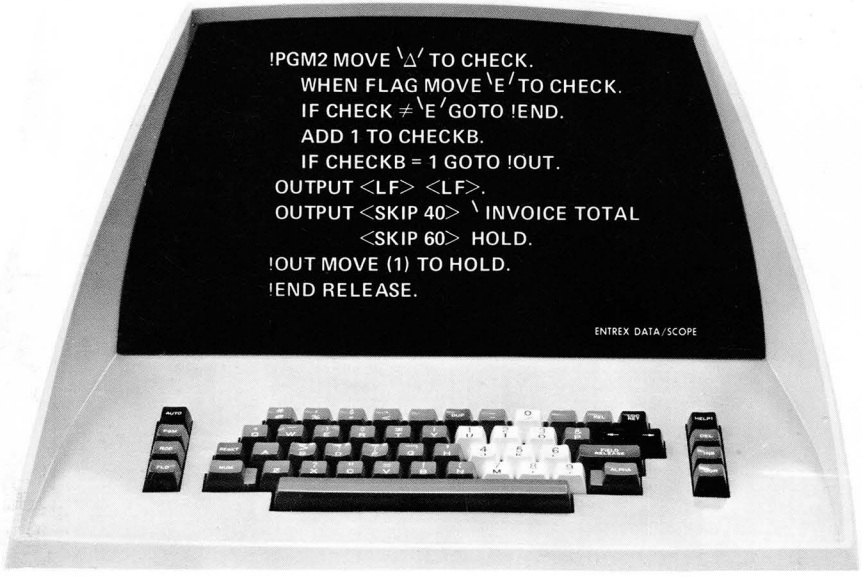
PROGRAMMER'S REFERENCE MANUAL

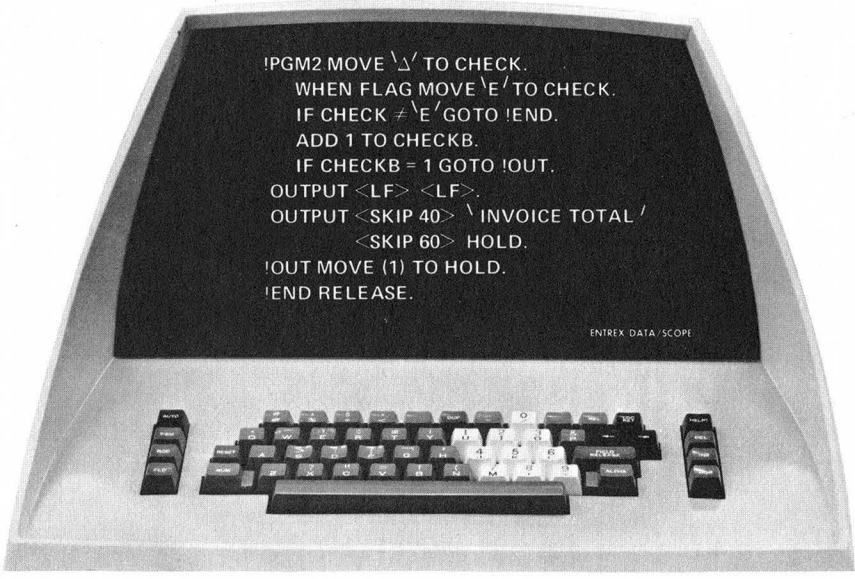


ENTREX SYSTEM 480

ORDER NUMBER S-15 **$9.00**

PROGRAMMER'S REFERENCE MANUAL (RELEASE VII)

MARCH 1974



ENTREX INC.
168 Middlesex Turnpike
Burlington, Mass. 01803

TABLE OF CONTENTS

SECTION 1 INTRODUCTION

SECTION 2 OPERAND TYPES

SECTION 3 INSTRUCTIONS

TABLE OF CONTENTS (Cont)

SECTION 4 PROGRAM GRAMMAR

SECTION 5 COMPILER ERROR CODES

PREFACE

This Programmer's Reference Manual describes the syntactic features of ENTREX's System 480 Validator language. With this language, extended data editing, output reformatting, and validating may be accomplished. This manual provides the basic program building blocks; examples and descriptions of how these are combined to form working routines and programs are given in the System 480 Formatting Techniques Manual (order no. S-13).

A publications "Comments Mailer Form" is included in this manual (last page). If errors, ambiguities, or inconsistencies are encountered, forward these comments to ENTREX, Inc. via this form.

SECTION 1
INTRODUCTION

ENTREX'S System 480 provides extended editing, validating, and output reformatting capabilities using its COBOL like VALIDATOR language. Three types of programs may be written – error detection, output reformatting and sorting. These routines and programs are written in a "free-form" style and may range from simple to complex depending on the User's application.

As with any high-level language, various techniques are employed to accomplish specific tasks. With the VALIDATOR language these tasks are accomplished using either a Record End Edit routine, Batch End Edit routine, sort routine, or output program. These are explained in detail in sections 5, 6, 7, and 8 in ENTREX's System 480 Formatting Techniques Manual (order no. S-13). However, a brief summary of each will be offered in the following paragraphs:

Record End Edit Routine

The key parameters of a Record End Edit Routine are:

- Used for simple range, crossfooting, contents, and extension checking,
- Performed when the entire record has been entered and released (Entry, Update, and Verify Mode (if correction is made)).
- Used exclusively for intra-record operations,
- Three variables are allowed in a Record End Edit routine,
- An error flag or error message is used to note errors found by the routine,
- Primarily used to check for operator entry errors.

Batch End Edit Routine

The key parameters of a Batch End Edit Routine are:

- Used for complex crossfooting, extensions, and range checks as well as batch totaling/subtotaling operations,

- Performed upon batch termination in all modes,
- Variables (accumulators) are cumulative (retains values from record-to-record) and up to 99 may be assigned. Variables are initialized (set to zero) when starting the batch,
- Three methods of specifying errors:
  - Error flag,
  - Error message and error tone,
  - Create a batch error log.
- Primarily used when errors will be corrected in a separate operation (*not* during entry).

Sort Programs

The key parameters of a sort routine are:

- Used to sort records within one or more batches in ascending or descending order.
- Performed upon supervisor request,
- Data can be validated while being sorted,
- Variables (or accumulators) are cumulative (retain values from record-to-record); up to 99 may be assigned. Variables are initialized (set to zero) when starting a batch,
- Errors can be flagged or the sort can be immediately terminated,
- Three methods of specifying errors:
  - Error flag,
  - Error message and error tone,

Output Programs

The key parameters of an output program are:

- Used for reformatting and outputting data to an output device,
- Performed upon supervisor request.

- Variables (or accumulators) are cumulative (retain values from record-to-record); up to 99 may be assigned. Variables are initialized (set to zero) when starting a batch,
- Three methods of specifying errors:
  - Error flag,
  - Error message and error tone,
  - Printed error listing.

PROGRAM CODING

System 480 programs and routines are written in a "free-form" style on ENTREX Editor Coding Forms (order no. M-102). The sheets consist of ten lines of forty characters each and represent one page or screen in the output format library. As many pages as required may be used.

MANUAL LAYOUT

Within the Validator language there are five operand types, these include: field, numeric literal, alphameric literal, variable and arithmetic expressions. These operand types are described in section 2.

The VALIDATOR language is composed of action (processing) and conditional instruction statements. Action instructions are used to perform arithmetic, editing, output, error signalling, and program control functions. Conditional statements perform logical and special test functions. These instructions are described in section 3.

Section 4 describes the VALIDATOR program grammar and punctuation rules. Section 5 provides compiler error codes and their definitions.

SECTION 2
OPERAND TYPES

INTRODUCTION

Within the VALIDATOR language there are five operand types that can be manipulated; these include: field, numeric literal, alphameric literal, variable and arithmetic expression. These will be described in the following paragraphs.

FIELD NUMBER

Field numbers can be from 1 to 2047 and must be enclosed in parentheses. For example:

- (5),
- (200),
- (2013).

A field number can further be defined to a subfield level. To manipulate data on the character level, the convention (n: P–Q) should be used, where:

- n – the field number,
- P – the first character position in the sub-field,
- Q – the last character position in the sub-field.

Sub-field examples include:

- (5: 2–4),
- (200: 70–98),
- (2013: 5–6).

NUMERIC LITERAL

A numeric literal is a string of digits not more than 14-characters long. If the number is a negative value, an oversign convention is used (e.g., $78\overline{6}$). The oversign can be positioned over any character in the string, except the first. Numeric literal examples include:

- 786,
- 56954,
- $6\overline{8}84914$,
- $123456789\overline{.}$

ALPHAMERIC LITERAL

An alphameric literal is an alphabetic character, a numeric character string, a word, or a sentence composed of *any* keyboard character up to 120 characters long. Alphameric literals are distinguished by quotation marks. These literals are used primarily for outputting messages or headers to the screen display or printing operations, respectively. Examples of alphameric literals include:

- 'EXTENSION ERROR – "PLEASE CHECK".'
- "BATCH TOTAL".
- "QUANTITY".

–NOTE–

Literal may also be enclosed by single quote. This is important if User wishes a double quote within the literal (e.g., 'EXTENSION ERROR – "PLEASE CHECK".').

Another alphameric literal convention used primarily in output operations is nnn 'x' where nnn is the number of times (1–120) the single character in quotes ('x') is repeated. A typical example would be 5 '0' instead of 00000.

–NOTE–

This convention is used to define a variable's size when used in conjunction with the MOVE verb.

VARIABLE

A variable is defined as a storage area which is set aside, contains some value, and is assigned a unique name. This value may be numeric (14-characters long) or alphanumeric (20-characters long). The variable name may be from one to eight-characters long, the first character being A–Z and the remaining characters being A–Z or 0–9.

Record End variables are initialized at *Record End* because inter-record operations are prohibited and are defined as 14-digit operands (alphameric data cannot be stored).

Batch End, Sort and Output variables are initialized at *Batch Start* or can be set to zero during operation, as follows:

MOVE 0 TO VARIABLE.

Variable types and sizes are defined by the MOVE statement as follows:

Operand	Variable Size	Variable Type
Field/Subfield	Operand's Size	Alphameric
Alpha Literal	Operand's Size	Alphameric
Numeric Literal	14 Characters	Numeric
Arithmetic Expression	14 Characters	Numeric
Variable	Operand's Size	Operand Type

–NOTE–

Arithmetic operations may be performed utilizing either numeric or alphameric variables, however, the destination variable type (after the arithmetic operation) will be changed to numeric and the logical size will remain unaltered.

The DECLARE instruction is used to define a variable as the first statement in a Record End Routine, Sort Routine, Batch End Routine, or Output Program, as follows:

DECLARE CUSNUBR, TOTAL, ..., SUBTOTAL.

ARITHMETIC EXPRESSION

An arithmetic expression is composed of the previously defined operands (Field Number, Numeric Literal, or Alphameric Literal) connected by a plus (+), minus (-), times (*) or divide (/) sign; for example:

- (2) + (3),
- QUANTITY * NUMBER,
- 2/(3) * TOTAL - (5).

Arithmetic expressions are written and processed from left to right, i.e., there is no hierarchy of operators.

SECTION 3
INSTRUCTIONS

INTRODUCTION

The VALIDATOR language is composed of action (processing) and conditional instruction statements. Action instructions are used to perform arithmetic, editing, output, error signalling, and program control functions. These instructions include:

- ADD
- BYPASS
- CLEAR
- DECLARE
- DIVIDE
- FLAG
- GOTO
- MOVE
- MULTIPLY
- OUTPUT
- PAUSE
- PERFORM
- RELEASE
- SORT
- STOP
- SUBTRACT

Conditional statements perform logical and special test functions. These instructions include: IF and WHEN. These instructions are described in the following pages.

ADD

DESCRIPTION:

The ADD instruction is used to add the contents of a field, literal, variable or arithmetic expression to the contents of a variable. The total is stored in the variable; the operand remains the same. At the conclusion of this operation, the destination operand (variable) is considered to be numeric, however, the logical size remains unchanged. If the physical size of the system accumulators (14 characters) is exceeded, results of this and future arithmetic operations are unpredictable.

FORMAT:

$$\text{ADD} \left\{ \begin{array}{l} \text{FIELD} \\ \text{LITERAL} \\ \text{VARIABLE} \\ \text{ARITH. EXPR.} \end{array} \right\} \text{TO} \left\{ \text{VARIABLE} \right\}.$$

CODING EXAMPLES:

- ADD (1) TO TOTAL.
- ADD (2) + (3) TO CREDITS.
- ADD WEIGHT TO RATE.
- ADD 1 TO COUNT.
- ADD (13:3–5) TO TEMP.

BYPASS

DESCRIPTION:

This instruction terminates processing of the current batch and initiates processing for the next batch in the name file. Most commonly it will be used in conjunction with a conditional statement to allow the testing of customer set switches. This function bypasses the entire data batch without stepping through each character individually.

FORMAT:

BYPASS {OPTIONAL AT END STATEMENT}.

CODING EXAMPLES:

- BYPASS.
- IF (1) = "Y" BYPASS.
- BYPASS, AT END OUTPUT <EOF> <RWD>.

CLEAR

DESCRIPTION:

The CLEAR instruction is used to insert a space in the left-most character position of any specified field, sub-field or into a character position, *if* that position contains an error flag. If the position does not contain an error flag, the position is left undisturbed.

FORMAT:

CLEAR $\left\{\text{FIELD}\right\}$.

CODING EXAMPLES:

- CLEAR (2).
- CLEAR (2:3).
- CLEAR (2:3–5).

DECLARE

DESCRIPTION:

The DECLARE instruction is used exclusively for assigning variables. Variables are assigned with the DECLARE statement as the first sentence in a program. This minimizes the possibility of referencing an invalid variable within an operating program. Three variables may be assigned for Record End Routines and 99 variables may be assigned for Batch End Routines, Sort Routines and Output Programs.

FORMAT:

DECLARE $\left\{\text{VARIABLE NAME}\right\}$, $\left\{\text{VARIABLE NAME}\right\}$, ..., $\left\{\text{VARIABLE NAME}\right\}$.

CODING EXAMPLES:

- DECLARE CUSNUBR.
- DECLARE TOTAL, SUBTOTAL, QUANTITY.

Notice that the variables are separated with a comma or a space and end with a period. This is practiced for readability purposes, however, there must be a space or comma (with no space) divider.

DIVIDE

DESCRIPTION:

The DIVIDE instruction is used to divide the contents of a field, literal, variable, or arithmetic expression into the contents of a variable. The total is stored in the variable and the contents of the operand remain the same. At the conclusion of this operation, the destination variable is considered to be numeric, however, the logical size remains unchanged. If the physical size of the system accumulators (14 characters) is exceeded, results of this and future arithmetic operations are unpredictable.

FORMAT:

$$\text{DIVIDE} \left\{ \begin{array}{l} \text{FIELD} \\ \text{LITERAL} \\ \text{VARIABLE} \\ \text{ARITH. EXPR.} \end{array} \right\} \text{INTO} \left\{ \text{VARIABLE} \right\}.$$

CODING EXAMPLES:

- DIVIDE (1) INTO TOTAL.
- DIVIDE (2) + (3) INTO CREDITS.
- DIVIDE WEIGHT INTO RATE.
- DIVIDE 100 INTO COUNT.
- DIVIDE (13:3–5) INTO TEMP.

FLAG

DESCRIPTION:

The FLAG instruction is used to insert an error character (#) into the left-most character position of any specified field, sub-field or into a character position. This instruction is usually used in conjunction with the IF or WHEN verbs.

FORMAT:

FLAG $\left\{\text{FIELD}\right\}$.

CODING EXAMPLES:

- FLAG (2).
- FLAG (2:3).
- FLAG (2:3–5).

GOTO

DESCRIPTION:

The GOTO instruction is used to branch or skip over/back to some other point in the program (designated by a "sentence label"). A sentence label is a word or an abbreviation which can be up to eight-characters long. The first character must be from A–Z; all others may be any alphanumeric character combination. In addition, the sentence label must begin with an exclamation point (!).

–NOTE–

A branch may be executed to anywhere within a program with the exception of into or out of the contents of a subroutine.

FORMAT:

GOTO { !SENTENCE LABEL } .

CODING EXAMPLES:

- GOTO !TEST1.
- GOTO !A103675.
- GOTO !PGM3.

IF

DESCRIPTION:

The IF statement is used for performing simple and compound logical comparisons. Comparisons are considered to be either alphameric or numeric. Alphameric comparisons are performed one character at a time from left to right. Should one operand be shorter than the other, it is assumed to be right space-filled to allow comparison for entire length of longer operand. All alphameric comparisons are made utilizing a standard EBCDIC collating sequence. In a numeric compare, operands which look different but have equal values are considered equal. For instance, '–ØØ21' is equal to '–21' and equal to '2J' (least significant digit oversign). When comparing an alphameric operand to a numeric operand, the comparison is numeric.

The results of the IF comparison are used to determine the logical direction of a program. If the comparison is true, the next *instruction* is executed; if it is false the next *sentence* is executed.

FORMAT (Simple IF Instruction):

IF { FIELD / LITERAL / VARIABLE / ARITH. EXPR. } { = / ≠ / > / < } { FIELD / LITERAL / VARIABLE / ARITH. EXPR. }

CODING EXAMPLES (Simple IF Instruction):

- IF (3) = (4)
- IF (1) ≠ 'TOTAL'
- IF TEMP < 99

FORMAT (Compound IF Instruction):

IF { FIELD / LITERAL / VARIABLE / ARITH. EXPR. } { = / ≠ / < / > } { FIELD / LITERAL / VARIABLE / ARITH. EXPR. } OR { = / ≠ / < / > } { FIELD / LITERAL / VARIABLE / ARITH. EXPR. }

IF
(continued)

CODING EXAMPLES (Compound IF Instruction):

- IF TEMP < 99 or > 70
- IF TEMP = 100 or > 50

FORMAT (Multi-Condition Compound IF Instruction):

$$\text{IF} \left\{\begin{array}{l}\text{FIELD}\\ \text{LITERAL}\\ \text{VARIABLE}\\ \text{ARITH. EXPR.}\end{array}\right\} \left\{\begin{array}{l}=\\ \neq\\ <\\ >\end{array}\right\} \left\{\begin{array}{l}\text{FIELD}\\ \text{LITERAL}\\ \text{VARIABLE}\\ \text{ARITH. EXPR.}\end{array}\right\} \quad \text{OR}$$

$$\text{IF} \left\{\begin{array}{l}\text{FIELD}\\ \text{LITERAL}\\ \text{VARIABLE}\\ \text{ARITH. EXPR.}\end{array}\right\} \left\{\begin{array}{l}=\\ \neq\\ <\\ >\end{array}\right\} \left\{\begin{array}{l}\text{FIELD}\\ \text{LITERAL}\\ \text{VARIABLE}\\ \text{ARITH. EXPR.}\end{array}\right\}$$

CODING EXAMPLES (Multi-Condition Compound IF Instruction):

- IF TIME = 1200 OR IF TEMP = 1800
- IF (1) = 'XY' OR IF (2) = 'AB'

To execute a numeric comparison between fields, the instruction must be coded as an arithmetic expression, for example:

- IF (3) = (4) — Alphameric – INCORRECT
- IF (3) * 1 = (4) — Numeric – CORRECT
- IF (3) + 0 = (4) — Numeric – CORRECT

If a *series* of IF instructions (e.g., IF N = Y and IF A + B = C) is required, the two instructions are separated by a comma(,) or a space. This format implies a logical AND as follows:

- IF TOTAL = 100, IF COUNT = 31
- IF (4) + 0 = (76), IF END = 76

MOVE

DESCRIPTION:

The MOVE instruction moves data into a specified variable. When this instruction is executed, the contents of the operand *overlay* the contents of the variable, and the operand's contents remain the same. For example, if TEMP is equal to 99 and TODAY is equal to 76, when TEMP is moved to TODAY – TEMP would equal 99 and TODAY would equal 99.

One property of the MOVE statement is that it may be used to define the logical size and type (alpha or numeric) of a variable. This is simply done by allowing the variable to take on the attributes of the data being moved to it.

FORMAT:

```
MOVE  { FIELD      }      TO    { VARIABLE } .
      { LITERAL    }
      { VARIABLE   }
      { ARITH. EXPR. }
```

CODING EXAMPLES:

- MOVE (1) TO TOTAL.
- MOVE (2) + (5) TO CREDITS.
- MOVE WEIGHT TO RATE.
- MOVE 1 TO COUNT.

MULTIPLY

DESCRIPTION:

The MULTIPLY instruction is used to multiply the contents of a field, literal, variable, or arithmetic expression times the contents of a variable. The total is stored in the variable with the contents of the operand remaining the same. At the conclusion of this operation, the destination variable is considered to be numeric. However, the logical size remains unchanged. If the physical size of the system accumulators (14 characters) is exceeded, results of this and future arithmetic operations are unpredictable.

FORMAT:

MULTIPLY { FIELD / LITERAL / VARIABLE / ARITH. EXPR. } TIMES { VARIABLE } .

CODING EXAMPLES:

- MULTIPLY (1) TIMES TOTAL.
- MULTIPLY (2) + (3) TIMES CREDITS.
- MULTIPLY WEIGHT TIMES RATE.
- MULTIPLY 1 TIMES COUNT.
- MULTIPLY (13:3–5) TIMES TEMP.

OUTPUT

DESCRIPTION:

The OUTPUT instruction is used to reformat and output data (batches, records or fields) to tape or other output devices. For example, this instruction may be used to generate error listings, headers (column headings or printouts), printouts of data entered, etc. In addition, control functions and operand modifiers are used in conjunction with the instruction to format outputs; these will be described on the following pages. The physical output occurs only at the end of the execution of the whole statement. Consequently, one OUTPUT statement, unless deferred (page 3–18), creates one logical record.

FORMAT:

OUTPUT { FIELD / LITERAL / VARIABLE / CONTROL } , { FIELD / LITERAL / VARIABLE / CONTROL } , ... { FIELD / LITERAL / VARIABLE / CONTROL } .

CODING EXAMPLES:

- OUTPUT (1), (2), (3).
- OUTPUT 'ENTREX'.
- OUTPUT (1), <LF>, (2),<LF>, (3), <TOP>.
- OUTPUT FILENAME.

OUTPUT
(operand modifiers)

DESCRIPTION:

Operand modifiers may be used to further define any field, variable, or arithmetic expression to allow for character editing. An operand modifier consists of a vertical bar (|) followed by an edit specification, and immediately follows the operand which it modifies.

FORMAT:

OUTPUT { FIELD / VARIABLE } { OPERAND MODIFIER }

The following are legal edit specifications:

- |PK – Packed decimal format
- |LS – Truncate all leading spaces
- |LZ – Truncate all leading zeroes
- |SG – Positively oversign last digit in field
- |TS – Truncate all trailing spaces
- |TZ – Truncate all trailing zeroes
- |'MASK' – Where MASK is an alphameric literal whose largest size is 2Ø characters including all MASK characters. If the operand is longer than the MASK, it will be truncated and any floating or fixed dollar sign will be lost. Table 3–1 provides a description of the legal characters used with the mask edit. Table 3–2 provides examples of the mask edit specification. It should be noted that zone portions of all characters in a masked operand are stripped, yielding only the non-overpunched digits Ø–9.

OUTPUT
(mask characters)

TABLE 3–1. LEGAL MASK CHARACTERS

Character	Description
_	An underscore in the edit mask is replaced by the corresponding digit from the specified variable.
Ø	A zero is used to indicate zero suppression. It is placed in the right-most position where zero suppression is to take place. It is replaced with the corresponding character from the variable unless that character is a zero.
*	An asterisk is used for asterisk protection and zero suppression. It is put in the right-most position where asterisk protection is required.
$	A floating dollar sign is used for zero suppression code and causes the insertion of a dollar sign in the position to the left of the first significant digit. A dollar sign in the left-most position of the MASK is considered fixed. A fixed dollar sign is placed in the same location each time.
. , ƀ	Decimal points, commas and blanks are placed in the output field in the relative positions they were written in the MASK unless they are to the left of significant digits.
CR –	The characters CR or a minus sign in the last positions of the edit MASK are undisturbed if the sign of the variable field is negative. If the sign is plus the CR or minus sign is blanked out.

–NOTE–

Zero, asterisk and floating dollar sign are mutually exclusive. If two or more should occur, the one in the least significant position will take precedence.

OUTPUT
(mask examples)

TABLE 3–2. MASK EDIT EXAMPLES

		Result	
Mask	± Operand	+Data	–Data
'_ _ _0._ _'	000005	.05	.05
'_ _$._ _'	000005	$.05	$.05
'$_ _0._ _'	000005	$␣␣␣.05	$␣␣␣.05
'$_ _*._ _'	000005	$***.05	$***.05
'_ _ _._ _-'	13560	135.60␣	135.60–
'_ _ _._ _CR'	13560	135.60␣␣	135.60CR
'_ _ _._ _␣CR'	13560	135.60␣␣␣	135.60␣CR
'$_ _ _0*._ _-'	149363	$*1493.63␣	$*1493.63–
'$_ _ _*0._ _-'	149363	$␣1493.63␣	$␣1493.63–
'_ _ _,_$0._ _ _-'	1763421	1,763.421␣	$1,763.421–
'_ _ _$␣0._ _CR'	17631	17␣6.31␣␣	$17␣6.31CR
'_ _0_ _ _'	000005	005	005

OUTPUT
(control functions)

DESCRIPTION:

Control functions may be defined as program control verbs (programming short cuts), e.g., OUTPUT <ALL>; or instructions which control the physical aspects of output devices, e.g., OUTPUT <TOP>, <RWND>, etc. These functions may be used at any time within the OUTPUT instruction and are enclosed in less-than (<) and greater-than (>) symbols.

FORMAT:

OUTPUT { <CONTROL FUNCTION> } .

CODING EXAMPLES:

The following paragraphs will describe the legal control functions used in the System 480 Validator language.

<ALL mm–nn>

This control function is used for outputting specific groups of fields or an entire file as entered (i.e., no reformatting). The above format is used to output multiple fields. For example, to output fields 2 through 6 one would code:

OUTPUT <ALL 2–6>.

If nn is not specified, field mm through the end of the record is outputted; for example:

OUTPUT <ALL 2>.

If neither mm nor nn is specified, the entire record will be outputted; for example:

OUTPUT <ALL>.

OUTPUT
(control functions)

<APPEND>

This function is used to "APPEND" or add data to a tape that was previously written on the System 480. This may be desirable in two different instances:

- To append data to a tape using consecutive write-to-tape operations; i.e., output two standard jobs back-to-back with no intervening tape marks, labels, etc. Hereafter referred to as consecutive appending.
- To append data to tape that was written previously (day before, etc.). Hereafter referred to as non-consecutive appending.

Obviously, there are two different procedures used depending on the instance with which one is faced, as follows:

- Consecutive appending – In this situation, the tape will be positioned after the *last* data block. Therefore, one would code:

  WHEN START OUTPUT <APPEND>.

- Non-consecutive appending – In this situation, one would load the tape to be appended, and either:

  – Position it after the first tape mark and code:

    WHEN START OUTPUT <BSP 1> <APPEND>.

  – Position it after the first tape mark, manually backspace one record and code:

    WHEN START OUTPUT <APPEND>.

–NOTE–

1. *<APPEND> is ignored at Beginning of Tape (BOT).*
2. *A successful <APPEND> execution turns off the When Start indicator.*

Considering the above notes, one may program for either – at BOT or within the data, as follows:

WHEN START OUTPUT <BSP 1> <APPEND>.
WHEN START OUTPUT <LABEL> (New Data) <EOF>.

If at BOT the following occurs:

- <BSP 1> is ignored,
- <APPEND> is ignored,
- When Start indicator remains *on*,
- Second WHEN START statement is executed.

If not at BOT the following occurs:

- <BSP 1> is executed,
- <APPEND> is executed,
- When Start indicator is turned *off*,
- Second WHEN START statement is ignored.

Using the <APPEND> function the following restraints must be observed:

- All output parameters of appended data must conform to those of the existing data.
- User cannot precede a <LABEL> by more than two tape marks.
- User cannot use more than five User labels following last data block.

<BATCH>

This control function causes the *current* batch name to be inserted into the next ten character positions of the output record.

OUTPUT
(control functions)

<BLK n>

The BLK function allows the EDITOR to output the physical tape block count 'n' specifies the size of the field in characters within which the count is output. If n is smaller than the actual count the count will be truncated. If n is zero there is no output regardless of the count.

–NOTE–

n must be equal to or less than 5.

<BSP nnnn>

This function is used for tape positioning (on a block level). When executed, the tape will be "backspaced" nnnn blocks or to the BOT marker, whichever comes first (nnnn can be any number from 1 through 2047). This function is usually used in conjunction with the <APPEND> function, for example:

- OUTPUT <BSP 3><ALL>.
- OUTPUT <BSP 1><APPEND>.

<COUNT xxxx>

This function outputs the number of characters contained within each record. The count may be in either binary or decimal digits from one to four character with leading or trailing spaces but no imbedded spaces. In the above format "x" may be either a "B", "D" or "S", as shown below:

- DDSS Two digit decimal count with two trailing spaces,
- BBSS Two digit binary count with two trailing spaces,
- DDD Three digit decimal count,
- SSBB Two digit binary count with two leading spaces.

OUTPUT
(control functions)

< DATE x >

This function accesses the system Global date which consists of six characters in a format entered by the User. "x", shown in the above format provides the capability to format the data field as follows:

- Left blank – six character formatless.
- Any legal keyboard character (except an underscore) – this option creates the format mmXddXyy.
- Underscore – this inserts blanks as follows: mm␣dd␣yy.

<DEFER>

This function provides the capability to close files on tape. If fixed length has been specified, it will pad out the current block, and write an industry compatible tape mark. If not fixed, it will write a short block followed by an industry compatible tape mark. If the tape drive is not the output device, the instruction is ignored.

< EOF >

This function provides the capability to close files on tape. If a pad character has been specified, it will pad out the current block, and write an industry compatible tape mark. If no pad character has been specified it will write a short block followed by an industry compatible tapemark. If a tape drive is not available, the instruction is ignored.

< HEX xx >

The HEX feature allows eight-bits to be outputted (described by the HEX convention) without going through code conversion. Therefore, 256 possible combinations of eight-bits may be outputted.

< JOB >

This function causes the standard job name used to enter the current batch to be inserted into the next eight character positions of the output record.

OUTPUT
(control functions)

< LABEL >

This function may appear anywhere within an OUTPUT statement and is used to specify that the current record is a label and not a data record. The occurrence of such a record would cause the output buffer to be handled as if an EOF were encountered, with the exception of writing a tape mark, after which the label would be output regardless of any specified blocking options.

< LF >

This function initiates a line feed and carriage return to be executed by the printer.

–NOTE–

A line feed will be initiated by the OUTPUT instruction itself. Therefore, "OUTPUT <LF> <LF>." would initiate three line feeds.

< PGM >

This function initiates the current record's input format number to be inserted in the output record.

< RWND >

This function initiates the tape drive to execute an unconditional tape rewind. If no tape is mounted the instruction is bypassed.

< SKIP nnnn >

This function inserts blanks into the output record starting from the current character position up to and including the character position specified by "nnnn".

< TOP >

This function initiates the printer to skip over the remaining lines on the page to the top of the next page on a printout.

PAUSE

DESCRIPTION:

The PAUSE instruction is used for error signalling. When this instruction is executed, an error message (up to 40 characters) is displayed, and an error tone is sounded. This instruction is used most often in conjunction with a conditional statement. If an alphameric literal (error message) is not coded, the message "PAUSE" is displayed.

FORMAT:

PAUSE { ALPHAMERIC LITERAL } .

CODING EXAMPLES:

- PAUSE 'EXTENSION ERROR – PLEASE CHECK'.
- IF RECNO = 1000, PAUSE 'OVER 1000 RECORDS IN BATCH'.

PERFORM

DESCRIPTION:

The PERFORM instruction, a program control instruction, is used to execute a specific subroutine within a program. When the subroutine has been performed, program control will return to the instruction immediately following the PERFORM statement. There are various constraints associated with the PERFORM instruction – as follows:

- A "sentence label" and the special words ENTER and EXIT must be used in conjunction with this instruction.
- A program branch (GOTO) *within* the specified confines (between ENTER and EXIT) of a subroutine is *legal.*
- A program branch (GOTO) *out of* a subroutine is *illegal.*
- A program branch (GOTO) from outside a subroutine to a statement within a subroutine is *illegal.*

FORMAT:

PERFORM !SENTENCE LABEL, ENTER { SUBROUTINE }. EXIT.

CODING EXAMPLES:

- PERFORM !TEST.
  - !TEST, ENTER WHEN PGM1 OUTPUT "1". EXIT.
- PERFORM !FINI.
  - !FINI, ENTER OUTPUT '!INVOICE INVERRORS'. EXIT.

–NOTE–

Notice that a period is required at the end of each sentence in the subroutine and after exit.

RELEASE

DESCRIPTION:

When this instruction is executed, the current record is released and the next record will be called-in; also, the program will branch to the beginning of the program for further processing.

If a RELEASE is not encountered, the function will be performed after the last statement in the program. In this case, a STOP function is implied when the last record has been processed. However, a "RELEASE AT END" statement may be incorporated as the last statement in the program.

FORMAT:

RELEASE { OPTIONAL AT END STATEMENT } .

CODING EXAMPLES:

- RELEASE.
- RELEASE, AT END GOTO !FINI.
- RELEASE, AT END OUTPUT <EOF> <RWD>.

SORT

DESCRIPTION:

The SORT instruction is used to sort a file of records by field, literal, control function, or variable in ascending or descending order. Ascending and descending sorts may be intermixed within a SORT statement. In addition, all control functions and address modifiers allowed by the OUTPUT statement are allowed by the SORT statement. Two address modifiers can be used in conjunction with this instruction:

- |AK – specifies ascending sort.
- |DK – specifies descending sort.

These modifiers are coded directly behind the operand:

SORT (1) |DK '1'.

–NOTE–

In the absence of either of the above modifiers, the sort defaults to ascending.

FORMAT:

SORT { FIELD / LITERAL / VARIABLE / CONT. FUNCT. } , { FIELD / LITERAL / VARIABLE / CONT. FUNCT. } , ... { FIELD / LITERAL / VARIABLE / CONT. FUNCT. }

CODING EXAMPLES:

- SORT INVNUMBR '1'.
- SORT <PGM>, (1) |AK (6:3).

STOP

DESCRIPTION:

The STOP instruction is used to halt a program. If the STOP instruction is not coded, the program will stop when the last record has been processed. The format for this instruction is given below.

–NOTE–

This instruction should not be used unless an abnormal break is desired in the program.

FORMAT:

```
STOP.
```

CODING EXAMPLES:

- STOP.
- WHEN OVERFLOW, STOP.

SUBTRACT

DESCRIPTION:

The SUBTRACT instruction is used to subtract the contents of a field, literal, variable, or arithmetic expression from the contents of a variable. The total is stored in the variable with the operand contents remaining the same. At the conclusion of this operation, the destination operand (variable) is considered to be numeric, however, the logical size remains unchanged. If the physical size of the system's accumulators (14 characters) is exceeded, results of this and future arithmetic operations are unpredictable.

FORMAT:

SUBTRACT { FIELD / LITERAL / VARIABLE / ARITH. EXPR. } FROM { VARIABLE } .

CODING EXAMPLES:

- SUBTRACT (1) FROM TOTAL.
- SUBTRACT (2) + (3) FROM CREDITS.
- SUBTRACT WEIGHT FROM RATE.
- SUBTRACT 1 FROM COUNT.
- SUBTRACT (13:3–5) FROM TEMP.

WHEN

DESCRIPTION:

The WHEN instruction functions exactly as does the IF instruction except that it tests certain "conditions" within the system as opposed to logical relationships within the data. These conditions include:

- WHEN BATCH – is used to test for the batch's first input record. This facilitates batch initialization procedures when multiple batches are being processed via the asterisk convention.
- WHEN EOT – checks for an "end-of-tape" marker during output operations.
- WHEN FLAG – this statement is used to check for the existence of an error flag within the current record. If possible, use this statement sparingly as it decreases system efficiency.
- WHEN OVERFLOW – The WHEN OVERFLOW statement is used for checking for *logical* arithmetic overflow. It refers to the last arithmetic operation that took place, and applies to operands which are arithmetic expressions as well as the ADD, SUBTRACT, MULTIPLY and DIVIDE instructions.
- WHEN (NOT) PGM n (where n = 0–9) – this instruction checks the current record's program level (input format number). This instruction can be written for negative logic programming (e.g., WHEN NOT PGM 2).
- WHEN START – tests for the first input record of a file.

–NOTE–

It is important to note that arithmetic overflow occurs in two different forms – logical and arithmetic. Logical overflow occurs when a number (within a variable) exceeds that specified in the associated MOVE statement. In this case the number will be truncated to the specified parameters. Physical overflow can not *be checked with WHEN OVERFLOW and occurs when the system encounters a number which is greater than 14 digits. When this occurs, the system displays "SYSTEM ARITHMETIC OVERFLOW".*

WHEN
(continued)

FORMAT:

WHEN { CONDITION }.

CODING EXAMPLES:

- WHEN BATCH, OUTPUT '!' <JOB> <BATCH>.
- WHEN EOT, PAUSE 'MOUNT NEW TAPE'.
- WHEN FLAG, GOTO !ERROR.
- WHEN OVERFLOW, PAUSE 'EXCEEDED 999'.
- WHEN PGM 2 SORT INVNUBR '1'.
- WHEN NOT PGM 1 GOTO !DONE.
- WHEN START, PERFORM !HEADER.

SECTION 4
PROGRAM GRAMMAR

INTRODUCTION

When writing a computer program to a specific computer language, certain grammatical rules must be observed. System 480 VALIDATOR language is no exception to this rule. The VALIDATOR grammar and punctuation rules are described in the following paragraphs.

Sentence Structure

In all cases, a program sentence must contain *one* and *only* one action instruction. Any number of conditional instructions (IF and WHEN) can precede the action instruction, for example:

- IF (1) = (2). (Incorrect – no action instruction).
- IF (1) = (2), ADD (5) TO TOTAL (Correct).

An exception to this rule is the RELEASE instruction.

When the System 480 processes a conditional instruction, one of two executions will occur, depending on whether the statement proves true or false:

- TRUE – The System 480 will execute the next instruction,
- FALSE – The System 480 will branch to the next sentence disregarding the action instruction.

An example of this program sequence is:

- Line 1 – WHEN PGM 3 GOTO !PRINT.
- Line 2 – WHEN PGM 4 GOTO !OUT.

If the current record is Program Level 3, the program will branch to !PRINT, however, if the current record is not Program Level 3, the program will disregard "GOTO !PRINT" and branch to Line 2.

Sentence Punctuation

Periods (.) are used as a sentence delimiter. It is critical that the period be used correctly to insure that sentences with conditional statements are executed properly. *All* sentences must end with a period.

Commas are commonly used to separate statements. When separating two conditional statements, the comma implies a logical "AND". Commas may also be used to separate sentence labels from sentences as well as anywhere they make sense and are used primarily for program legibility.

Spaces must separate all instructions, arithmetic operators, and operands; a rule that is similar to the English language.

Sentence Labels

Sentences may be preceded with a label so that they may be branched to with a GOTO statement or called with a PERFORM statement. A label must be immediately preceded by exclamation point and may be up to 8-characters long with the first character being A–Z and all other characters A–Z or 0–9.

VALIDATOR PROGRAMMING EFFICIENCIES

System 480 programming can be best accomplished using good programming techniques (Section 4, System 480 Formatting Techniques Manual) and bearing in mind the System 480 idiosyncrasies described in the following paragraphs.

Program size (binary) can be determined using the data shown in Table 4–1.

TABLE 4–1. BINARY SIZES

Element	Binary Words
Field	1
Sub-field	2
Variable	1
Literals	
Numeric	4
Alpha	(CHARS/2) + 1
Repetitive Alpha	2
Operand Modifiers	
'Mask'	(CHARS/2) + 2
Others	1
Control Functions	
Without Descriptors*	1
With One Descriptor	2
With Two Descriptors	3
Action Verbs	1
Conditional Verbs	1
Arithmetic Operators (+, -, *, /)	1
Label (In GOTO/PERFORM)	2

*Descriptors modify the control function, e.g., SKIP may have one descriptor; ALL may have two.

Generally, the larger the binary program, the slower it will execute. Programs are disk resident and are read into core 120 words at a time. Only 120 words are kept in core at any one time, therefore, each increment of 120 will slow the program down. Conversely, if the program is less than or equal to 120 words, the interpreter will execute at peak efficiency. Therefore, a Record End Edit Routine should be limited to 120 words.

Whenever possible, use numeric literals in place of alphameric literals since they take up less space in the binary program.

Use as few variables as possible during Batch End/Sort/Output processing. Processing speed will be degraded for every new 11 variables. That is, processing speed will *not* vary if using 1 or 11 variables and will not be degraded again until 23 variables are used.

Use short messages whenever possible.

In output statements use the repetitive alpha literals whenever possible (i.e., 20 '0' instead of '00000000000000000000').

Use arithmetic expressions whenever possible:

- Instead of:
```
MOVE (1) TO PRODUCT
MULTIPLY (2) TIMES PRODUCT.
IF PRODUCT ≠ (3), PAUSE.
```
- Use:
```
IF (1) * (2) ≠ (3), PAUSE.
```

Use common programming techniques such as subroutining. Position least often used subroutines towards end of program. This could, conceivably, save disk accesses; for example:

```
WHEN START, PERFORM !SETUP.
```

!SETUP should be the last subroutine in the program since it will be used only once.

SECTION 5
COMPILER ERROR CODES

INTRODUCTION

When an output program, Batch or Record End routine, or a sort routine has been keyed into the system and terminated, the system does a quasi-COBOL compile. If any coding or keying errors are found, these are displayed in the following format:

PAGE NO. – LINE NO. – ERROR CODE NO.

Error code definitions are given in numeric sequence below:

–NOTE–

If an error is encountered from error codes 1–55, the entire statement will be compiled. However, if an error is encountered from error codes 71 through 83, the compilation for this statement will end.

NUMBER	DEFINITION
01	Numeric literal illegal operand within context of statement.
02	Alpha literal illegal operand within context of statement.
03	Field illegal operand within context of statement.
04	Variable illegal operand within context of statement.
05	Arithmetic expression illegal operand within context of statement.
06	Operand modifier illegal within context of statement: An operand modifier is legal only in an OUTPUT or SORT instruction.
07	Control function illegal within context of statement: A control function is legal only in an OUTPUT and SORT instruction.
08	Size of numeric literal exceeded: A numeric literal can contain no more than 14 digits.

NUMBER	DEFINITION
09	Operand type cannot be determined.
10	Alpha literal format error: An opening quote character can not be immediately followed by a closing quote character. At least one character must be inserted between the quotes.
11	Size of alpha literal exceeded: An alpha literal cannot contain more than 120 characters (excluding quotes).
12	Repetitive alpha literal format error: Only one character is permitted within quotes. The opening and closing quote characters must be identical both single or double.
13	Repetitive alpha literal format error: The repeat count cannot be greater than 120.
14	Size of variable name exceeded: The name of a variable cannot contain more than 8 characters.
15	Illegal use of EXIT instruction: An EXIT instruction must be the first and only instruction of a sentence.
16	Illegal use of a DECLARE instruction: A DECLARE instruction must be the first and only instruction of a sentence.
17	Field format error: A field number of no greater than 2047 can be specified. The field number must be numeric only (0–9).
18	Field format error: A field number can be followed only by a right parenthesis or a colon, if a sub-field is specified.

NUMBER	DEFINITION
19	Sub-field format error: The correct format for a sub-field specification is: (starting character position) – (ending character position) A character position may be any number from 1 through 99. The starting character position must be less than or equal to the ending character position. If the two character positions are equal, an ending character position specification is not necessary. In this case the dash (–) must not appear. A sub-field specification must be followed by a right parenthesis.
20	Operand modifier specification error: The EDIT MASK or PACK operand modifier must be the last modifier of a string of modifiers. This implies that a string of modifiers may not contain both an EDIT MASK and a PACK modifier.
21	Operand modifier type cannot be determined.
22	Size of variable name exceeded: The name of a variable used in a DECLARE instruction can not contain more than eight characters.
23	COUNT control function format error: The COUNT function requires an operand.
24	BLK control function format error: The BLK function requires an operand. This operand is a number which may not exceed the maximum allowable character size (5).
25	HEX control function format error: The HEX function requires an operand. This operand must be two consecutive hexadecimal characters.
26	ALL control function format error: The use of an operand in an ALL function is optional. When specified, it must be a number no greater than the maximum allowable field number (2047). If both a starting and ending field number are specified, they must be separated with a dash (–); the last field number must be greater than the starting field number.

NUMBER	DEFINITION
27	BSP/SKIP control function format error: The BSP/SKIP function requires an operand. The operand must be a number no greater than the maximum allowable field number (2047).
29	Control function format error: A control function specification must be terminated with a greater than symbol (>).
30	Control function type can not be determined.
32	DECLARE instruction format error: All variable names within a DECLARE instruction must be separated by either a comma, space or both. The last variable must be followed by a period. The first character of a variable name must be an A through Z; all following characters must be an A through Z or 0–9.
33	Sentence label format error: A label preceding a sentence can not contain more than eight characters, excluding the exclamation point. The first character of the label must be alpha (A–Z) and must immediately follow the exclamation point.
34	Sentence label format error: A label which precedes a sentence must be separated from the sentence by either a space, comma or both.
35	Instruction type can not be determined.
36	Format error in ADD, SUBTRACT, MULTIPLY, DIVIDE or MOVE instruction: These instructions require either a TO, FROM, TIMES, or INTO separating the source and destination operands.
37	Format error in a GOTO or PERFORM instruction: The sentence label must be immediately preceded by an exclamation point.

NUMBER	DEFINITION
38	Format error in a GOTO or PERFORM instruction: A sentence label within a GOTO or PERFORM instruction can not contain more than eight characters (excluding exclamation point). The first character of the label must be alpha (A–Z) and must immediately follow the exclamation point.
39	RELEASE or BYPASS instruction format error. Valid formats of RELEASE and BYPASS instructions are: RELEASE. BYPASS. RELEASE AT END... BYPASS AT END... For the second format, commas may be used in place of, or with blanks separating, words.
40	WHEN instruction operand type can not be determined.
42	WHEN (NOT) PGM instruction format error: The PGM number specified in a WHEN (NOT) PGM instruction must be a one digit number (0–9).
43	WHEN (NOT) PGM instruction format error. Valid formats for this instruction are: WHEN PGM n, WHEN NOT PGM n. Spaces must appear between all words in this instruction.
44	WHEN (NOT) PGM instruction format error: The PGM in this instruction must immediately be followed by either a comma and/or a space.
45	IF instruction format error. Valid formats for this instruction are: 1) IF <S1> R <D1> 2) IF <S1> R <D1> OR IF <S2> R <D2>... 3) IF <S1> R <D1> OR R <D2>... where <S> = source operand <D> = destination operand R = relationship In type 2, an OR IF implies a new source and destination operand. For type 3, OR without IF implies reuse of the previously specified source operand. In this case the relationship must be repeated, even if it is the same as the last specified relationship.

NUMBER	DEFINITION
	Valid relationships are: >, <, =, ≠. Spaces are required between words and operands but are not required between operands and relationships. Commas may be used whenever they make sense.
46	Sentence format error: The last statement of any sentence must be followed by a period. The first non-conditional statement of a sentence must be the last statement of a sentence.
47	Subroutine format error: ENTER can not be followed by a period.
48	Invalid termination of source program: The last sentence of a program must be terminated with a period.
49	Subroutine format error: A subroutine was closed (EXIT) but had never been opened (missing ENTER).
50	Variable name format error: The first character of a variable name must be alpha (A–Z).
51	Program error: SORT instruction legal only within a SORT routine.
52	Program error: OUTPUT instruction illegal within RECORD END and SORT routines.
53	Variable table overflow: Only 3 variables are allowed within a RECORD END routine; or 99 variables for other routines.
54	Size of PAUSE statement exceeded: A PAUSE statement cannot exceed 40 characters (excluding quotes).
55	AT END not valid in Record End Edit Routine: AT END statement illegal within Record End Edit Routine.

NUMBER	DEFINITION
71	Multiply defined variable: A variable can only be defined once within a program.
72	Undefined variable: A variable must be defined within a DECLARE instruction prior to other references to it.
73	Subroutine format error: The last subroutine in the program was not closed (missing EXIT).
74	Subroutine format error: A new subroutine was opened (ENTER) before the preceding subroutine was closed (missing EXIT).
75	Subroutine format error: A PERFORM instruction is illegal within a subroutine. The nesting of subroutines is not allowed.
76	Subroutine format error: A subroutine may be entered only by using a PERFORM instruction. The only valid sentences which can immediately precede a subroutine are those containing one of these instructions: STOP, EXIT, GOTO or RELEASE. The AT END option of the RELEASE instruction is not valid.
77	Variable table overflow: Only 99 different variables can be specified within a program.
78	Multiply defined label: Two or more sentences can not be preceded by the same label. Each sentence label must be unique.

NUMBER	DEFINITION
79	Multiply defined label reference: The GOTO/PERFORM instructions reference a multiply defined label.
80	Undefined label reference: The GOTO/PERFORM instructions reference an undefined label.
81	Invalid subroutine call: A label specified in a PERFORM instruction must be one which precedes a valid subroutine entrance, e.g., !LABEL, ENTER . . .
82	Illegal branch into a subroutine: A GOTO instruction can not be performed on a subroutine from outside that subroutine.
83	Illegal branch out of a subroutine: A GOTO instruction can not be performed outside a subroutine.

ENTREX, INC.
Technical Publications Remarks Form*

TITLE: **PROGRAMMER'S REFERENCE GUIDE**

ORDER No. **S-15**

DATED: **MARCH 74**

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME _____

COMPANY _____

TITLE _____

_____

DATE _____

*Your comments will be promptly investigated by appropriate technical personnel, and action will be taken as required.

ENTREX, INC. 168 Middlesex Turnpike, Burlington, Mass. 01803