run the object program:

If the program is on cards,

1. Place the condensed deck in the card reader. (If for any reason the user does not wish to clear storage before loading the object program, remove the first two cards from the deck. These are the clear storage cards generated by the processor.)
2. Turn ON the I/O check-stop switch and sense switches as needed by the program.
3. Press the check-reset, start-reset, and load keys.

If the program is on loadable tape,

1. Mount the program tape on Tape Unit 1.
2. Turn ON the I/O check-stop switch and sense switches as needed by the program.
3. Press the check-reset, start-reset, and tape-load keys.

If the output of the assembly was on a listing tape, the condensed card deck can be punched from the tape using the utility program described in *Listing Tape*. Run the object program using the preceding instruction for a condensed deck.

Index

Actual Address 10, 13, 16
Adding Library Routines 44
Address Adjustment 9, 10, 13, 14, 15
Address Constants 13, 15
Address Types 10
Allowable Blanks in Autocoder Statements 49
Alphameric Constants 15
Alphameric Literals 11
ALTER-Alter 26
Altering the Object Program 57
Area-Defining Literals 11
Area-Definition Statements 5, 6, 16
Assembly Listing (with Error Codes) 49
Asterisk Operand 10, 19
Autocoder Listing Format 43
Autocoder Output 49
Autocoder Phases 47
Autocoder Programming System 5
Autocoder Transmittal Tape 41, 42

Blank Operand 10
Blank Constants 15
Blocking 48
Bootstrap Card 49, 51
Bootstrap Record, Loadable Tape 51
Branch Instructions 9, 20, 21

CALL Macro 32
Call Routines 32
Card Overlay Library Routine 43, 44
CHAIN Macro 36
Change Cards 43
Clear Core-Storage Cards 49, 51
Clear Core-Storage Routine, Loadable Tape 51
Coding Sheet 8
Collating Sequence 9
Comments 9
Condensed Assembled Program Deck 51
Constants 6, 10, 14
Conversion of SPS Statements 49
Copying the System Tape 45
CTL — Control Card 22

d-Character 9, 20
DA — Define Area 16
DC — Define Constant (No Word Mark) 15
DCW — Define Constant with Word Mark 14
Declarative Operations 14
DELETE — Delete 37
Deleting Library Routines 44
Disk Input/Output Instructions 39
Displaying the Library 45
DS — Define Symbol 16
DSA — Define Symbol Address 16

END — End Card 25
ENT — Enter New Coding Mode 25
EQU — Equate 19
Error Codes, Assembly Listing 50
EX — Execute 24

HEADR — Header 27

IBM-Supplied Macros 32
Identification 10
Imperative Operations 20
INCLD Macro 34
Index Locations 13
Indexing 9, 10, 13, 16, 19
Indexing (DA Entry) 17
Input/Output Devices, Auxiliary 39
INSERT — Insert 38
Instruction Statements 6, 9

JOB — Job Card 21

Label 9, 29
Librarian Run 44
Library 27
Library Entry 27
Line Number 9
Listing Tape 52
Literal 10, 30
Loadable Tape 51
Loader 6, 8, 24
LTORG — Literal Origin 24

MA Macro 43, 44
MA Macro — Modify Address 37
Machine Language 5, 6
Machine-Language Coding 39
Machine Requirements 5
Macro Instructions 31
Macro Operations 27
Macro Processing 34
Macro System 5, 27
Maximum Number of Labels 49
Messages During Updating Program 45
Mnemonic Operation Codes 5, 6, 9, 20
Model Statements 27, 28
Modify Address Feature 22, 37
Modify Address Library Routine 43, 44
Modifying Library Routines 44

Numeric Constants 14
Numeric Literals 10

Operand 5, 9
Operation 9
ORG — Origin 22
OVLAY Macro 43, 44
OVLAY Macro — Card Overlay 36

Page Number 9
Parameters 27, 28, 31
Pass 1. Selection Program and Librarian 47
Pass 2. Input, IOCS Processor, and Macro Phase 47
Pass 3. Translator Phase 48
Pass 4. Relative Addressing Phase 48
Pass 5. Label Phase 48
Pass 6. Operand Phase 48
Pass 7. Listing and Condensed Cards Phase 48
Pass 8. Loadable Tape and New Source Deck Phase 48
Patching Example 57
Patching the Object Program 57
Pre-System Run 42
PRINT — Print Library Routine 38
Processing Overlap 40
Processor Control Operations 6, 21
Processor Program 5, 6
Program Assembly 47
Program Overlay 11, 24

Programming with Autocoder 6
PUNCH — Print and Punch Library Routine 38

Read Redundancy 52
Reassembly Run 56
Reassembly with Alterations 56
Reaassembly Without Alterations 56
Resequenced Source Deck 51
Retrieving Assembly Output 56
Running the Object Program 58

SFX — Suffix 25
Source Program 5, 6
Special Features 5, 13, 22, 39
SPS Statements, Conversion of 49
Symbol Table 49
Symbolic Address 9, 10, 16, 30
Symbolic Language 5, 6
System Card Deck Format 42
System Halts — Librarian Run 46
System Halts — Pre-System Run 42
System Halts — System Run 43
System Halts and/or Messages — Program Assembly 52
System Run 43
System Tape 37
System Tape Format 43

Tape Overlay Library Routine 43, 44
Tape Read Errors 52
Tape Redundancy Procedures 52
Tape-to-Card Utility Program 52
Tape-to-Printer Utility Program 52
Tape Write Errors 52
TOVLY Macro 43, 44
TOVLY Macro — Tape Overlay 36

Updating the System Library 44

Write Redundancy 52
Writing the System Tape 42
Work Areas 5, 11, 15, 16

XFR — Transfer 25

0 4 12 10 MFW

READER'S SURVEY FORM

Autocoder (on Tape) Language
Specifications and Operating Procedures
IBM 1401 and 1460 , C24-3319-0

• Is the material:	*Yes*	*Satisfactory*	*No*
Easy to read?	☐	☐	☐
Well organized?	☐	☐	☐
Fully covered?	☐	☐	☐
Clearly explained?	☐	☐	☐
Well illustrated?	☐	☐	☐

• How did you use this publication?

As an introduction to the subject ☐
For additional knowledge of the subject ☐

• Which of the following terms best describes your job?

Customer Personnel		*IBM Personnel*	
Manager	☐	Customer Engineer	☐
Systems Analyst	☐	Instructor	☐
Operator	☐	Sales Representative	☐
Programmer	☐	Systems Engineer	☐
Trainee	☐	Trainee	☐
Other _____		Other _____	

• Check specific comment (if any) and explain in the space below:
(*Give page number*)

☐ Suggested Change (Page) ☐ Suggested Addition (Page)
☐ Error (Page) ☐ Suggested Deletion (Page)

Explanation:

Space is available on the other side of this page for additional comments.
Thank you for your cooperation.

C24-3319-0

Staple

Fold

Fold

FIRST CLASS
PERMIT NO. 170
ENDICOTT, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Products Division
Development Laboratory
Endicott, N. Y. 13764

Attention: Product Publications, Dept. 171

Fold

Fold

Cut Along Line

IBM 1401 and 1460 Printed in U. S. A. C24-3319-0

IBM

International Business Machines Corporation

Data Processing Division

112 East Post Road, White Plains, N. Y. 10601

Additional Comments: