PROGRAMS FOR IBM 1401 CARD SYSTEMS: PRELIMINARY SPECIFICATIONS

This bulletin is a minor revision of, but does not supersede, the original edition, form J28-0209. The changes are in the first paragraph of Clear Storage Routine (with respect to the card column mentioned) and in the first and third paragraphs of Print Storage Program.

This bulletin describes five IBM 1401 utility programs, five frequently-used subroutines and four program error-detection aid routines for use in program testing. The above programs were developed for use on 1401 systems using card input and card or printed output. Each program is easily adapted to perform its function in conjunction with the user's programs. Flexibility of the programs provides for using either 100 or 132 positions of printing, where applicable, as well as providing for adapting the programs for use on 1401 systems equipped with 1400, 2000 or 4000 positions of core storage.

TABLE OF CONTENTS

UTILITY PROGRAMS

In programming the IBM 1401 to solve problems or to perform functions assigned to it by the user, several routines are required repeatedly. For example, prior to loading a program it is usually desirable to clear storage to blanks. Thus, a clear storage program is usually used prior to each loading of programs or data. Also, if a printed or punched record of storage contents is desired, a program is required to provide it. These programs and some others which are general in application and used frequently in the operation of data processing systems are classified as utility programs. Several utility programs will be supplied to assist in efficient operation of the 1401 system.

The utility programs specified in this bulletin are:

Clear Storage
Card Loader
Print Storage
Punch Storage
Punch-List-Sequence Check

Each program is described separately; descriptions of the control cards necessary to specify the variables required to adapt the utility programs to the user's needs and/or equipment are included.

CLEAR STORAGE ROUTINE

Prior to loading information into the 1401 system it is frequently desirable to clear storage. This routine, contained in two cards, clears all of storage to blanks and leaves a word mark set in location 001. Column 27 of card number two contains one of three alphabetic letters, depending upon the number of storage positions with which the 1401 system is equipped:

T for 1400 positions
Z for 2000 positions
I for 4000 positions

CARD LOADER PROGRAM

This program is designed to load into the 1401 system those instructions (in machine language) and constants which are punched into cards. The instructions and constants supplied by the user should be punched in one-per-card format as specified below under Instruction or Constant Card Format.

Instruction or Constant Card Format

Column 1 is punched with a factor code. The codes are defined as follows:

I - Factor is an instruction.

K - Factor is a constant with a word mark to be set in the high-order position.

N - Factor is a constant with no word mark to be set.

* - Denotes a comment card (acceptable in this routine only to provide compatibility with the Punch-List-Sequence Check program).

/ - Denotes an END card (optional; may be used to begin the execution of the object program immediately after loading).

Columns 2-4 are punched with the number of characters in the instruction or constant, with the number right-justified and preceded by zeros when necessary; e. g. , an eight-character instruction is punched 008.

Columns 5-7 are punched with the specified storage location of the factor, as follows:

For an instruction, the storage location of the high-order position is specified.
For a constant, the storage location of the low-order position is specified.

Columns 8-onward, as needed, are punched with the instruction, constant, or comments. Remarks are acceptable, not only on comments cards, but also when punched to the right of instructions or constants. Instructions are coded in machine language.

A comments card requires only an asterisk in column 1 and the remarks in columns 8-onward. The comments card is not processed by this routine, but is allowed to be included in the deck for convenience when processing the deck using the Punch-List-Sequence Check program.

Self-loading subroutines in machine language may also be loaded along with related programs using this loader program. The procedure is to place the subroutine card deck, preceded by its location designator card (see page 24), behind the related program deck.

The Card Loader Program requires approximately 110 storage locations, and it loads cards at the rate of 800 per minute.

This program is self-loading; i. e. , it contains loading instructions as well as program instructions and constants. The loading instructions cause the program instructions and constants to be stored in the locations assigned to them. The Card Loader Program may be located in storage either in locations 0090-0199 or locations 1290-1399, depending upon which of two program decks is used.

PRINT STORAGE PROGRAM

This program is designed to print out selective portions of storage. The portion of storage to be printed out is specified in a control card (the last card of the program deck) which contains the locations of the lower and the upper limit of core storage and the model number of the IBM 1403 printer used with the system. The lower limit address is to be punched in card columns 1-3, the upper limit address in columns 4-6, and the

4

1403 model number in column 7. Model 1, represented by a 1-punch in column 7, specifies a 1403 equipped with 100 print positions, while model 2, represented by a 2-punch in column 7, specifies 132 print positions. The Print Storage Program is self-loading and relocatable. The printed output will contain an indication of the positions of any word marks in the selected portion of storage.

Relocatable programs are those programs whose instructions and constants may be assigned any locations in storage. Instructions and constants of relocatable programs are initially assigned storage locations beginning with location 000. Relocatable programs may be loaded into any locations in core storage by the user who supplies the starting location. The starting location, specified in a location designator card as explained on page 24, is the location in storage where the first instruction of the relocatable program is to be stored. The loading of a relocatable program is accomplished by its loading program which modifies the appropriate storage locations, incrementing each one by the starting address.

This program requires approximately 145 storage locations.

Control Card Format

1	2	3	4	5	6	7	
Selected Block of Storage:							
	Lower Limit Address			Upper Limit Address		1403 Model Number	

PUNCH STORAGE PROGRAM

This program may be used to punch out selective portions of storage. The portion of storage to be punched out is specified in a control card (the last card of the program deck) which contains the locations of the lower limit and the upper limit of core storage. The lower limit is to be punched in card columns 1-3 and the upper limit in columns 4-6. The card output resulting from this program is a self-loading deck. If desired, the user may specify in the control card, columns 7-9, the location of the instruction in the object program to be executed after reloading the selected portion of storage punched out. When this option is used, the Punch Storage Program produces an END card which is selected into pocket 4 of the 1402 Card Read Punch. This card, when reloaded as the last card of the output produced by the Punch Storage Program, clears the read area and causes the program to branch to the specified starting location after the information punched out is reloaded into storage. The Punch Storage Program is self-loading and relocatable.

This program requires about 160 storage locations. It punches out the selected portion of storage at the rate of 250 cards per minute.

Control Card Format

1	2	3	4	5	6	7	8	9
Selected Block of Storage:								
	Lower Limit Address			Upper Limit Address			Starting Location	

PUNCH-LIST-SEQUENCE CHECK

This program is designed to punch, list, and sequence check, or any combination of these operations, information from an input program deck whose format conforms to that specified for the Card Loader Program, page 3 . The options selected by the user, as well as the designation of the storage capacity of his 1401 system, are specified either in a control card or by setting the sense switches as shown in Figure 1 on a following page. For example, if the user chooses to punch a self-loading program deck from an input deck which is of the format specified for the Card Loader Program, and if his 1401 system has a storage capacity of 2000, he may specify his output option and storage capacity in either one of two ways:

1. A control card with a 1-punch in column 1 and a 2-punch in column 4.
2. Setting the sense switches B and F DOWN; C, D, E and G UP.

If a control card is used, it is placed with the input cards as the first card of the deck. Program operation in determining the output options and storage capacity of the system first checks for the presence of a control card, taking the information from it if specified therein. If a control card is not present, the program checks the position of the sense switches, taking the information from their settings. If no output option is specified (neither punch, list, nor sequence check), the following message is printed: OUTPUT NOT SPECIFIED. If no 1401 system storage capacity is specified, STORAGE SIZE NOT SPECIFIED is printed out.

If the option to punch is chosen, the output cards will be a self-loading program deck, the first two cards of which contain a clear storage routine. Comments cards in the input deck, identified by an asterisk punch in column 1, are not punched. Optional use of one header card in the input deck is provided, as follows: One header card per program, identified in the input deck by an asterisk in column 1 and an H in column 2, and containing heading information in columns 8-80, will be punched. When an input header card is included, it must precede all other program cards of the input deck, following only the control card (if one is used) of the Punch-List-Sequence Check program. The output header card contains 1001 in columns 1-4; the heading information is contained in columns 8-80.

For the listing option, the program prints from the input program deck the factor code from column 1, the number of characters in the instruction or constant from columns 2-4, the storage location of the instruction or constant from columns 5-7, the instruction or the constant and comments, if any, from columns 8-80. Comments cards, if present, are also listed.

6

For the option of sequence checking, the program checks not only that the addresses of the factors (i. e., instructions and constants) are in sequence, but also that no addresses are skipped. If either condition is not met, the words SEQUENCE BREAK are printed on a separate line regardless of whether or not the listing option is chosen, to call attention to the change in normal sequence.

Sequence checking of storage locations of factors is done in the following manner: A counter, set to an appropriate starting address for the first factor, is incremented by the number of characters in the count field of the first factor. This result is the computed storage location of the first factor. A test is made to determine whether the computed and the assigned (i. e., punched in the card) storage locations are the same. If they are not the same, the comment SEQUENCE BREAK is printed on a separate line. The counter computing the storage addresses is reset to the assigned location, however, and the computed location of the next factor is determined by incrementing the counter by the number of characters in that factor. A test is made to determine whether the computed and assigned storage locations agree for this factor. If the locations agree, the program proceeds to operate upon the next factor. It should be noted that when the locations do not agree, the program assumes that the assigned storage location punched in the instruction or constant card is correct; the break in sequence is indicated by the printed line mentioned above.

If the user's option specifies listing and sequence checking, the next printed line after the comment SEQUENCE BREAK is the normal printed line for that instruction or constant. If the option is sequence checking but not listing, then on the line below the comment SEQUENCE BREAK appears an image of the card without the normal spacing between characters and alignment in format.

As an example of the procedure for using this program, the following conditions are assumed:

> A user has a 1401 program card deck punched in the format specified for the Card Loader Program; the first card of his input deck is a header card, containing an asterisk punch in column 1, an H punch in column 2, and the name of the program in columns 8-onward. He chooses to produce a self-loading output deck using the Punch-List-Sequence Check program, list the input cards, and sequence check the storage locations of the input deck program instructions and constants. His 1401 system is equipped with 4000 positions of storage. The user chooses to specify his output options and 1401 system capacity by preparing a control card, in which he punches a 1 in column 1, a 1 in column 2, a 1 in column 3, and a 3 in column 4. Had he chosen to do so, he could have used the sense switches of the IBM 1401 Processing Unit rather than the control card to specify his output options and storage capacity, using the following settings: switches B, C, D, and G DOWN; switches E and F UP.

> The user readies his 1401 system, places the program deck for the Punch-List-Sequence Check program in the 1401 read feed hopper and loads that program into the system. The input program deck, assembled as described in the next paragraph, is placed in the read feed hopper and program execution is begun by depression of the 1402 read feed Start key.

In this example the user puts together an input deck in the following order: Control card, followed by the one header card, followed by the cards of his object program. The input deck is placed in the 1402 read feed hopper, blank cards in the punch feed hopper and the Punch switch on the 1402 Card Read Punch is turned ON. Depression of the Start key of the 1402 read feed starts program operation. The first two cards punched out contain a clear storage routine. The next card of output is the header card. The output deck is a self-loading program deck. Comments cards of the input deck are not punched out. The listing includes the comments cards of the input deck. If storage locations were skipped or if they were not assigned in sequence, indications are printed out as previously explained. Upon encountering an end-of-file condition the program halts. After placing another input program deck, assembled as described above, in the read feed hopper, operation upon it may be started by depression of the 1401 read feed Start key, transferring program control to the location of the first instruction of the Punch-List-Sequence Check program.

| | Card Columns | | | |
	1	2	3	4
Control Card Format	1-Punch Specifies to Punch a Self-Loading Program Deck	1-Punch Specifies to List	1-Punch Specifies to Sequence Check	Specifies 1401 System Storage Capacity: 1-Punch - 1400; 2-Punch - 2000; 3-Punch - 4000.
Corresponding Sense Switches	B Down	C Down	D Down	E; F; or G — Down

Figure 1
Methods of Specifying Output Options and 1401 System Storage Capacity

SUBROUTINES

Arithmetic computation routines such as multiply and divide are frequently necessary as subordinate parts, or subroutines, of data processing programs. Several subroutines will be supplied which the user may incorporate into his main programs.

The subroutines specified in this bulletin are:

Multiply I   (for storage space economy)
Multiply II   (for speed economy)
Divide
Dozens-to-Units Conversion
Units-to-Dozens Conversion

Also presented are the instructions to transfer program control from a main program to a subroutine.

Depending upon their structure and use, subroutines are usually classified as either open or closed. An open subroutine is a set of instructions, integrated into the logical flow of a main program, for performing a given function. If the function performed by the subroutine is required more than once during the execution of the main program, the instructions of the open subroutine appear at each place needed in the main program. Since the open subroutine is a part of the logical flow of the main program, data needed by the subroutine is provided by the main program in the same manner as data furnished to any other part of the main program.

A closed subroutine is a set of instructions, not integrated into the logical flow of a main program, for performing a given function. The set of instructions of a closed subroutine appears only once in a main program, even though the function performed by the subroutine is used at many places in the program. A closed subroutine performs initialization of work areas and the necessary movement of data associated with its function. At each point in the main program where the closed subroutine is to be used, a linkage is inserted to provide (1) an entry to the subroutine, (2) a method for determining the address of the instruction in the main program to be executed after leaving the subroutine and (3) a means for obtaining the data required by the subroutine.

Each subroutine described in this bulletin is a closed routine. As closed subroutines, they each contain instructions which initialize work areas and move information about as required in conjunction with performing the arithmetic for the operation. In some applications it may be advantageous to use open subroutines rather than closed ones, taking advantage of known contents of certain storage locations to eliminate unnecessary operations. Thus the user, by modifying the closed subroutines for some applications, will save a small amount of the execution time and a considerable amount of the storage space required.

If no sign determination of the result is required, the multiply and the divide subroutines may be modified to save a small amount of time and a significant amount of storage space.

Each subroutine is described and the instructions and parameter definitions to be inserted into the main program are specified.

Program decks for each of the five subroutines presented in this bulletin will be supplied in either of two formats: (1) symbolic, [1] or (2) self-loading, relocatable, in machine language.

LINKAGE FROM MAIN PROGRAM TO SUBROUTINE

In order to transfer control from the main program to a subroutine, the user must include in the main program two instructions and the storage address of the low-order positions of the data fields required.

The two instructions are as follows:

LINE		COUNT		LABEL		OPERATION		(A) OPERAND				(B) OPERAND			
								ADDRESS	±	CHAR. ADJ.	RES.	ADDRESS	±	CHAR. ADJ.	RES.
3	5	6	7	8	13	14	16	17	23		27	28	34		38
0 1 0		7				M V		*	–	3		(SR Label)	–	1	
0 2 0		4				B		(SR Label)							
0 3 0															

The first instruction places the location of its (A) operand in a three-character work area, defined in the subroutine, whose location is just ahead of the first instruction of the subroutine. This instruction provides the subroutine with sufficient information so that it can locate the parameter definitions and establish the location in the main program to which the subroutine transfers control after completing its operations.

The second instruction is an unconditional branch instruction, transferring program control from the main program to the first instruction of the subroutine.

The parameters required by the subroutine must follow the above two instructions. Their formats are as follows:

LINE		COUNT		LABEL		OPERATION		(A) OPERAND				(B) OPERAND			
								ADDRESS	±	CHAR. ADJ.	RES.	ADDRESS	±	CHAR. ADJ.	RES.
3	5	6	7	8	13	14	16	17	23		27	28	34		38
0 1 0		3				D S A		*				P A R A M 1			
0 2 0		3				D S A		*				P A R A M 2			
0 3 0															

Control will be returned to the main program at the instruction following the last parameter definition.

The following example illustrates the specifications given above.

---

[1] See the bulletin "IBM 1401 Symbolic Programming System: Preliminary Specifications," form J28-0200.

10

LINE		COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND			
3	5	6 7	8 ... 13	14 ... 16	ADDRESS 17	± 23	CHAR. ADJ.	RES. 27	ADDRESS 28	± 34	CHAR. ADJ.	RES. 38
0 1 0		•		•	•				•			
0 2 0		•		•	•				•			
0 3 0		7		M V	N A M E				0 1 1 2			
0 4 0		7		M V	*	–	3		M U L T 1	–	1	
0 5 0		4		B	M U L T 1							
0 6 0		3		D S A	*				R A T E			
0 7 0		3		D S A	*				H O U R S			
0 8 0		7		M V	P R O D				0 1 6 0			
0 9 0		•		•	•				•			
1 0 0		•		•	•				•			
1 1 0												

The instruction shown on line 030 represents the last instruction in the main program to be executed prior to encountering the multiply subroutine linkage. In the subroutine it is desired to multiply rate times hours to yield the product PROD. The two instructions on lines 040 and 050 are required to provide entry to the multiply subroutine. Line 060 specifies the multiplier parameter (RATE), and line 070 specifies the multiplicand parameter (HOURS). The instruction on line 080 is the first instruction in the main program to be executed after leaving the subroutine.

MULTIPLY I SUBROUTINE

This multiply subroutine is provided for use on 1401 systems not equipped with the multiply-divide optional feature. Multiply I is designed to multiply two numbers, each having a maximum of ten digits, using a minimum of storage space. The result is located in the symbolic location labeled PROD as a 20-digit number with the proper sign. The sign of the result is either one or two zone bits (B and A, or B) of the binary coded decimal codes B, A, 8, 4, 2, 1. If the result of multiplication is positive, the units position of location PROD contains (in addition to its binary coded decimal digit) both zone code bits B and A; if the result is negative, it contains the zone code bit B. The user may relocate the product, if desired, by an appropriate instruction; or he may use the product area for further operations on the result of the multiplication.

This subroutine requires approximately 230 storage locations.

In order to incorporate Multiply I as a subroutine into a main program, the user appropriately inserts the following instructions and parameters in his main program:

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND			
3 5	6 7	8 13	14 16	ADDRESS 17	± 23	CHAR. ADJ.	RES. 27	ADDRESS 28	± 34	CHAR. ADJ.	RES. 38
0 1 0	7		M V	*	–	3		M U L T 1	–	1	
0 2 0	4		B	M U L T 1							
0 3 0	3		D S A	*				(Multiplier label)			
0 4 0	3		D S A	*				(Multiplicand label)			
0 5 0											

The timing estimates for Multiply I are as follows:

> For the maximum size of multiplier and multiplicand, ten digits each, an average time of approximately 45 milliseconds is required.

> For multipliers and multiplicands one-half the maximum size (five digits each) an average time of approximately 28 milliseconds is required.

In this subroutine the multiply time depends not only upon the number of digits in the multiplier and multiplicand, but also upon the value (0-9) of each digit of the multiplier. When comparing the amounts of time required to multiply a given multiplicand by various one-digit multipliers, the least amount of time is required to multiply the multiplicand by 0, and the greatest amount of time is required to multiply the multiplicand by 9. The average time, therefore, corresponds to a multiplier halfway between 4 and 5. For this reason the average multiply times in the two examples given above are based upon an average value of 4.5 for each multiplier digit.

As was previously mentioned, a small amount of time and a considerable amount of storage can be saved over that otherwise required if Multiply I is made an open subroutine or if it is altered to require no sign determination for the product. Making it an open subroutine will save approximately two milliseconds of time and 75 storage locations. This is a saving of approximately 4% of the time and 32% of the storage space required.

Modifying this subroutine to make no sign determination for the product will reduce the time and storage requirements, compared to the unmodified version, by approximately one millisecond and 60 locations. This is a saving of about 2% and 25%, respectively, of the time and storage space required.

MULTIPLY II SUBROUTINE

Like Multiply I, this multiply subroutine is designed for use on 1401 systems not equipped with the multiply-divide optional feature. Multiply II is available to multiply two numbers, each having a maximum of nine digits, in a minimum amount of time for large numbers. The result is an 18-digit number located in symbolic location PROD with the proper sign. The BCD (i.e., binary coded decimal) zone bit structure for the sign of the product is the same as that explained in the Multiply I Subroutine. In the Multiply II Subroutine the salient feature is saving time, whereas in the Multiply I Subroutine it is conserving storage space.

To incorporate Multiply II as a subroutine of a main program, the same procedure and similar instructions as given for Multiply I Subroutine are used. The symbolic name of the subroutine is MULT2 in place of MULT1.

This subroutine requires approximately 340 storage locations.

The timing estimates for Multiply II are as follows:

> For the maximum size of multiplier and multiplicand, nine digits each, a multiply time of approximately 26 milliseconds is required. Unlike Multiply I, multiply time for this subroutine is not affected by the value of multiplier digits because a table of factors of the multiplicand is used.

> For a multiplier and a multiplicand approximately one-half the maximum size (i. e., five digits each) a multiply time of approximately 21 milliseconds is required.

As previously mentioned, a small amount of time and a considerable amount of storage space can be saved over that otherwise required by modifying Multiply II for use as an open subroutine or by modifying it to require no sign determination for the product. The approximate amount of time and storage space saved when Multiply II is made an open subroutine is the same as that for Multiply I, namely two milliseconds (8%) and 75 locations (22%).

Modifying Multiply II to make no sign determination for the product will reduce the time and space requirements, compared to the unmodified version, the same as for Multiply I, or approximately one millisecond (4%) and 60 locations (18%).

DIVIDE SUBROUTINE

This subroutine is provided to accomplish division on 1401 systems not equipped with the multiply-divide optional feature. The length of divisor, dividend and quotient each may range from one to 20 digits, according to the conditions given on page 20. The results of division are stored by the subroutine in the symbolic field labeled QUOT (see Figure 3). The sign of the quotient and the sign of the remainder are represented in their respective locations as zone bits of the BCD code, as explained in the Multiply I Subroutine. The remainder sign is the same as the sign of the dividend and is included in the units position of the remainder (see Figure 3). The quotient sign is determined by the subroutine from a sign analysis of the dividend and the divisor and is included in the units position of the quotient.

The Divide Subroutine accomplishes division by means of repetitive subtraction. For each quotient digit required, beginning with the high-order quotient digit position, the divisor is subtracted from the dividend; subtraction cycles continue until the result of subtraction turns negative, while the number of subtraction cycles is counted. The number of subtraction cycles required before the result turns negative is the quotient digit. Figure 2 illustrates the method of division used by the subroutine.

Shift-Factor, n

One requirement of this subroutine is that the divisor be so aligned with the high-order positions of the dividend that no more than nine repetitive subtraction cycles are required before the result turns negative. In the example shown in Figure 2, considering the units position of the divisor with respect to the units position of the dividend, the divisor is offset to the left of the dividend one position. The number of positions of left-offset of the units position of the divisor with respect to the units position of the dividend is the shift-factor, n. The Divide Subroutine uses the shift-factor, n, for two purposes:

1. To position the divisor with respect to the dividend prior to developing the first (high-order) quotient digit.

2. To control the number of quotient digits developed by terminating division after developing (n + 1) quotient digits.

As will be noted from each example of division given, the value of n is one less than the number of quotient digits required in a problem. The value of n, which is a positive, 2-digit number ranging from 00-19 inclusive, must be specified by the user once for each subroutine application, as shown on page 21.

The method of division used by this subroutine, illustrated in Figure 2, is as follows:

Division Problem: (+43) ÷ (+3) = quotient (2 digits) and remainder

In this example the quotient will be computed to two digits. The divisor (+3), the dividend (+43) and the shift-factor n (01) are specified to the subroutine as explained on page 21. The following program action is accomplished automatically by the subroutine.

The subroutine clears the field labeled QUOT to zeros. The dividend (43) is loaded into the symbolic field labeled QUOT. In order to provide one position of storage in which each quotient digit will be developed (one at a time), the dividend is moved one position to the left. The relative alignment of dividend and divisor, specified by the shift-factor n, is maintained by positioning the divisor, prior to the first subtraction, so that its units position will be subtracted from the location QUOT - (n + 1) of the field labeled QUOT. Development of the high-order quotient digit begins.

1. The divisor (3) is subtracted from the dividend (04). The sign of the result ($\overset{+}{01}$) is tested; since it is positive, 1 is added to the contents of storage location QUOT (having been initially set to 0, it now contains 1) and subtraction will continue.

2. The divisor (3) is subtracted from the previous result of subtraction ($\overset{+}{01}$). The sign of the result ($\overset{-}{02}$) is tested; since it is negative, 1 is not added to the contents of storage location QUOT (it contains 1) and the divisor is added back to the result ($\overset{-}{02}$), restoring the previous positive result ($\overset{+}{01}$) which is less than the divisor.

Development of high-
order quotient digit:

1 = net number of
subtraction cycles

Development of the next
quotient digit (in this case,
the last):

4 = net number of
subtraction cycles

Subroutine has finished
division, having developed
(n + 1) quotient digits.

```
                                    0  4  3  0
                                    -  3
                                    _____
                                       +
                                    0  1  3  1
                                    -  3
                                    _____
                                       -
                                    0  2  3  1
                                    +  3
                                    _____
                                       +
                                    0  1  3  1

                                    1  3  1.0
                                    -  3
                                    _____
                                       +
                                    1  0  1  1
                                    -  3
                                    _____
                                       +
                                    0  7  1  2
                                    -  3
                                    _____
                                       +
                                    0  4  1  3
                                    -  3
                                    _____
                                       +
                                    0  1  1  4
                                    -  3
                                    _____
                                       -
                                    0  2  1  4
                                    +  3
                                    _____
                                       +     +
                                    0  1  1  4
```
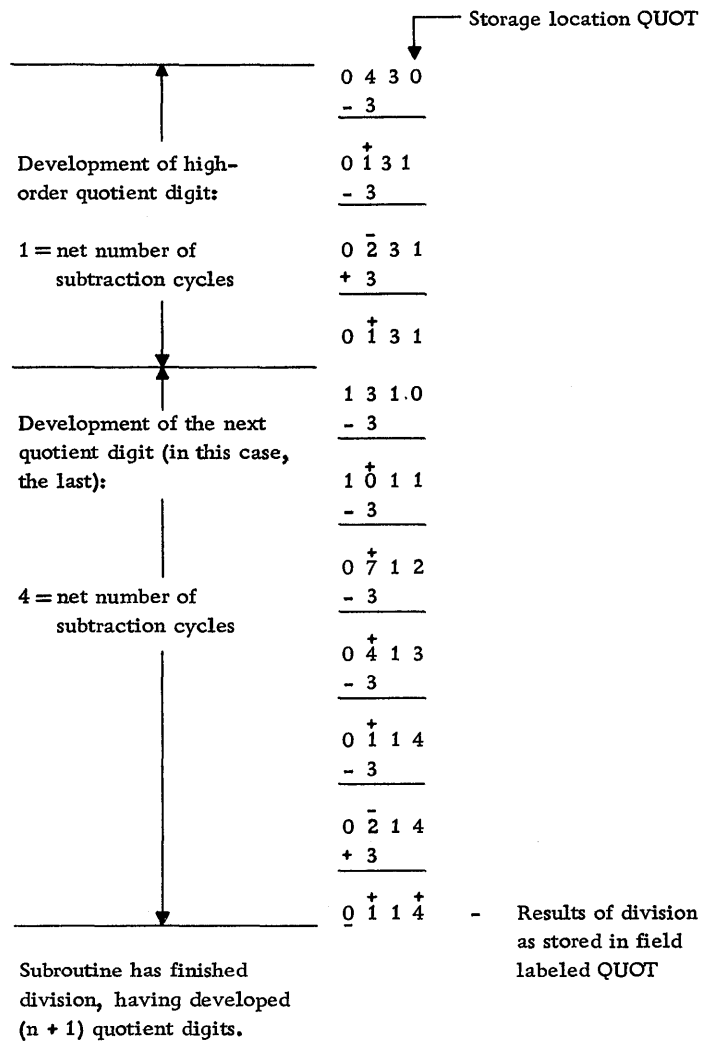
– Results of division
as stored in field
labeled QUOT

Figure 2

Division Problem: (+ 43) ÷ (+3) = Quotient (2 Digits) and Remainder

Information Supplied to the Divide Subroutine:

Divisor:	$\overset{+}{3}$
Dividend:	$\overset{+}{43}$
Shift-Factor, n:	$0\overset{+}{1}$

Development of the high-order quotient digit (1) is complete. To develop the
second quotient digit the subroutine moves the contents of the field labeled QUOT,
which contains 0131, one position to the left and places a zero in the low-order
position (storage location QUOT).

3. The divisor is subtracted from the (shifted) previous result, i. e. , the previous
    positive remainder. The sign of the result $(1\overset{+}{0})$ is tested; since it is positive,
    1 is added to the contents of storage location QUOT (making the contents 1)
    and subtraction will continue.

4. The divisor is subtracted from the previous result $(\overset{+}{10})$. The sign of the
    result $(0\overset{+}{7})$ is tested; since it is positive, 1 is added to the contents of storage
    location QUOT (making the contents 2) and subtraction will continue.

5. The divisor is subtracted from the previous result $(0\overset{+}{7})$. The sign of the result
    $(0\overset{+}{4})$ is tested; since it is positive, 1 is added to the contents of storage location
    QUOT (making the contents 3) and subtraction will continue.

6. The divisor is subtracted from the previous result $(0\overset{+}{4})$. The sign of the result
    $(0\overset{+}{1})$ is tested; since it is positive, 1 is added to the contents of storage location
    QUOT (making the contents 4) and subtraction will continue.

7. The divisor is subtracted from the previous result $(0\overset{+}{1})$. The sign of the result
    $(0\overset{-}{2})$ is tested; since it is negative, 1 is not added to the contents of storage
    location QUOT (it contains 4) and the divisor is added back to the result $(0\overset{-}{2})$,
    restoring the previous positive result $(0\overset{+}{1})$ which is less than the divisor.

Development of the second quotient digit (4) is complete. Furthermore, division
is complete because of a signal to the subroutine that (n + 1) quotient digits have
been developed. The results of the problem in this example appear in storage in
the symbolic field labeled QUOT as $\underline{0}$ $\overset{+}{1}$ 1 $\overset{+}{4}$.

Location of Results

The quotient consists of (n + 1) digits. The low-order position of the quotient and the
quotient sign are located in symbolic location QUOT. The remainder consists of the
same number of digits as the divisor field. The low-order position of the remainder and
the remainder sign are located in storage at symbolic location $\left[\text{QUOT} - (n + 1)\right]$. Defining
the length of the divisor field as LDVR, the zero, which always precedes the remainder,
and the associated word mark both are located in symbolic location $\left[\text{QUOT} - (n + 1 + \text{LDVR})\right]$.
It should be noted that the quotient and the remainder are not separated by a word mark.
Figure 3 illustrates the contents of the symbolic field QUOT.

Length of Dividend

In the example shown in Figure 2, the division problem was (+43) ÷ (+3), to yield a two-digit
quotient and a remainder. However, if the number of quotient digits desired is increased,
the dividend length must be increased by adding a sufficient number of zeros to the low-
order end. The two following examples, shown in Figures 4 and 5, illustrate the effect
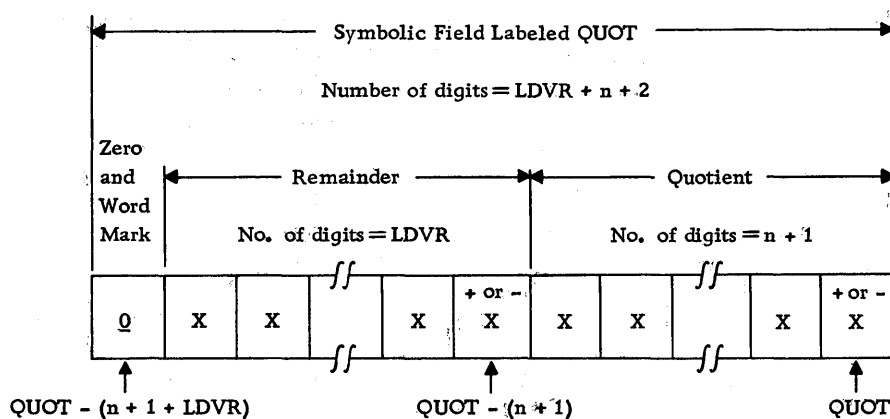
Figure 3

Contents of Symbolic Storage Field QUOT

of the dividend length as well as the shift-factor. The first of these (shown in Figure 4), while using the same values of divisor and dividend shown in Figure 2, requires that the quotient be computed more precisely than in Figure 2. The second example to follow, in addition to illustrating the method of division, points out the use of the range of data in applying the Divide Subroutine.

Referring to Figure 4, the division problem is: (+43) ÷ (+3) = quotient (4 digits) and remainder.

The length of the dividend (43) must be increased before use by the subroutine by adding two zeros to the low-order end if the quotient is to be computed to four digits. The altered dividend appears in storage as 4300. The problem to be solved by the subroutine is represented as:

$$(+43.\,00) \div (+3) = XX.\,XX \text{ and remainder}$$

The value of n (the shift-factor) specified to the subroutine is equal to one less than the number of quotient digits desired. In this example, therefore, n is equal to 3; it is specified to the subroutine as 03. It should be noted that the value of n causes the divisor to be positioned with respect to the altered dividend as supplied to the subroutine (e.g., 4300).

Referring now to Figure 5, gross weekly pay divided by total hours worked per week will yield average pay for this example. Average pay will be computed to mils. Two practical restrictions will be assumed and used advantageously:

1. Average pay does not exceed 9.999 dollars.
2. Total number of hours worked per week does not exceed 99.9.

The form of the computed quotient, average hourly pay, is X. XXX. Gross pay and total hours worked per week are in the form XXX. XX dollars and XX. X hours, respectively. The divisor, total hours per week, will be used in the form given,
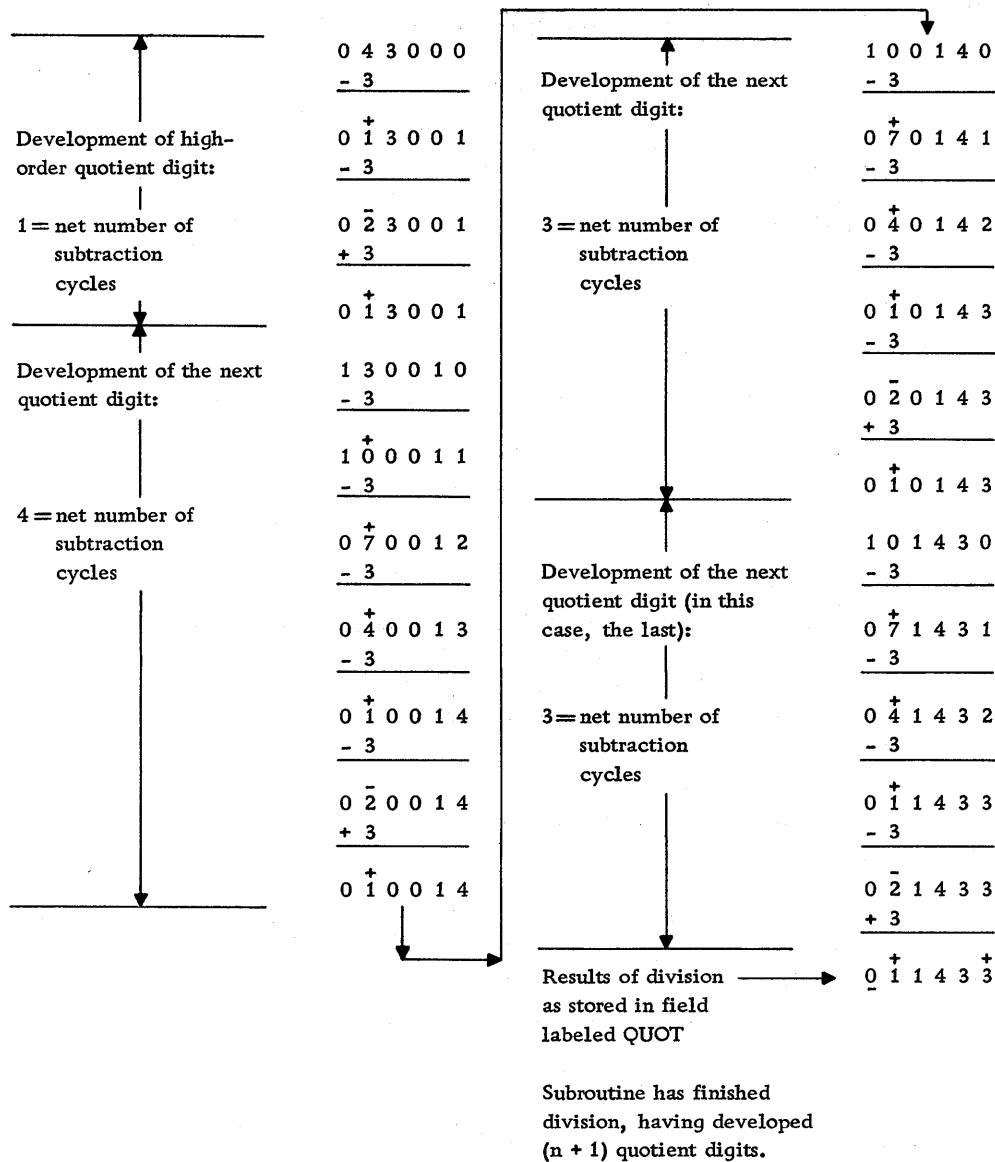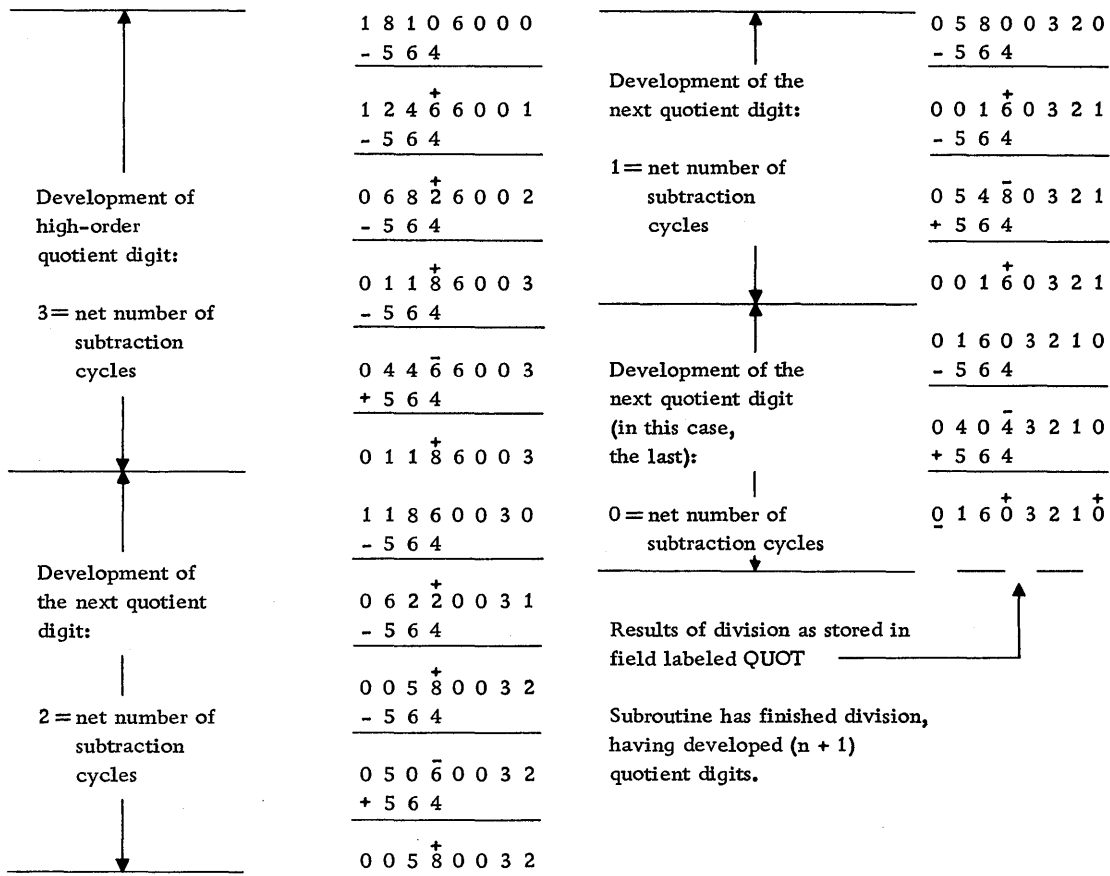
Development of high-order quotient digit:

1 = net number of subtraction cycles

Development of the next quotient digit:

4 = net number of subtraction cycles

```
0 4 3 0 0 0
-   3
   +
0 1 3 0 0 1
-   3
   -
0 2 3 0 0 1
+   3
   +
0 1 3 0 0 1

1 3 0 0 1 0
-   3
   +
1 0 0 0 1 1
-   3
   +
0 7 0 0 1 2
-   3
   +
0 4 0 0 1 3
-   3
   +
0 1 0 0 1 4
-   3
   -
0 2 0 0 1 4
+   3
   +
0 1 0 0 1 4
```

Development of the next quotient digit:

3 = net number of subtraction cycles

Development of the next quotient digit (in this case, the last):

3 = net number of subtraction cycles

Results of division ⟶
as stored in field
labeled QUOT

```
1 0 0 1 4 0
-   3
   +
0 7 0 1 4 1
-   3
   +
0 4 0 1 4 2
-   3
   +
0 1 0 1 4 3
-   3
   -
0 2 0 1 4 3
+   3
   +
0 1 0 1 4 3

1 0 1 4 3 0
-   3
   +
0 7 1 4 3 1
-   3
   +
0 4 1 4 3 2
-   3
   +
0 1 1 4 3 3
-   3
   -
0 2 1 4 3 3
+   3
   +       +
0 1 1 4 3 3
```

Subroutine has finished
division, having developed
(n + 1) quotient digits.

Figure 4

Division Problem: (+43) ÷ (+3) = Quotient (4 Digits) and Remainder

Information Supplied to the Divide Subroutine:

Divisor:	$\overset{+}{3}$
Dividend:	$43\overset{+}{0}0$
Shift-Factor, n:	$0\overset{+}{3}$

Development of
high-order
quotient digit:

3 = net number of
subtraction
cycles

Development of
the next quotient
digit:

2 = net number of
subtraction
cycles

```
1 8 1 0 6 0 0 0
- 5 6 4
            +
1 2 4 6 6 0 0 1
- 5 6 4
          +
0 6 8 2 6 0 0 2
- 5 6 4
          +
0 1 1 8 6 0 0 3
- 5 6 4
        -
0 4 4 6 6 0 0 3
+ 5 6 4
          +
0 1 1 8 6 0 0 3

1 1 8 6 0 0 3 0
- 5 6 4
        +
0 6 2 2 0 0 3 1
- 5 6 4
        +
0 0 5 8 0 0 3 2
- 5 6 4
          -
0 5 0 6 0 0 3 2
+ 5 6 4
        +
0 0 5 8 0 0 3 2
```

Development of the
next quotient digit:

1 = net number of
subtraction
cycles

Development of the
next quotient digit
(in this case,
the last):

0 = net number of
subtraction cycles

Results of division as stored in
field labeled QUOT

Subroutine has finished division,
having developed (n + 1)
quotient digits.

```
0 5 8 0 0 3 2 0
- 5 6 4
          +
0 0 1 6 0 3 2 1
- 5 6 4
        -
0 5 4 8 0 3 2 1
+ 5 6 4
          +
0 0 1 6 0 3 2 1

0 1 6 0 3 2 1 0
- 5 6 4
        -
0 4 0 4 3 2 1 0
+ 5 6 4
          +         +
0 1 6 0 3 2 1 0
```

Figure 5

Division Problem: (+18106) ÷ (+564) = Quotient (4 Digits) and Remainder

Information Supplied to the Divide Subroutine:

Divisor:	$\overset{+}{564}$
Dividend:	$1810\overset{+}{600}$
Shift-Factor, n:	$\overset{+}{03}$

XX. X hours. The form of the dividend, however, must be altered prior to its use in the Divide Subroutine to yield the four quotient digits which are required in the statement of the problem.

To use a specific example:

Gross weekly pay $181.06
Number of hours worked in the week 56.4

To determine the number of zeros required to be added to the low-order positions of the dividend, the usual rule is used: the number of decimal places in the dividend equals the sum of the number of decimal places in the quotient and the divisor. In this example, the sum of the number of decimal places in the divisor (XX. X) and the quotient (X. XXX) is four; therefore the form of the dividend (XXX. XX) must be altered by the addition of two zeros to the low-order end so that it has four decimal places. The altered form of the dividend will be XXX. XX00, and for the specific example it will be supplied to the subroutine as 1810600. The problem to be solved by the subroutine is represented as:

$$(+181.0600) \div (+56.4) = X.XXX \text{ and remainder}$$

To evaluate n, the shift-factor: n = the number of quotient digits - 1; since there are four quotient digits required in this example, n is equal to 3. The shift-factor 03 specifies to the Divide Subroutine that it must left-offset the units position of the divisor three positions with respect to the units position of the dividend prior to the first subtraction operation.

Conditions Governing Length and Position of Divisor and Dividend

The conditions governing the maximum length (i. e. , the maximum number of digits) and the position of divisor and dividend are as follows:

1. The length of the dividend field must be no greater than 20.

2. The length of the divisor field plus the shift-factor n must be no greater than 20.

3. The length of the dividend field must be greater than the shift-factor n.

4. There must be no more than nine repetitive subtraction cycles executed before the result of subtraction turns negative. This condition can occur, of course, only when developing the high-order quotient digit.

5. The number of significant digits (i. e. , the number of digits excluding leading zeros) in the dividend must not exceed the length of the divisor field plus the length of the quotient field.

A divisor whose value is zero is one special case which does not meet condition 4. Other cases which do not meet condition 4 occur if too small a value of the shift-factor n is specified.

If condition 4 above is not met, the Divide Subroutine exits, setting the overflow indicator and returns program control to the main program. The overflow condition will be detected only if a subsequent logic operation (i.e., test and branch on overflow) is performed by the main program before an add or subtract operation resets the indicator. The test and branch instruction shown on line 060 following the parameter specifications is recommended to detect an overflow condition.

To incorporate the Divide Subroutine in a main program, the user appropriately inserts the following instructions and parameters in his main program:

LINE	COUNT	LABEL	OPERATION	(A) OPERAND				(B) OPERAND				d
				ADDRESS	±	CHAR. ADJ.	RES.	ADDRESS	±	CHAR. ADJ.	RES.	
3 5	6 7	8 13	14 16	17 23			27	28 34			38	39
0 1 0	7		M V	*	-	3		D I V I D E	-	1		
0 2 0	4		B	D I V I D E								
0 3 0	3		D S A	*				(Divisor Label)				
0 4 0	3		D S A	*				(Dividend Label)				
0 5 0	2		D C W	*		x x						
0 6 0	5		B	E R R O R X								Z
0 7 0												

On line 050 above, xx represents the value of the shift-factor n. This subroutine requires approximately 415 storage locations.

The timing estimates for the Divide Subroutine are as follows:

> For a divisor of 10 digits and a dividend of 20 digits yielding a quotient of 10 digits each having an average value of 4.5, this subroutine requires approximately 71 milliseconds. In a manner analogous to Multiply I, the average value of a quotient digit is taken to be that value of the digit corresponding to the average division time. It lies halfway between digit values 4 and 5; hence it is equal to 4.5.

> For a divisor of 5 digits and a dividend of 10 digits yielding a quotient of 5 digits each having an average value of 4.5, this subroutine requires approximately 35 milliseconds.

As mentioned previously, modifying Divide for use as an open subroutine or modifying it to require no sign determination for the quotient reduces the amount of time and storage space otherwise required. The amount of time saved when Divide is made an open subroutine is approximately three milliseconds, or 4%; the space saved is approximately 125 locations, or 30%.

Modifying this subroutine to make no sign determination for the quotient reduces the time and space required compared to the unmodified version by approximately one millisecond, or 1%, and 75 storage locations, or 18%.

DOZENS-TO-UNITS CONVERSION SUBROUTINE

This subroutine converts dozens to units. The dozens field may range from 0/12 dozen to 9,999,999 11/12 dozens. The fractional portion of dozens is located in the low-order position of an eight-digit (maximum) field. The integers 0 through 9 denote the fractions 0/12 through 9/12 in the units position of the dozens field; the symbols minus (-) and ampersand (&) denote respectively the fractions 10/12 and 11/12 in the units position of the dozens field. The result of conversion is located in symbolic locations UNITS as a nine-digit number.

To incorporate Dozens-to-Units Conversion as a subroutine of a main program, the user appropriately inserts the following instructions and parameter in his main program:

LINE 3 ... 5	COUNT 6 ... 7	LABEL 8 ... 13	OPERATION 14 ... 16	(A) OPERAND				(B) OPERAND			
				ADDRESS 17	± 23	CHAR. ADJ.	RES. 27	ADDRESS 28	± 34	CHAR. ADJ.	RES. 38
0 1 0	7		M V	*	-	3		D Z C V R T	-	1	
0 2 0	4		B	D Z C V R T							
0 3 0	3		D S A	*				(Dozens field label)			
0 4 0											

This subroutine requires approximately 145 storage locations.

The estimates for timing for this subroutine are as follows:

> For the maximum size of the dozens field, 8 digits, yielding a maximum size of the units field, 9 digits, this conversion subroutine requires approximately three milliseconds.

> For values approximately one-half the maximum given above, the time required is approximately the same, being 3% less than that required for fields of maximum size.

As mentioned previously, making Dozens-to-Units Conversion an open subroutine reduces the time and storage otherwise required. The approximate reductions are slightly less than one millisecond (27%) and 45 storage locations (31%).

UNITS-TO-DOZENS CONVERSION SUBROUTINE

This subroutine converts units to dozens, including fractions of dozens 0/12 through 11/12. The units field may range from one to eight digits (a nine-digit units field is possible if it creates an eight-digit result including the fraction of dozens). The result of conversion is located in symbolic location DOZENS as an eight-digit number, whose units position integers 0 through 9 denote the fractional parts of dozens 0/12 through 9/12; and whose units position symbols minus (-) and ampersand (&) denote respectively the fractions 10/12 and 11/12.

To incorporate Units-to-Dozens Conversion as a subroutine of a main program, the user appropriately inserts the following instructions and parameter in his main program:

LINE	COUNT	LABEL	OPERATION	(A) OPERAND ADDRESS	±	CHAR. ADJ.	RES.	(B) OPERAND ADDRESS	±	CHAR. ADJ.	RES.
0 1 0	7		M V	*	−	3		U N C V R T	−	1	
0 2 0	4		B	U N C V R T							
0 3 0	3		D S A	*				(Units field label)			
0 4 0											

This subroutine requires approximately 205 locations in storage.

Timing estimates for this subroutine are as follows:

> For the maximum size of 8 digits in the units field, yielding a maximum of 7 "dozens quotient" digits, each having an average value of 4.5, this subroutine requires approximately 33 milliseconds. (For an explanation of the average value of 4.5, see the timing estimates for the Divide Subroutine, page 21.)

> For a units field one-half the maximum size, the time required is approximately 23 milliseconds.

As in the other subroutines presented in this bulletin, making Units-to-Dozens an open subroutine reduces the time and storage otherwise required. The approximate reductions are the same as those for the Dozens-to-Units Conversion Subroutine, being slightly less than one millisecond (3%) and 45 storage locations (22%).

PROGRAM ERROR-DETECTION AIDS

It is recommended practice, of course, to machine test data processing programs before putting them into productive operation. Programming errors discovered during program testing should be traced to find their cause. The user usually localizes the causes of program errors by a procedure such as comparing the actual contents of certain storage locations with the contents he expects from the previous steps of his program. He may, therefore, wish to insert halt instructions at strategic places in his program so that he may manually read out or print out storage contents; or he may wish to insert linkages into his program at convenient places to automatically print out blocks of storage contents, to aid in finding the first occurrence of a program error.

Several programs will be supplied to aid the user during program testing to detect the cause of program errors, if any exist. The program error-detection aids presented in this bulletin are:

  Insert Halts
  Insert Linkages to Fixed Print Storage
  Insert Linkages to Selective Print Storage
  Remove Linkages

Caution should be observed when selecting the instruction after which linkages to a routine are placed. For example, if a linkage to a print storage routine is placed after an unconditional branch instruction, the print storage routine will not be executed. If a linkage to a print storage routine is placed after a conditional branch instruction, the print storage routine will be executed only if the condition is not met. If a linkage to a print storage routine is placed after an instruction which is modified by another operation, the object program will not function properly and the print storage routine might not be executed.

All of the above programs are relocatable and self-loading. Each requires that the first card of the deck be a housekeeping card, which is supplied with the deck, and that the second card of the deck be a location designator card, which the user prepares. In the latter card the user specifies the starting location, represented by kkk, of the relocatable error-detection aid program. The format for the location designator card is as follows:

  Columns 33-39   L046183
  Columns 40-43   1001
  Columns 44-46   kkk   (starting location of relocatable program)

The instruction in columns 33-39 causes the starting location punched in columns 44-46 to be loaded into locations 181-183. When a relocatable program is being loaded, the loading program instructions (which are included in the program decks of self-loading programs) use the starting address to modify the appropriate storage addresses of program instructions and constants.

Each program is described separately; descriptions of the control cards necessary to specify the variables to adapt the error-detection aids to the user's needs and/or equipment are included.

24

INSERT HALTS

This program is designed to insert halt instructions into an object program. The user specifies in a control card the location and length (character count) of the instruction after which he desires to insert a halt instruction, and the location of the "patch" area. The patch consists of the instruction after which the halt is to be placed and the halt instruction; the patch area may be any available area in core storage large enough to contain these instructions. If more than one patch is desired, they may be placed in one patch area, as indicated below in the explanation of the control card, columns 5-7.

The halt instruction has either one of two formats: · xxx or · xxxb, where:

decimal is the operation code for stop.

xxx represents the machine address of the instruction in the object program to be executed next after executing the halt instruction. This address is provided automatically by the Insert Halts program.

b is one blank position required by the Insert Halts program for only the last halt instruction in the patch area.

The size of patch areas required will be illustrated by the following two examples:

If it is desired to insert a halt instruction after only the following object program instruction:

M 035 226

a patch area containing at least twelve storage locations is required, as follows: seven storage locations for the object program instruction above and five locations for the halt instruction. In this instance the halt instruction is required to contain one blank position because it is the last halt instruction in the patch area.

Using another example, if it is desired to insert halt instructions after each of the following object program instructions:

.
.

M 035 226

.

A 044 S56

..
.

E S56 299

.

/ 080

.

.

a patch area containing at least 42 storage locations is required. Of this number 25 locations are required for the object program instructions and 17 locations are required for the halt instructions. The 17 locations are required for three halt instructions with the format · xxx and the last halt instruction with the format · xxxb.

This program is self-loading and relocatable. It requires approximately 205 storage locations.
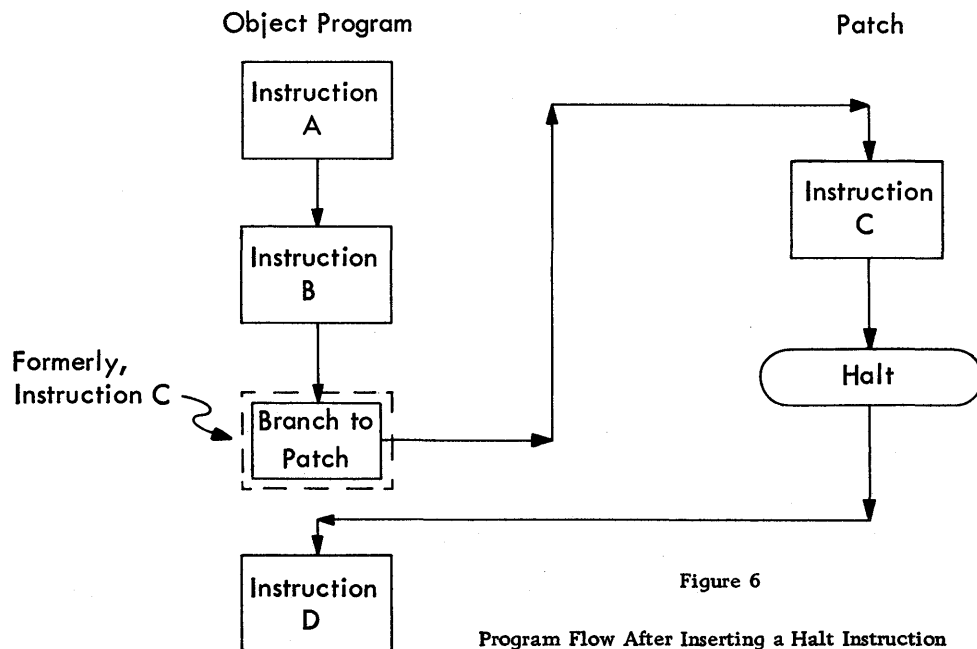
Control Card Format

1	2	3	4	5	6	7
	Location of Instruction Removed		Instr. Length		Address of Patch Area, or *bb	

Columns 1-3 are to be punched with the machine address in the assembled program of the instruction following which the halt instruction is to be placed. This instruction must be four to eight characters in length; it will be removed from its assembled location and placed in the patch area by the program.

Column 4 is to be punched with the length of the instruction to be removed to the patch area. The length, as stated above, may be from four to eight characters.

Columns 5-7 are to be punched with either the machine address of the patch area, or *bb. If the patch is to be placed behind the previous patch, column 5 should contain an asterisk punch and columns 6 and 7 should be blank.

Program flow after insertion of a halt instruction is illustrated in Figure 2 below.

Figure 6

Program Flow After Inserting a Halt Instruction

If an END card (optional) is desired to be used to begin execution of the object program after Insert Halts program execution is completed, its format should be /III080 in columns 1-7, which clears the read area to blanks and branches to location III, the starting address of the object program. If an END card is not used, locations 001-080 contain the word marks set by the self-loading Insert Halts program.

The order of the cards required for this routine is as follows: The first card is the housekeeping card, supplied with the Insert Halts program deck. The second card, prepared by the user, is the location designator card. The self-loading Insert Halts program deck supplied by IBM follows. The control cards, prepared by the user, follow the Insert Halts program deck. The optional END card, prepared by the user, is the last card.

The operating procedure for this routine is as follows: An object program into which it is desired to insert halt instructions is loaded into the 1401 system. The card deck for Insert Halts program, assembled in the order indicated in the previous paragraph (with the END card as the last card), followed by the data cards for the object program if they are required, are placed in the 1402 read feed hopper and loaded. At this point in the procedure the following program action has occurred:

1. The object program has been loaded.

2. The Insert Halts routine has been loaded and executed; i. e. , the linkages to the halt instructions have been inserted into the object program.

If the optional END card was used, as recommended, object program execution is begun automatically after the Insert Halts program has been executed. Execution of the object program continues, including reading of the data cards if the object program calls for this operation, until an inserted halt instruction stops the program. Upon the program's stopping, the user may take such action as he had planned; e. g. , he may check input locations, intermediate results, or output locations. Depression of the Start key on the 1401 Processing Unit console causes program execution to proceed. Object program execution continues until the next inserted halt instruction stops it, and so on until the program has been completely executed or until a programming error is encountered which cannot be corrected immediately.

After obtaining the desired information about the object program being tested with the aid of the Insert Halts program, the user may wish to restore the object program to its original condition; i. e. , to remove from the program the inserted halt instructions. He may do this by using another program, Remove Linkages, which is described on page 33.

INSERT LINKAGES TO FIXED PRINT STORAGE

This program inserts into an object program linkages to a routine which prints out the contents of storage between specified limits. The limits of storage printout are specified by the user in the print storage control card, the format of which is shown below.

The linkages to the print storage routine will be inserted by this program after any appropriate object program instruction of the user's choice. The program action in inserting the linkages of this routine is similar to that of Insert Halts program.

The Insert Linkages to Fixed Print Storage program consists of two routines, a patch routine (Phase I) and a print storage routine (Phase II). The print storage routine uses only one control card, while the patch routine may utilize several control cards because linkages may be inserted to print storage at several points in a program. Patch routine control cards are required to specify the points in the program at which storage printout is desired. Formats for the control cards are given below.

Control Card Format — Patch Routine (Phase I)

1	2	3	4	5	6	7
	Location of Instruction Removed		Instr. Length		Address of Patch Area, or *bb	

The fields of this control card are the same as those explained in the Insert Halts program, page 26.

Control Card Format — Print Storage Routine (Phase II)

1	2	3	4	5	6	7
Selected Block of Storage:						
	Lower Limit Address			Upper Limit Address		1403 Model Number

The above two machine addresses, columns 1-6, are specified by the user to cause the contents of the corresponding block of core storage to be printed out. In this routine the units and tens positions of the lower limit of the block of storage being printed determine the leftmost print position of the first printed line of the block. In like manner the units and tens positions of the upper limit determine the rightmost position of the last printed line. For example, if the lower limit and the upper limit of the block of storage being printed is, respectively, 623 and 974, the printed results would have the following format:

Line No.	Contents of Storage Locations	Print Position No.
1	623-700	23-100
2	(1s representing word marks for line 1)	
3	701-800	1-100
4	(1s representing word marks for line 3)	
5	801-900	1-100
6	(1s representing word marks for line 5)	
7	901-974	1-74
8	(1s representing word marks for line 7)	

Prior to execution of the object program (i. e. , the program to which this program error-detection aid routine is being applied), the contents of the block of core storage defined in the print storage control card is automatically printed out, together with a line indicating

the limits of storage printed and a heading line identifying the units positions of the printed output (i. e., 1234567890 and repeat, to identify print positions 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and so on to 100). Prior to each subsequent printout the object program instruction which was last executed is printed out.

Column 7 of the print storage control card specifies the model number of the 1403 printer. A 1-punch specifies model 1 printer, equipped with 100 print positions; a 2-punch specifies model 2 printer, equipped with 132 print positions.

If an END card (optional) is used to begin execution of the object program after execution of the Insert Linkages to Fixed Print Storage routine is completed, its format should be /III080 in columns 1-7, which clears the read area to blanks and branches to location III, the starting address of the object program. If an END card is not used, locations 001-080 contain the word marks set by the self-loading print storage routine.

After this program has inserted linkages into an object program, program flow proceeds as indicated in Figure 7.

The minimum size of the patch area is from 18 to 22 storage locations per object program instruction removed to the patch area depending upon the length of the instruction so removed.

The Insert Linkages to Fixed Print Storage program requires approximately 260 storage locations and prints storage at the rate of 600 lines per minute.

The order of the cards required for this routine is as follows:

1. Housekeeping card (first card)   Supplied with the program deck
2. Location designator card   Prepared by the user
3. Phase I program deck   Supplied by IBM
4. Phase I control cards   Prepared by the user
5. Phase II program deck   Supplied by IBM
6. Phase II control card   Prepared by the user
7. END card (optional) (last card)   Prepared by the user

The operating procedure for this routine is as follows: An object program is loaded into the 1401 system, and the Insert Linkages to Fixed Print Storage deck, assembled as described in the previous paragraph, is used to load that routine into the 1401 system. At this point in the procedure the following program action has occurred:

1. The object program has been loaded.

2. Phase I, the patch routine, has been loaded and executed. The linkages have been inserted into the object program.

3. Phase II, the print storage routine, has been loaded.

Next, the initial printout occurs, as follows: the first line indicates the storage limits of the printout, the second line is a heading line identifying the units positions of printed output (as explained above), and the third and succeeding lines, as required, contain

alternately the contents of storage and locations of word marks (as indicated on page 28).
If the optional END card, placed as the last card of the insert linkages deck, was used,
object program execution begins automatically. When the object program branches to
the patch area, it executes the instruction prior to the print storage linkages (e. g. ,
Instruction C, Figure 7) and executes the linkage instructions to the program error-
detection aid. The print storage routine prints the last object program instruction executed
(e. g. , Instruction C, Figure 7), a heading line identifying the units positions of the
printed output, and the contents of the specified block of storage. Each printed line of
storage contents is followed by a printed line indicating the word marks for that line, as
shown on page 28. The object program proceeds automatically, executing the next
instruction (e. g. , Instruction D in Figure 7); and so on until the object program has
been completely executed, printing out the instructions and storage contents as specified
by the user. Thus the Insert Linkages to Fixed Print Storage routine produces a printed
record of the contents of the storage locations specified by the user to aid in locating a
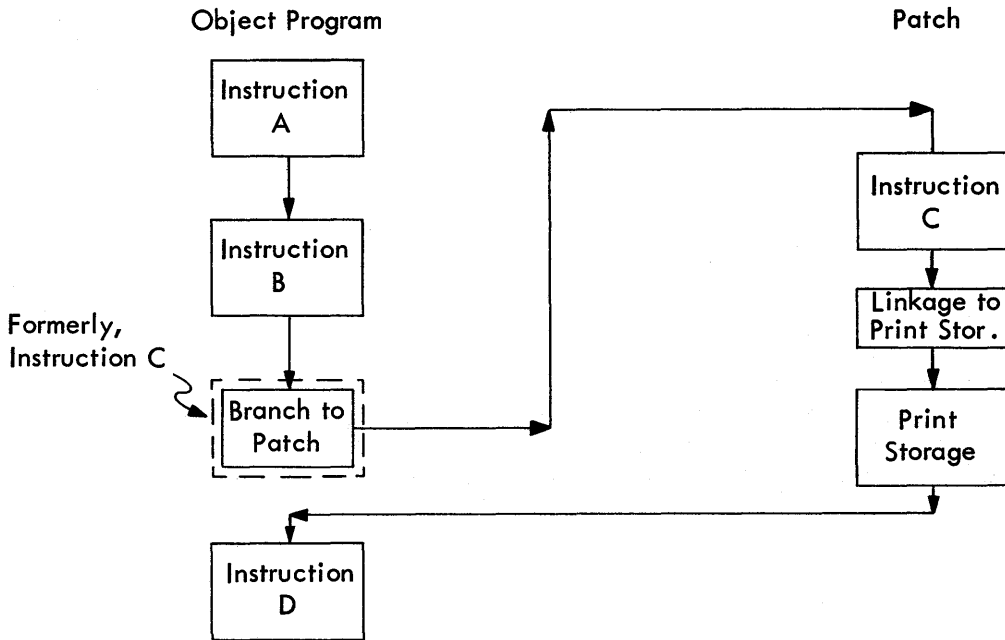programming error.

Figure 7

Program Flow After Inserting Linkages to Print Storage

INSERT LINKAGES TO SELECTIVE PRINT STORAGE

This program inserts into an object program linkages to a routine which prints out storage
contents between selective limits. The limits of storage printout using this routine is
specified for each instruction after which printout is to occur, whereas the limit of
storage printout using the Insert Linkages to Fixed Print Storage routine is specified
once for each object program application.

As with the Insert Linkages to Fixed Print Storage, the linkages to the print storage routine will be inserted by this program after any appropriate object program instructions of the user's choice. The program action in inserting the linkages is similar to that of Insert Halts program.

The Insert Linkages to Selective Print Storage program consists of two routines, a patch routine (Phase I) and a print storage routine (Phase II). The patch routine requires one or more control cards whose format is given below.

Control Card Format — Patch Routine (Phase I)

1 2 3	4	5 6 7	8 9 10	11 12 13
Location of Instruction Removed	Instr. Length	Location of Patch Area, or *bb	Selected Block of Storage: Lower Limit Address	Upper Limit Address

The first three fields, card columns 1-7, are the same as those explained in the Insert Halts program, page 26. Columns 8-10 and 11-13 specify, respectively, the lower and the upper limits of storage being printed. As in the Insert Linkages to Fixed Print Storage program, the units and tens positions of the lower limit address determine the leftmost print position of the first line, and the units and tens positions of the upper limit address determine the rightmost position of the last printed line.

Control Card Format — Initial Printout

1 2 3	4 5 6	7
Initial Storage Printout: Lower Limit Address	Upper Limit Address	1403 Model Number

The two machine addresses above in columns 1-6 specify the block of storage selected to be printed out prior to execution of the object program and column 7 specifies the 1403 model number. A 1-punch in column 7 specifies model 1, equipped with 100 print positions; a 2-punch specifies model 2, equipped with 132 print positions. If no initial storage printout is desired, a card with columns 1-6 left blank should be substituted for the card whose format is shown above.

As explained in the Insert Linkages to Fixed Print Storage program, if it is desired to use an END card (optional) to clear the read area to blanks and begin execution of the object program, its format should be /III080 in columns 1-7, III being the object program starting location. The flow of program control for this program is similar to that shown on page 30.

The minimum size of the patch area required is from 24 to 28 storage locations per object program instruction removed to the patch area depending upon the length of the instruction so removed.

The Insert Linkages to Selective Print Storage program requires approximately 300 storage locations and prints storage at the rate of approximately 600 lines per minute.

The order of the cards required for this routine is as follows:

1. Housekeeping card (first card)   Supplied with the program deck
2. Location designator card   Prepared by the user
3. Phase I program deck   Supplied by IBM
4. Phase I control cards   Prepared by the user
5. Phase II program deck   Supplied by IBM
6. Initial printout control card   Prepared by the user
7. END card (optional) (last card)   Prepared by the user

The operating procedure for this program is as follows: An object program is loaded into the 1401 system, and the Insert Linkages to Selective Print Storage program, the deck for which was described in the previous paragraph, is then loaded. At this point in the procedure the following program action has occurred:

1. The object program has been loaded.

2. Phase I, the patch routine, has been loaded and executed. The linkages have been inserted into the object program.

3. Phase II, the printout storage routine, has been loaded.

If a block of storage was specified in the initial printout control card in columns 1-6, an initial printout occurs as follows: the first line indicates the storage limits of the printout, the second line is a heading line identifying the units positions of printed output (as explained on page 29), and the third and succeeding lines, as required, contain alternately the contents of storage and locations of word marks (as indicated on page 28). If no block of storage was specified in columns 1-6 of the initial printout control card, no initial printout occurs. If the optional END card, placed as the last card of the insert linkages deck, was used, program execution begins automatically. When the object program branches to the patch area, it executes the instruction prior to the print storage linkages (e.g., Instruction C, Figure 7) and executes the linkage instructions to the program error-detection aid routine. The print storage routine prints on the first line the last object program instruction executed (e.g., Instruction C, Figure 7) and the printout storage limits specified for the instruction, on the second line the heading identifying the units positions of the printed output, and on the third and succeeding lines, as required, alternately the contents of the block of storage specified for the instruction and the locations of word marks (as shown on page 28). The object program proceeds automatically, executing the next instruction (e.g., Instruction D in Figure 7); and so on until the object program has been completely executed, printing out the instructions and storage contents specified by the user. Like the Insert Linkages to Fixed Print Storage program, this routine produces a printed record of the contents of storage locations specified by the user to aid in locating a programming error.

As explained in the Insert Halts and the Insert Linkages to Fixed Print Storage programs, the linkages may be removed from the object program by using the Remove Linkages program, described on page 33.

REMOVE LINKAGES

This program removes from an object program those linkages previously inserted to aid in detecting program errors. The linkages removed are replaced by the instructions previously removed from the object program. The control cards required for this program are the same control cards used with the programs (i. e. , Insert Halts, Insert Linkages to Fixed Print Storage, or Insert Linkages to Selective Print Storage) to insert the linkages into the object program. The Remove Linkages program is self-loading and relocatable. It requires approximately 160 storage locations and removes linkages at the rate of 800 cards per minute.

If an optional END card is used to begin, automatically, execution of the object program after execution of the Remove Linkages routine is completed, its format should be /III080 in columns 1-7. This instruction clears the read area to blanks and branches to location III, the starting address of the object program. If an END card is not used, locations 001-080 contain the word marks set by the self-loading Remove Linkages program.

In addition to removing linkages previously inserted into an object program by one of the insert linkage programs described in this bulletin, it may be convenient to insert new linkages, all in one machine pass. This may be done by using the Remove Linkages program. The following is an example of the order of cards required to remove linkages previously inserted into an object program by Insert Linkages to Fixed Print Storage and to insert new linkages using Insert Linkages to Selective Print Storage:

1. Housekeeping card for Remove Linkages program deck (first card) — Supplied with the Remove Linkages program deck

2. Location designator card — Prepared by the user

3. Remove Linkages program deck — Supplied by IBM

4. Control cards for Phase I and Phase II of the Insert Linkages to Fixed Print Storage program — Prepared by the user (the same control cards used to insert the linkages now being removed)

5. Housekeeping card for Insert Linkages to Selective Print Storage program — Supplied with the program deck

6. Location designator card for Phase I of the Insert Linkages to Selective Print Storage program — Prepared by the user

7. Phase I program deck for Insert Linkages to Selective Print Storage program — Supplied by IBM

8. Phase I control cards of the Insert Linkages to Selective Print Storage program — Prepared by the user

9. Phase II program deck of the Insert Linkages to Selective Print Storage program   Supplied by IBM

10. Initial printout control card of the Insert Linkages to Selective Print Storage program   Prepared by the user

11. END card (optional) (last card)   Prepared by the user

Program operation using the example just described in the paragraph above is as follows: Assuming that the object program with linkages previously inserted is stored in the 1401 system, the deck used with Remove Linkages routine (assembled as described in the previous paragraph) is loaded into the 1401 system. At this point in the operating procedure the following program action has occurred:

1. The Remove Linkages routine has been loaded and executed. The linkages have been removed from the object program and the object program has been restored to its original condition.

2. Phase I, the patch routine, of the Insert Linkages to Selective Print Storage program has been loaded and executed. The new linkages have been inserted into the object program.

3. Phase II, the printout storage routine, of the Insert Linkages to Selective Print Storage program, has been loaded.

4. If a block of storage was specified in the initial printout control card in columns 1-6, the contents of this block have been printed out, together with a printed line indicating the storage limits specified and a heading line identifying the units positions of printed output; otherwise no initial printout occurs.

If the END card, whose use is optional but recommended, is used as the last card of the Remove Linkages deck, program execution begins automatically. From this point on the operation is the same as that previously described for the Insert Linkages to Selective Print Storage routine.

34

IBM 1401 PUBLICATIONS

The following IBM 1401 Systems literature has been published as of the date of this bulletin:

Form Number   Title

GENERAL INFORMATION MANUAL

F20-0208   IBM 1401 Data Processing System
                  From Control Panel to Stored Program

BULLETINS

J28-0200   IBM 1401 Symbolic Programming System: Preliminary Specifications

J28-0207   Timing Estimates for the 1401 Generalized Sort Program

The following IBM 1401 Machine literature has been published as of the date of this bulletin:

GENERAL INFORMATION MANUAL

D24-1401   1401 Data Processing System

BULLETIN

G24-1402   Special Features

G24-1406   Increased Storage
                     IBM 1401 Models A4, B4, C4, D4
                     IBM 1401 Models A5, B5, C5, D5
                     IBM 1401 Models A6, B6, C6, D6