FACTOR Manual

DECEMBER, 1970

FAIRCHILD
SYSTEMS TECHNOLOGY

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLES

FIGURES

SECTION I

INTRODUCTION

FACTOR is a procedural programming language which consists of control statements for the Sentry 400 Test System.

FACTOR provides two basic types of statements: (1) arithmetic and logical control statements, such as those which normally comprise procedural languages; (2) test control statements which set up and execute functional/ parameter tests on electronic elements or devices.

FACTOR is an acronym derived from 'Fairchild Algorithmic Compiler-Tester ORiented'.

The sections in this manual discuss such subjects as the codes and symbols used by FACTOR; the test statements which are part of FACTOR, its operating procedures, expressions, control statements, input/output statements, and subprograms.

SECTION II

ELEMENTS OF FACTOR

2.1 INTRODUCTION

This section provides the syntax information and describes the input devices necessary for producing programs which are legal input to the compiler.

2.2 CHARACTER SET

Letters	A, B, C, ...Z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special Characters	! ' # % & ' ( ) * + , - . / : ; <=>?@[ ]↑SPACE←$

Appendix A shows the internal code for the character set. Valid characters are the printing ASCII characters which are produced directly by the teletype, except for "["and"]" which are upper case K and M respectively. In the case of the 029 card punch some substitutions are required. These are shown in Appendix A. Appendix A also shows where the 029 character set differs from the teletype character set. The rules for using the character set in FACTOR programs are discussed in the text of this manual.

2.3 RECORD FORMAT

A FACTOR source program is prepared on punched cards, or can be entered via the teletype keyboard. A FACTOR program is a list of statements or commands, which define particular actions. When the program is executed, these commands are carried out, sequentially, in the order written, unless a specific FACTOR command is used which alters the command sequence.

A statement is the basic record of a FACTOR program. "END" or "ELSE" or a semicolon, ";", are all legal terminators for statements. An example of a statement is:

```
LABEL:  A = B+1;
```

This statement consists of a label (optional) and a command. The format of statements is essentially free form (spaces are ignored by the compiler).

2.3.1 Cards

When the user prepares his card deck of source statements there are several options:

Up to 72 columns of the card may be used for one or more statements, providing a semicolon delimits each statement. Also, a FACTOR statement may be started on one card and be carried over to the next, provided that individual words of tester instructions are not divided between two cards.

Cards can be sequenced either alphabetically or numerically, or both. The sequence characters are placed in columns 73 through 80. This is why only columns 1 through 72 may be used for statements, since otherwise a portion of the statement would be interpreted as a sequence symbol.

The normal form of sequence numbers is a fixed alpha identifier in (say) columns 73-75, followed by numeric digits in the remaining columns through column 80. These (five) digits will ascend in sequence through the program by a convenient increment, one, ten, etc. Sequence symbols are checked for progression. Gaps (e.g. sequencing by tens) in the sequence may be left, so that program corrections and additions may be made without changing every sequence number in the deck. If a single deck contains more than one alpha identifier, these identifiers must be chosen so that the TRASCII ascending collating sequence is maintained, otherwise a sequence error will be produced.

When an error is detected, the compiler will always type the full current record and a message "SEQUENCE ERROR".

Examples:

```
          Legal Sequences                      Illegal Sequences

NUMERIC   1    3    3   10    20   99999999    0   1   3   2   5   4
COMBINED  1LB  2LB  3LB                        1A  0A  3A  1A
ALPHA     A    C    D   E     E    E   F       A   B   P   E   C
```

2.3.2 Disc

Disc files may be used as a source program input to FACTOR. They must be type "string" and are loaded onto the disc as discussed in the DOPSY manual.

2.3.3 Paper Tape

Source programs may be prepared on punched paper tape. The format rules are the same as previously described for cards. Sequence numbers are not normally used when preparing the tape via a teletype, since paper tape is restricted to 72 readable characters per line. The end of the input line is signalled by "carriage return"-"line feed".

Two editing characters may be punched on the paper tape for correcting punch errors. A character back space is obtained by typing the teletype "control" and "B" keys simultaneously. The number of times this key combination is typed corresponds to the number of previously entered characters which are to be ignored. A line delete is obtained by typing "control" and "L" simultaneously. This character deletes the current line only. It is necessary to type "carriage return" and "line feed" after the last END statement in the program.

The file is terminated by a // followed by "carriage return" and "line feed".

At least 20 characters of blank tape should be provided for leader/trailer.

2.3.4 Teletype

Source programs may be entered directly via the teletype keyboard using the format and rules described in Section 2.3.3, describing paper tape input. In this mode of operation the statements in each input record are compiled as they are entered. The two editing characters discussed above for paper tape apply here also, viz: "control"-"B" and "control"-"L".

2.4 ERROR MESSAGES

Most of the error messages issued by FACTOR are self-explanatory. They are listed below with some comment for clarification. The error messages are accompanied by an up-arrow "↑", where appropriate, to indicate the position in the statement text where the error was detected.

Text	Description
"VARIABLE NAME" DEFINED PREVIOUSLY	Notifies user of duplicate lable definition within the same block.
"VARIABLE NAME" IMPROPERLY USED	A label has been referenced which has not been defined, or a mistake in logic has been detected, i.e., using a variable as a scaler when it has been defined as an array.
SEQUENCE ERROR	An error has been found in the sequence numbers punched in columns 73-80 of the source card deck.
SS FULL	The compiler's capacity for the storage of symbols has been exceeded. (Reduce the number of symbols used.)
NW FULL	There are too many noise words.
WORK FULL	The program has a compound tail too large to be processed as one statement.
DISC OVERFLOW	There is not enough space on the disc for further object program to be built up in working storage.
TOO MANY NESTED BLOCKS	The allowable maximum number of nested blocks has been exceeded. (See Appendix I)
SYSTEM 2 ERROR	"Never-happen" error-return (e.g. PUTW E-O-F).

PROGRAM TOO BIG	The program exceeds FACTOR capabilities (see Appendix I).
MISSING))	A left or right paren has been left out.
EXPRESSION SYNTAX	An expression has been written incorrectly.
MISSING]]	A left or right bracket has been left out.
MISSING NAME	An identifier should have been specified in this syntactical position.
MISSING NUMBER	A number should have been specified.
STATEMENT SYNTAX	A statement has been incorrectly written.
STATEMENT TERMINATOR	A syntactical delimiter, or a semicolon is missing.
NUMBER SYNTAX	A number has been specified incorrectly.
INVALID TERMINATOR	An expected terminator or delimiter is incorrectly specified or missing.
I/O SPECIAL ERROR	The I/O control word has indicated an error.
END OF FILE INPUT	The input file has been exhausted without finding an END statement.
TOO MANY VARIABLES	The allowable maximum number of variables per block has been exceeded (see Appendix I).

SECTION III

FACTOR OPERATING PROCEDURES

3.1 INTRODUCTION

A program written in FACTOR must be compiled before it can be executed on the Sentry-400. The Compiler converts the FACTOR English-like statements into a program file of object codes. The object code is then input to the TOPSY system, which interprets and executes the program.

3.2 PROGRAM INITIATION

To initiate the FACTOR compiler, using the DOPSY monitor, the user must type:

```
// COMPILE [TTK/TTR/CR/'file name'][TTP/LP] [LIST/OBJ/LISTOBJ]
```

This command format is general and includes all possible options.

3.2.1 Input

The first group of enclosed options in the above command indicates that the compiler input may be specified from the teletype keyboard (TTK), from paper tape via the teletype paper tape reader (TTR), from cards via the card reader (CR), or from a file on the disc ('file name') which was previously created as outlined in the DOPSY Manual.

Not more than one of these options may be specified. The user may, however, elect not to specify any option, in which case the compiler will expect its input from the current principal input device (PID) assigned to DOPSY.

3.2.2 Output

The second and third groups of options, in the general format command, are used to specify the output desired.

The listing may be specified for either the teletype or the line printer. Again, no more than one option may be entered with the command. If no entry is made, the output, if any, will go to the principal output device which is currently assigned to DOPSY. If LIST is selected, then source statements only are listed; if LISTOBJ is selected, then both source statements and their resulting object code will be listed. If either OBJ or LISTOBJ is selected, then the compiler places its translated program in working storage on the disc. Note the distinction between 'LISTOBJ' and 'LIST OBJ'.

A typical initiation command might be:

```
// COMPILE CR LP LISTOBJ
```

followed by a carriage return, if entered at the teletype. This command would read its source program from the card reader and produce both an object program in working storage and also a listing of the source statements and the interleaved object code on the line printer.

If a specified input and/or output device is not available, an error halt will be taken to 100B with the I/O device ordinal in the accumulator.

When a program error is detected, one of two procedures is taken: (1) if the error is recoverable, i.e., if the compiler can continue, FACTOR will continue to compile and notify the user of further errors, (2) if the error is not recoverable, the DOPSY monitor will be called and an asterisk will be typed to notify the user that DOPSY is in control again.

Note that when parens or brackets are missing the up-arrow error position indicator may be placed within the error message text. Two parentheses or brackets are used in the text of the error message, since a single symbol might be obliterated by the up-arrow, making the message illegible.

3.3 INTERPRETER INTERFACING

FACTOR produces an object program which must be saved by the user if it is to be executed. Once a compilation has been completed return is made to the DOPSY system monitor with the compiled program in working storage. The user then has the option of correcting any errors in the source program and redoing the compilation, or if the program compiled error free, he may save the object program by creating a type "DATA" file on the disc:

```
// CREATE DATA 'file name'
```

(See the DOPSY System Manual for a description of the CREATE command). The user program may now be executed under the control of the TOPSY interpreter. TOPSY is called by typing // TOPSY, followed by a carriage return. The operation of the program from this point, using the /. LOAD command, etc., is described in the Sentry 400 TOPSY User's Manual.

SECTION IV

EXPRESSIONS

4.1 INTRODUCTION

An expression is a grouping of one or more numbers, variables, and functions combined with arithmetic or Boolean operators and parentheses so as to represent a quantity or an operation. Note that a single number or variable is considered an expression by this definition.

4.2 NUMBERS

FACTOR accepts numbers in any of the three forms discussed below, viz: integers, decimal fractionals or exponentials. In all cases, numbers are converted to a floating point internal representation for manipulation in the computer. The range of allowable decimal numbers is:

$$2.7105 * 10^{-20} \leq |N| \leq 9.2228 * 10^{18}$$

where $|N|$ means the "magnitude of N".

4.2.1 Integers

An integer is defined as a whole number, including zero. It will be interpreted as octal if immediately followed by a B. It will be interpreted as decimal if it is not followed by a B. It may be either signed (preceded by a + or -) or unsigned. If unsigned, it will be interpreted as positive.

The limits for decimal and octal integers are:

Form	Limits
decimal integer	$-8388607 \leq n \leq 8388607$
octal integer	$40000001B \leq n \leq 37777777B$ (in two's compliment form)

The following integers

1) are acceptable:	2) are unacceptable:
0	4,000,000 (Commas not allowed)
4000000	10000000000000000000 (Too large)
+2361	125 B (Imbedded space between number and octal specifier)
-5	
6B	

4.2.2 Decimal Fractionals

A decimal fractional is any decimal number with a fractional part preceded by a period. These numbers cannot have a B (octal) notation. An attempt to use octal notation in combination with a decimal fractional results in an error message. Decimal fractionals may be signed or unsigned.

The following fractional numbers

```
1)  are acceptable:     2)  are unacceptable:

    4.0                     4.     (A number cannot end with a period)
    0.0                     1.234B (A fractional number cannot be specified
     .671                           octal)
    +.734650
  -42.0
    0.734650
```

4.2.3 Exponentials

Exponentials are either decimal integers or decimal fractionals, followed immediately by an E and a decimal integer. They also may be signed or unsigned.

The following exponential numbers

```
1)  are acceptable:     2)  are unacceptable:

    0.1E2                   0.1E2+    (The sign must come between E and its
   +1.23E-5                            integer)
    7E-3                    1E        (The exponent must have an E number)
   -1.0E+5                  .234 E5   (Imbedded spaces are illegal)
   -5E+2                    2BE2      (Octal numbers may not be exponentially
                                       specified)
```

4.3 VARIABLES

In FACTOR, a variable denotes any quantity which is referred to be a name rather than by an explicit value. A variable may take on many values, one at a time, rather than being restricted to only one value. The values which are assigned to a variable may be any of the forms as discussed above. Also, variables may be either scaler or Boolean. A variable identifier (see below) may reference either a single variable or a set of variables considered as an array.

4.3.1 Variable Identifiers

Variable identifiers are names which are given to variables by the programmer. There are no restrictions imposed, except that the identifier must begin with a letter and contain only letters and digits. Identifiers can be of any length; note, however, that FACTOR retains only the first 8 characters. Consequently, the user must insure, when using long identifiers in the same block, that the first 8 characters are unique, or else an error message will be produced. Furthermore, reserved words must not be used as identifiers. The following is a list of reserved words in FACTOR:

AND	INSERT
BEGIN	~~LEG~~ LEQ
BLOCK	LT
BY	MEASURE
CALL	NEG
CPMU	NEQ
DCL	NOISE
DISABLE	NOT
DO	ON
ELSE	OR
ENABLE	PAUSE
END	READ
EOR	REM
EQ	SET
FOR	SOCKET
FORCE	SUBR
FUNCT	THEN
GE	THRU
GOTO	XCON
GT	XPMU
IF	

It is good practice, since the usage of identifiers is totally determined by the programmer, to use names (identifiers) which represent the meaning or use of a variable. PI, for instance, could be the name given to a variable whose value is set equal to π. COUNTER might be the name given a variable which used as a general purpose counter, and so on. (This does not imply, however, that FACTOR attaches any significance to these names. They are purely artificial devices which aid the user's memory and make a program more intelligible.

If variables are referenced without being declared with a statement they are assumed to be scalar; if they are not assigned an initial value they are given a value of zero. Refer to Section VI for variable declaration information.

The following are acceptable variable identifiers:

```
A
CHISQUARE
ALARGEIDENTIFIER
A1B2C3D4
PHOENIX
```

The following are not acceptable variable identifiers:

123	(Identifiers may not start with a digit)
A BC	(Special characters, including blanks, are not allowed)
END	(Reserved words are illegal)

This is a general definition of identifiers which holds for the other several types of identifiers which are used in FACTOR.

4.3.2 Scalar Values

The FACTOR variable in its simplest form is scalar. Scalars are defined as being nonarrayed, nonBoolean quantities. (Note that, as defined below, an array element may be a scalar value and/or a Boolean value). In addition, scalars may take on any legal numbered values. To use the scalar value which is currently assigned to a variable, the user writes the variable's identifier in his statements or expressions.

4.3.3 Boolean Values

Boolean values are quantities which when evaluated have a value of either one (true) or zero (false). Expressions involving Boolean operators (paragraph 4.6) can only take on a true or false value, i.e., a one, or a zero; thus expressions are not evaluated for any other absolute value. Whenever the user references the variable identifier the current Boolean value will be returned.

4.3.4 Array Values

An array is an ordered series of values which are grouped together positionally with respect to some variable identifier (usually the first array element identifier). The elements of the array are restricted to either signed or unsigned numbers (i.e. alpha values are illegal). FACTOR arrays are restricted to one dimension.

To obtain an array value, the user must follow the array identifier with an expression which is enclosed in brackets. The value of the expression is the subscript which tells FACTOR which element of the array is wanted. If the subscript is zero, i.e., A[0] for instance, FACTOR returns the array size. Any other value of the subscript will refer to the appropriate element in the array of values. For example, A[2] would reference the second element in the array A. Note: if the value of the expression is negative or greater than the array size, a terminal error will result during execution.

4.4 FUNCTIONS

Functions are parametered calls and are used to obtain a value through a standardized set of operations. The FUNCT statement is described in Section VIII.

4.5 ARITHMETIC EXPRESSION EVALUATION

Arithmetic expressions are evaluated left-to-right according to the following rules.

1. Parenthesized expressions are evaluated first. If parenthesized expressions are nested, the innermost expression is evaluated, then the next innermost until the entire expression has been evaluated.

2. Within parenthesis and/or whenever parenthesis do not govern the order or evaluation, the hierarchy of operations in order of precedence is

   a) Negation (NEG)
   b) Multiplication or division (*,/),
   c) Addition or subtraction (+,-).

   Note: Exponentiation is not allowed.

   Example:
   The expression

   A*(Z-((Y-R)/T)) + VAL

   is evaluated in the following sequence.

   $Y\text{-}R \rightarrow e_1$
   $e_1/T \rightarrow e_2$
   $Z\text{-}e_2 \rightarrow e_3$
   $e_3{*}A \rightarrow e_4$
   $e_4{+}VAL \rightarrow e_5$

4.6 "BOOLEAN" OPERATORS (LOGICAL AND RELATIONAL)

Symbol	Operation
OR	inclusive or
EOR	exclusive or
AND	logical and
LT	less than
EQ	equal to
LEQ	less than or equal
GT	greater than
NEQ	not equal
GE	greater than or equal
NOT	not

A description of the above operators follows.

4.6.1 OR

The following truth table explains the result P OR Q, where P, Q are two expressions:

P	Q	P OR Q
1	1	1
1	0	1
0	1	1
0	0	0

4.6.2 Exclusive OR (EOR)

P EOR Q is similarily explained in the following truth table:

P	Q	P EOR Q
1	1	0
1	0	1
0	1	1
0	0	0

4.6.3 AND

The following is the truth table for P AND Q:

P	Q	P AND Q
1	1	1
1	0	0
0	1	0
0	0	0

4.6.4 NOT

The NOT operator forms the ones complement of the expression upon which it operates. For example, if A is true, (1), then NOT A is false, (0).

Prior to forming the ones complement the expression is evaluated and then fixed as an integer. The integer form is limited to 24 bits, therefore conversion underflow or overflow from floating point to fixed format must be considered by the programmer.

4.6.5 Relational Operators

The remaining group of Boolean operators are relational operators. They deal with the comparison of two or more arithmetic values. The result of the comparison is either a Boolean true or false.

For example, consider the following comparison:

```
        A LT B
```

If the values of the two variables is 16 and 25, respectively, the comparison is effectively:

```
        16 LT 25,
```

which is a true statement.

Consider now:

```
        B LT A, i.e. (25 LT 16)
```

This is of course a false condition. Similar examples could be given for EQ, LEQ, GT, NEQ and GE.

4.7 BOOLEAN EXPRESSIONS

A Boolean expression defines whether a true or a false condition exists.

The order of operations for Boolean expressions depends upon the precedence values of the operators, unless parentheses are used which override the precedence order (a recommended procedure, especially in the case of complicated expressions). The precedence order of Boolean operands is:

a) relational operators (LT, LEQ, EQ, GE, GT, NEQ),
b) NOT,
c) AND,
d) OR and EOR.

The following are examples of Boolean expressions:

1. A (where A is either true or false)
2. A OR B EOR C
3. A GE B OR A LT C

In example two the expression is evaluated from left to right: A is ORed with B and then the result is EORed with C. In example three the expression A GE B is evaluated for a true or false condition: the expression A LT C is evaluated: the results of these two operations are ORed together.

4.8 MIXED EXPRESSIONS

FACTOR allows free mixing of arithmetic and Boolean expressions, without adhering to pure Boolean values. It is the responsibility of the programmer to ensure that values in mixed expressions can be fixed to valid integers (see Section 4.6.4) when they are involved in a Boolean expression. Further, arithmetic operators take precedence over Boolean operators in mixed expressions.

SECTION V

BLOCK-COMMAND AND PROGRAM CONCEPTS

5.1 INTRODUCTION

Blocks are groups of program statements between the delineators, BLOCK (or BEGIN) and END. Local variable storage and local labels do not exist outside of the block which contains them; in other words, local variables or labels cannot be referenced (accessed) outside of their parent block. A block is in effect then an independent compilation, since a program can consist of several completely independent blocks.

5.2 LABEL

A label is an identifier similar to a variable identifier except that it is always followed by a colon and it refers to a statement. The complete definition and all restrictions which apply to labels can be found in paragraph 4.3.1, which describes variable identifiers. The following is an example of a label identifying an assignment statement.

```
        TOTAL:  A = 1 + 2 + 3 + C/D;
```

5.3 BLOCK CONCEPT

A block consists of a beginning and closing statement. In addition, a block can be either independent or dependent. The following information will describe the block and its function.

5.3.1 Establishing the Block

A block is established in two ways.

1. It may be opened directly by writing the command BLOCK and closed by the command END; cf the command BEGIN (Section 7.5) which is very similar in concept.

2. A block will also open following the FUNCT and SUBR commands. These commands are discussed in Section VIII.

The initial BLOCK declaration need not be specified because FACTOR will assume a BLOCK 0. For clarity, the user may spell out the initial block in his listing. It must have an END statement, however.

5.3.2 Nesting Blocks

Blocks do not need to be completely independent. One of the easiest methods of introducing block dependence is by "nesting" one block within another. This results in the execution of the inner block being dependent on the execution of the outer block. Nesting can occur up to eight levels on the Sentry 400 implementation. Nesting is illustrated in the following example:

```
BLOCK
     BLOCK
     END;
     BLOCK
          BLOCK
          END;
     END;
END
```

The inner block of a nested set is considered part of the enclosing blocks. Another form of dependence is that of global variables. A global quantity is one that is accessible to a block, but is not necessarily contained in (i.e., is not local to) that block. Variables and labels can be either local or global. This is illustrated in the following example:

```
BLOCK
L: DCL A, B/10/;
     BLOCK
     DCL A,C;
     END;
END
```

Each block in the above example contains the local variable A. The A in the inner block cannot be accessed from the outer block and vice versa. The variable B in the outer block is accessible from either block, but the variable C can be accessed only from the inner block. In this example, then, B is a global variable, but C and the two variables A are all local.

Note, that if there had been a label L in the inner block, any reference to it within the inner block would have used that L rather than the one in the outer block. Any nested set of blocks establishes a block context; i.e., a relationship of local and global variables. From the example, it can be seen that a reference to a variable or label is associated with the occurrence of that identifier or label in the same block, if it is present. If it is not, then the next outer block is examined, etc.,

It should be noted that it is possible to make variables global from within a nested block in FACTOR by simply never declaring the variable as local. When the nested block is closed, the variable, and any residual value is relocated to the next outer block, where it may now be considered as global to any further nesting. When this outer block is closed, if it was nested, the variable will again be relocated to the next outer block and so forth until block 0 is closed.

The fact that the declaration of a variable within a block makes it local has important implications for the FACTOR user. After leaving a block, i.e., closing it with an END command, the values of all variables declared within the block, and thus made local, are lost. Upon re-opening the block, the values fo these variables are initailized to 0.

SECTION VI

VARIABLE DECLARATION AND VALUE ASSIGNMENT

6.1 INTRODUCTION

As described in paragraph 4.3, variables may be used in expressions without giving them initial values or by declaring them. If they are not declared, they will be assumed to be scalar. If they are not given an initial value, they will be automatically given an initial value of zero. Variables may be declared and assigned values at any point in the program.

A variable may also be used as an array reference, but then it must be declared. Thus, a declaration (DCL) statement must always be executed before any references are made to the declared arrays. If this rule is violated, TOPSY will indicate the programming error with a terminal error at run time (see Appendix F).

If the DCL statement is executed more than once in a currently open block, all but the first execution will be ignored; however, a value assignment will always occur at every execution. This point must be emphasized. It means that the evaluation of an array size will occur once only: at the first DCL for that array. But, values will be assigned for every DCL specified.

6.2 DCL

The DCL command is used to reserve storage for variables, assign initial values, and to make a variable local to the block in which it is declared.

Two types of variables may be declared: scalars and arrays.

6.2.1 Scalar Declaration

The general formats of the scalar declaration are as follows:

```
DCL     V1;
DCL     V1, V2,.....VN;
DCL     V1/VALUE 1/,V2/VALUE 2/,...VN/VALUE N/;
DCL     V1, V2/VALUE 2/,V3...VN;
```

V1...VN stand for variables number 1 through N. VALUE 1...VALUE N stand for single signed or unsigned numbers which declare the value of a variable. When declared without value, the variable is set equal to 0. Multiple declaration and assignment can be made with one statement. As shown in the last two examples, each variable of a multiple declaration can be optionally assigned an initial value.

6.2.2 Array Declaration

The general formats of the array declaration are as follows:

```
DCL     V1[A1SIZE];
DCL     V1[A1SIZE],...VN[ANSIZE];
DCL     V1[A1SIZE]/AE1,...AEM/,...,VN[ANSIZE]/AE1,...AEM/;
DCL     V1[A1SIZE],V2[A2SIZE]/AE1...AEM/,...,VN[ANSIZE];
```

Formats are similar to those for scalar declaration; however, the array identifier, V1, requires an argument to specify the number of elements, i.e., the size of the array. This quantity is enclosed in square brackets. The size is specified by an expression which allows it to be variable or fixed. The evaluation of array size and allocation of storage is performed by TOPSY at run time. The array size of necessity is automatically truncated to the nearest integer.

The elements of an array may be optionally assigned initial values when declared. The assignment is specified by the terms AE1 through AEM as shown above; M is the size of the array. The initial values of the elements are restricted to signed or unsigned numbers. (Alpha is illegal). If the size and number of initial value assignments do not agree, the missing (trailing) elements are set to zero. If too many elements are specified, a run time error will occur.

Note: The distinction must be drawn between the value in square brackets used in a DCL statement, where it represents the array size, and the value in square brackets used in a non-DCL statement, where it specifies the array element desired.

Examples:
```
DCL    ARR[10]; REM ARRAY SIZE = 10 ELEMENTS;
For J = 1 THRU 10
ARR[J] = 2*J; REM COMPUTE ARRAY ELEMENT VALUES;
FIFTH = ARR [5]; REM ASSIGN VARIABLE FIFTH
        THE VALUE OF THE FIFTH ARRAY ELEMENT;
```

6.3 ASSIGNMENT STATEMENT

The assignment (or "replacement") statement is the most fundamental of all FACTOR statements. It takes the general form:

```
VARIABLE = EXPRESSION
```

This command results in the replacement of the value of the variable on the left by the value of the expression on the right. (In general, it is not an equation, since the variable on the left may form part of the expression on the right).

Thus the statement:

```
A = A+B;
```

means: take the value of the variable A, add the value of the variable B to this value and replace the value of the variable A with the result.

SECTION VII

CONTROL STATEMENTS

7.1 INTRODUCTION

Control statements are used to direct the flow of the program by a transfer of control to different part of the program. Such a transfer may be imperative (e.g., GOTO) or conditional (e.g., IF).

The control statements to be discussed in this section are PAUSE; GOTO; IF; BEGIN; and FOR.

7.2 PAUSE

The PAUSE statement is used to stop the execution of further statements until START is depressed. The format for this statement is:

```
PAUSE expression;
```

Prior to halting, the value of the expression is evaluated and printed on the Primary Output Device.

This statement can be used to provide programmed halt when debugging new FACTOR programs.

NOTE: Refer to the TOPSY User's Manual for instructions for using the monitor mode PAUSE command.

7.3 GOTO

A program is essentially a series of statements which in general are executed sequentially and thereby accomplish a particular task. The computer thus operates one step at a time. However, it is essential to be able to enter or leave the sequence of instructions at any desired point.

This is the function of the GOTO statement. When executed, a GOTO statement usually changes the program flow from the statement immediately following it to the one specified within the GOTO statement.

The general form of the GOTO statement is

```
GOTO    LABEL;
```

where LABEL as defined in Section 5.2 specifies the statement to be executed next.

7.4 IF

The GOTO statement provides a method for altering the sequence of statement executions unconditionally. It is also essential to be able to change the sequence of execution based on what happens as the program executes, i.e., a conditional change of execution. This is the principal use of the IF statement.

The simplest form of the IF statement is:

```
IF relation THEN statement-1;
Statement-2;
```

Upon execution of the IF statement, if the relation is true, statement-1 is executed followed by statement-2 (unless statement-1 carries control away from statement-2). If the relation is false, statement-2 is executed instead.

For instance

```
IF A EQ 3 THEN GOTO LABEL;
```

will, if it is true that A is equal to 3, cause the sequence of execution to change to the point in the program having a statement labeled LABEL. If A is not equal to 3 the next sequential statement, after the IF statement, will be executed.

The true-false nature of the above relation gives a clue to the second general form of the IF relational clause. It is:

```
IF Boolean-expression THEN statement;
```

where "Boolean-expression" is any legal expression as defined in paragraph 4.6.

Suppose, for example, that we wish to continue doing something until the value of A and B, two variables we are manipulating, both become less than some terminal value, 0. We could make this decision and monitor the values of A and B with one IF statement as follows:

```
IF A LT 0 AND B LT 0 THEN GOTO DONE;
```

The process we wish to continue doing immediately follows the IF statement.

In all cases of IF statement usage, the following THEN can introduce any type of FACTOR statement.

7.4.1 <u>The Conditional ELSE</u>

The simple IF statement is one which causes a statement to execute if a relation or Boolean expression is true and skips statement execution if the relation or expression is false. A complete conditional statement does more. It specifies a second statement to be carried out if, and only if, the relation or expression is false.

The general forms are:

```
          IF relation THEN S1 ELSE S2;
          IF Boolean expression THEN S1 ELSE S2;
```

where S1 and S2 are any two statements. When the result of the 'IF' operation is true, S1 will execute and S2 will be ignored. When the result is false, S1 will be skipped over and S2 executed. S2 may be any statement, including another IF statement. This nesting of conditionals can go to any depth.

```
          Example:
                  IF relation THEN S1 ELSE IF relation THEN S2 ELSE S3;
```

7.5 BEGIN

FACTOR allows the grouping of a series of statements within the bracket commands BEGIN and END;. The command, END;, must immediately follow the last command executed. Note the ; which is an integral part of the END; bracket. One of the purposes of this command is to allow a compound statement to follow the THEN of the IF command. For example:

```
          IF relation THEN
                    BEGIN
                       statement;
                       statement;
                       statement;
                    END;
```

The above example is an example of a compound statement, and is an acceptable method of writing the IF statement. The statements between BEGIN and END; are legal and, as far as the IF statement is concerned, are considered to be one statement.

Note: In many respects BEGIN and BLOCK are equivalent, except that declarations after a BEGIN are transferred to the BLOCK head containing the BEGIN.

7.6 FOR

One of the techniques most widely used in programming is that of the program loop. This is the repetition of some program statement or statements over and over with different parameters. The FOR statement is the looping mechanism within FACTOR.

The general format of the FOR statement is:

```
          FOR variable = expression THRU expression DO statement;
```

where variable, expression and statement may be in any legal form defined in this manual.

Several statements may be included in the DO loop portion of the FOR statement by specifying a compound statement with BEGIN and END; .

An example of a typical loop is one designed to solve the following problem. Suppose it is desired to set the elements of an array to zero. This can be achieved with the IF statement sequence of statements, in the following

```
        I = 1
NXT:    A[I]=0;
        I = I+1
        IF I LE A [0] THEN GOTO NXT;
```

but it is better accomplished with the statement:

```
FOR I = 1 THRU A[0]DO A[I]=0;
```

The simple FOR statement provides an index value which has three important features:

```
     (1)     an initial value,
     (2)     an (assumed) increment of +1,
and  (3)     a limit
```

In the above example, I takes on the values 1, 2, 3,..., A[0], where A[0] is the last value corresponding to the size of the array.

The implementation of the FOR causes the address of the index, the increment and the limit to be evaluated each time the loop is executed. Therefore, caution must be exercised within the loop when chaing values that might affect this evaluation.

The loop will be executed the number of times specified by the initial value, limit, and increment. (This may be zero.) Also, there is no restriction on transfers of control into or out of the loop. When the loop has finished its specified number of executions, control will pass to the next sequentially executable statement, unless this sequence is interrupted by a statement in the DO loop.

In the above discussion an automatic increment of +1 from the initial value to the final value was assumed. There is second form of the FOR statement; this allows the user to specify, using BY , some value which will be used as the increment. This adds considerably more power to the FOR statement.

can increment be an exp. ?

It should be pointed out that because the values may be all positive, all negative, or mixed positive and negative, the user should consider the range of possible values he expects. It makes sense to go from a negative number to a more negative number in negative increments or from a positive number to a negative number by negative increments. Going from positive to more positive or negative to positive, the increment must be positive. Going from -2 to +6 in increments of -2, as from +8 to +2 in increments of +2 is not logical and will be flagged as errors.

Caution must be exercised when using fractional values for the index, since it is possible to introduce a step error. For example, a statement such as:

```
          FOR I = 0 THRU 1000 BY 0.1 DO I = I + 1;
```

may operate the DO statement more than 10,000 times because of a rounding error in the floating point conversion of 0.1.

SECTION VIII

SUBPROGRAMS AND FUNCTIONS

8.1 INTRODUCTION

This section discusses the function and operation of subprograms, the calling of subroutines and how to make a function out of a subprogram.

8.2 SUBPROGRAMS

Programs frequently have groups of statements which can be used several times with different parameters. Of course, the required statements could be duplicated wherever they are needed in the program, but to do so is wasteful of user time and machine storage and is error prone. Therefore, it is desirable to be able to write statements in a way such that they may be executed from any point in the program with a different set of parameters each time they are executed. The subroutine statement allows this.

8.2.1 SUBR

The general formats of the subroutine declaration are as follows:

Format One:

```
          SUBR    Identifier;
                  statement 1;
                  statement 2;
                     •
                     •
                     •
                  statement n;
          END;
```

Format Two:

```
          SUBR    Identifier (VI1, VI2,...,VIN)
                  statement 1;
                     •
                     •
                     •
                  statement n;
          END;
```

The identifier after the SUBR command is used to reference the subroutine from the main program. The statements within the subroutine will not be executed until the subroutine is called from the main program by the SUBR identifier. Any number of statements are allowed within the subroutine.

The END; statement is necessary because the SUBR command effectively opens a new block. When it is completed, it must be closed. The END; indicates the last statement in the subroutine.

Format Two indicates another important feature of the SUBR statement. The terms VI1 through VIN represent variable identifiers 1 through N. They are enclosed in parentheses and indicate to FACTOR that whenever a reference is made to this subroutine, the reference will specify actual values which are to be substituted at specific places within the subroutine body. These identifiers are called formal parameters. There is a one for one correspondence between the position of the formal parameters and the position of the parameters or values used in the call. For reference and further explanation, see the next section on the CALL statement. The manner in which values transferred to the subroutine are used in the subroutine's statements is illustrated in the following example:

```
SUBR TOTAL (VI1, VI2, VI3);
     VI1 = VI2 + VI3;
END;
```

When the above subroutine is referenced:

```
CALL TOTAL (A1, A2, A3);
```

the values passed to it, obtained from the actual parameters A1, A2 and A3, will positionally replace VI1, VI2, VI3 and they will be used in the arithmetic expression and assignment. The value of the variable represented by VI2 will be added to that represented by VI3 and the total will be assigned to the variable represented by the formal parameter VI1.

As an example of a subroutine with no formal parameters specified, we will use a similar statement as follows:

```
SUBR TOTAL2;
     A = B+C;
END;
```

When TOTAL2 is called, the current values of the variables B and C are added and the total is assigned to the variable A. In this case A, B and C are not formal parameters. They are working variables with current values in the outer blocks to the SUBR statement block.

Because the subroutine forms a new block, it must be remembered that any variables which are declared in the subroutine will be local.

8.3 CALL

A subroutine is executed by using a CALL statement, which can be placed at any point in the program where the programmer can legally place a statement. The general formats are as follows:

```
          CALL SI;
          CALL SI (expression 1, expression 2,..., expression N);
```

SI is the identifier of the subroutine block to be activated. The values, changed by the subroutine statements and by any other task executed, will be accomplished as if the subroutine's body of statements has been placed at the point of the CALL statement. Then, the next sequential statement, following the CALL, will be executed.

The expressions are evaluated at the time of the execution of the call and, therefore, will remove many constraints which are ordinarily placed on the CALL values. As the subroutine statements are executed, the value of expression 1 will be used wherever formal parameter 1 was used. The same will hold true for other formal parameters and expressions. The only restriction is that when a formal parameter receives a result, the corresponding actual parameter should not be an expression but a single variable identifier.

8.4 FUNCT

The subroutine call, when encountered in the program execution, brings the subroutine statements into action to accomplish whatever processing is specified (ordinarily assigning new values to outer block variables). Control then usually passes to the next sequential statement.

When only one variable is assigned a new value, as a result of executing a subprogram, the call can be simplified by making the subprogram a function. When FUNCT is used, simply writing the identifier of the function will cause its statements to execute. However, the identifier now represents a value that may be used wherever a variable is legal. Thus, it is as though the function call represents a variable of the same name.

The general form of the function statement is:

```
          FUNCT    identifier (VI1,...,VIN);
              statement 1;
                      .
                      .
                      .
              statement n;
          END;
```

The format, except for the FUNCT command portion, is exactly like the SUBR statement. It also defines a new block and it will be called by the identifier, following the FUNCT. The difference is in the way the function is activated and also that it always returns a value for the function identifier. If no assignment of a value to the function identifier is made within the statements following FUNCT and before END; is encountered, a value of zero will be returned. Assignments of values to a function, which are external to the function declaration statement are illegal, i.e., the function name must not be used on the left hand side of an assignment statement.

The function identifier is not local to its function block and therefore must not be declared within the function statements.

As with a subroutine call, the FUNCT statement, with its compound tail of statements, is not executed until the function identifier is used in an expression.

8.4.1 Function Call

As stated, using the function identifier as a variable identifier in an expression will cause the function statement block to activate and to return a value for the identifier. Note the following example:

```
FUNCT TOTAL3 (VI1,VI2,VI3);
      TOTAL3 = VI1+VI2-VI3;
END;
```

When it is desired to reference a value by executing the above function, its identifier is used as follows:

```
NEWTOTAL = TOTAL3(A,2,B+C)+(A-B);
```

The function TOTAL3 will be evaluated using the current value of A for formal parameter VI1, 2 for formal parameter VI2 and the current value of B+C for parameter VI3. This overall value will then be added to the value calculated for the subexpression, A-B, using current values for A and B. Finally, the end value arrived at will be assigned to the variable, NEWTOTAL.

The same symbol must not be used for formal and call parameters.

It is worth considering a further example to illustrate the range of versatility of a function. Assume there are two sums to evaluate:

Sum 1:

$$A = \sum_{B=1}^{10} B2$$

Sum 2:

$$C = \sum_{D=1}^{E+H} E+F[G]/D$$

Because expressions are allowed in the function call, one general function statement could be set up to handle both sums as shown below:

```
          FUNCT   TOTAL (W,Z,Y,X,R);
              W = 0;
              FOR Z = Y THRU X DO W = W + R;
              END;
```

To evaluate Sum 1, the call is written as:

```
          TOTAL (A,B,1,10,B*B)
```

Sum 2 could also be evaluated by the same function with the call:

```
          TOTAL (C,D,1,E+H,E+F[G]/D)
```

The effective result of the two function calls is shown below:

```
          Sum 1:
              A = 0;
              FOR B = 1 THRU 10 DO A = A+ B*B;
              END;

          Sum 2:
              C = 0;
              FOR D = 1 THRU E+H DO C = C+E+F[G]/D;
              END;
```

The parameters of a function call can also include other function calls. In fact, a function may even call itself recursively. For instance, a factorial could be calculated as follows:

```
          FUNCT   FACTORIA (N) ;
              IF N = 1 THEN FACTORIA = 1
                  ELSE FACTORIA = N * FACTORIA (N-1);
          END;
```

This is an example of recursion, the use of a function within the same function. The user should remember that the number of recursive calls is determined (and limited) at run time by the size of core.

FAIRCHILD
SYSTEMS TECHNOLOGY
974 E. Arques Ave., Sunnyvale, Calif. 94086

INTERNAL CORRESPONDENCE

October 22, 1970

To: A. T. Smith cc. K. Rinaldo

From: H. R. Gillette

Subject: INSERT Statement in FACTOR

Development of the subject change has been completed and on direction of Ken Rinaldo I am transmitting complete material to you for incorporation into the standard system. The following materials are attachments to this memo and constitute the results of the change effort.

1. Two (2) boxes of cards, consisting of the following:

   a. A copy of the control program (*COMPI) for generation and loading of this part of the compiler. Red cards are DOPSY JCL statements, blue cards are ASM statements. The ORG card was deleted from REV 2 of the deck, location now being determined on the CREATE COREIMAGE card for DOPSY.

   b. A copy of the updated compiler (*COMP1) for generation and loading of the main FACTOR compiler. Red cards are DOPSY JCL statements, blue cards are REV 2 compiler statements unchanged and white cards are actual revisions. An inventory of the changes is included in the documentation.

   c. Finally is a test deck which should be run as a separate job. Two compiles will result - the generated object code should be identical.

2. A draft of documentation for the change, including as an appendix, an update to the FACTOR language manual.

3. Three listings, including:

   a. The output of the included test case compiles.

   b. The output of an assembly of '*COMPI'.

   c. The output of an assembly of '*COMP1'.

[illegible]
H. R. Gillette

HG/mp

Attachment

A. IDENTIFICATION

a. Title: FACTOR Modification

b. FST ID Code:

c. Machine: FST-1

d. Category: Change to FACTOR for CPR #J017

e. Language: Assembly

f. Programmer: H. R. Gillette

g. Documentor: H. R. Gillette

B. PURPOSE

Introduces the INSERT statement into the FACTOR language. INSERT allows a set of one or more FACTOR source statements to be catalogued (a subroutine, for example) on the disk and to be included in any FACTOR program simply by naming the file at the appropriate point. For example, the effect of the factor statement:

```
INSERT DECK
```

would cause the content of the file whose name is 'DECK' to be compiled as part of the source string, following which scan of the original source string would resume.

C. RESTRICTIONS

a. Machine Configuration: 8K minimum core.

b. Programming System Configuration: DOPSY system.

c. Other Subroutines Required: Does not apply.

d. External Symbols: Does not apply.

e. Register Usage: Changes introduced use A, E, X2, X3, X4, X5, and X7.

f. Relocatability: Part of FACTOR, does not apply.

g. Required Console Switch Setting: None.

h. Other Restrictions: The named file must have the following characteristics:

i) Be catalogued, available to DOPSY under the current JOB. The file name must be a valid FACTOR identifier.

ii) Not be of type CI or OB.

iii) Be of type "STRING" or be of type "DATA" with record length $\leq$ 20.

iv) Be a string of statements in the FACTOR language. Sequence checking is reset, so that character positions 73-80 of the first and last statement need have no sequence relationship to the requesting source deck.

v) The control overlay must be loaded at 14000B or above. Finally, INSERT becomes a reserved word of the FACTOR language.

D. USAGE

a. Entry Point: Does not apply.

b. Calling Sequence: A new "basic statement" is added to the FACTOR language called the "INSERT statement".

i) Syntax:

```
<INSERT statement> ::=INSERT <identifier>
```

ii) Semantics:

The first 6 (or fewer) characters of the identifier are used as a file name. If the file is catalogued and available, the environment of the current input stream is saved, the requested file is opened and becomes the current input stream. On detection of the end-of-file for the current input, the saved input environment is restored and input scan continues at the first character after the identifier. Inserted text is free to use the INSERT statement. Recursion in this manner is limited only by the amount of core used by tables.

c. Arguments and Parameters: Does not apply.

d. Error Conditions:

i) If the entity following INSERT is not a valid identifier, the following diagnostic message is issued:

"FILE NAME ERROR"

ii) If the requested file cannot be opened successfully, the following diagnostic message is issued:

"FILE NAME ERROR"

iii) If the requested file is of type CI or OB, the following diagnostic message is issued:

"FILE TYPE ERROR"

iv) Sequence checking is performed within an inserted file.

v) Normal syntax checking takes place within the inserted text.

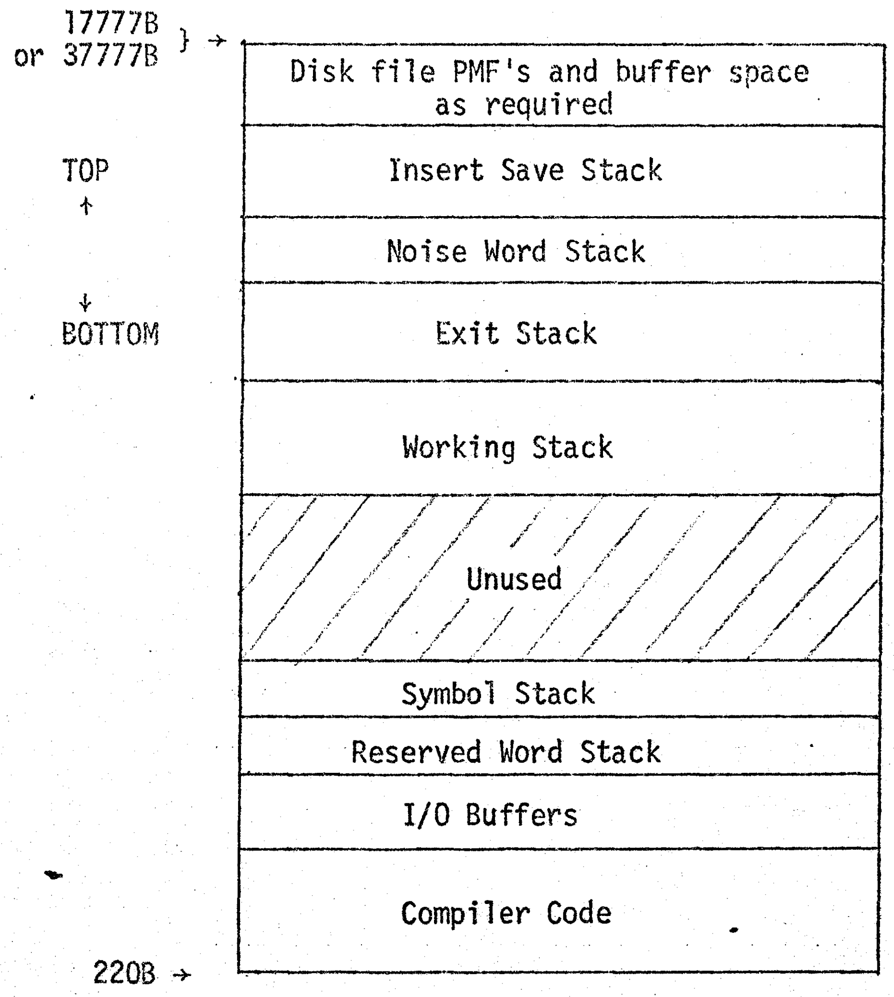
vi) If the inserted text is not a sequence of complete statements, an "END OF FILE INPUT" error occurs which aborts the compilation.

e. Space Requirements: The change adds 247 locations to the compiler, plus table space only if an INSERT occurs.

f. Input Format: FACTOR source statements.

E. <u>METHOD</u>

A new statement processor (INSERT) was added to the compiler. In addition, the table management routine (TADD) was rewritten to provide management for the dynamic allocation of space needed to save an input environment and for processing an insert file. With these changes, memory usage is as follows:



The routine TADD is called whenever an entry is added to any one of the dynamic stacks. If space is required, the work stack, exit stack, etc., are moved toward the symbol stack making room at the top of the area requiring space. TADD was rewritten to generalize the distance moved which is now table driven (by the MVSIZE vector). The move was previously a fixed 4 locations. In the current implementation (as also before), storage usage grows as required - there is no scheme for garbage collection and compression. If available storage is exceeded, the compiler simply commits suicide.

Management of the stacks is table driven, four values being defined for each region: TOP, BOTTOM, SIZE, and MVSIZE. The TOP of one region is not allowed to overlap the bottom of the next higher region. The management table is arranged into 4 vectors so that each area has an "index" name. The following two dimensional array defines the current implementation.

Index	Name	TOP	BOTTOM	SIZE	MVSIZE
0	NT	0	0	4	4
1	RW (Reserved Words)	RWTOP	END	3	3
2	SS (Symbol Stack)	SSTOP	SSBOT	4	4
3	WS (Working Stack)	WSTOP	SSBOT	1	4
4	ES (Exit Stack)	ESTOP	ESBOT	1	4
5	NW (Noise Word)	NWTOP	NWBOT	2	4
6	IN (Insert)	INTOP	INBOT	83	83
7	DE (Dummy Entry)	MAXMEM	MAXMEM	-	-

SIZE - defines the size of a single entry

MVSIZE - defines the amount of additional space to be allocated at the TOP when additional space is required

BOTTOM - low memory base

TOP - current top of area and high word allocation to most current entry

If TOP = BOTTOM, the area is empty.

Each insert entry requires 83 words which are allocated as follows:

```
INTOP      )
...        )   PMF for new file
INTOP-8    )

INTOP-9    )
...        )   Buffer, one sector in length,
INTOP-56   )   for new input file

INTOP-57   )               )
...        )   (INBUF)     )
INTOP-76   )               )
INTOP-77       (LSEQ2)     )
INTOP-78       (LSEQ1)     )   previous input file
INTOP-79       (FSTNO)     )   dynamic environment
INTOP-80       (SCNPTR)    )
INTOP-81       (INDCB)     )
INTOP-82       (INFID)     )
```

The call chain leading to INSERT is as follows:

$$\begin{tikzcd}
\text{CTAIL} \arrow[dr] & & & & \\
& \text{STSCN} \arrow[dr] & & & \\
& & \text{INSERT} \arrow[dr] \arrow[ul, bend right, "\text{on EOF for input file}"'] & & \\
& & & \text{STSCN} \arrow[dr] \arrow[ul, bend right, "\text{on EOS}"'] & \\
& & & & \text{etc.}
\end{tikzcd}$$

Both CTAIL and STSCN are recursively entered so that the statements in the inserted file may be any valid sequence of statement. The syntactic element following:

INSERT <file name>

must be "ELSE", "END", or ";" otherwise an error will result from the

STSCN processor. The end-of-file which results in the exit out of INSERT is detected in the NXTCHR routine which branches to INEOF to initiate restoration of the previous input file environment.

Finally, a utility subroutine MOVE was added to facilitate storage moves. MOVE is called by both INSERT and TADD. The calling sequence is as follows:

```
Set (x3) = FWA of From Area
    (x2) = LWA of From Area
  (PARX) = FWA of To Area
Then       BSM       MOVE
```

F. INVENTORY OF CHANGES

a. Add INSTAK definition to miscellaneous equates. INSTAK defines the "index" to the stack management vectors.

b. Add INTOP, INBOT, INSIZE definitions to storage management tables.

c. Add MVSIZE vector to storage management tables.

d. Change end-of-file return in NXTCHR to branch to INEOF.

e. Replace TADD with generalized version.

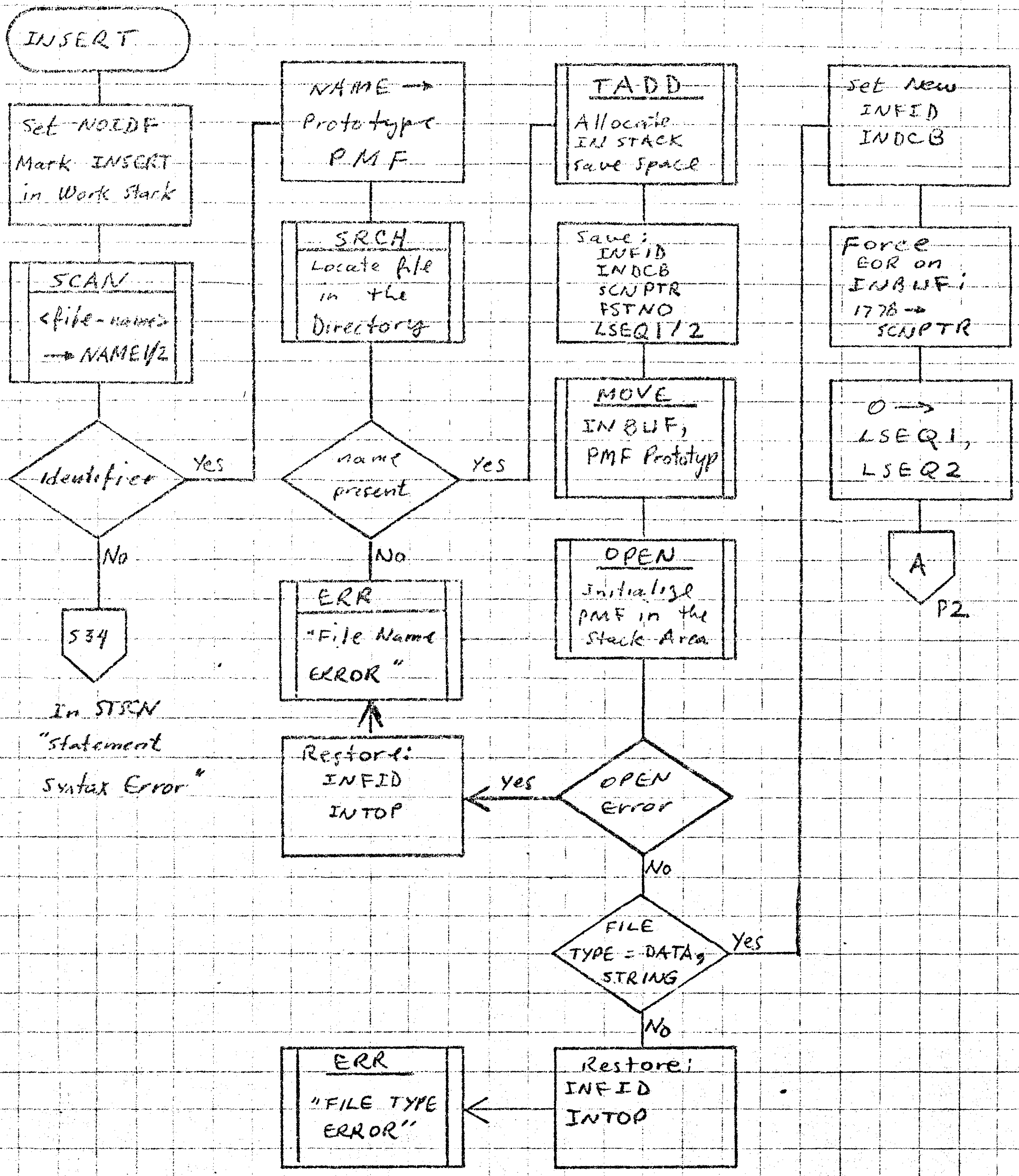
f. Add INSERT processor

g. Add MOVE subroutine.

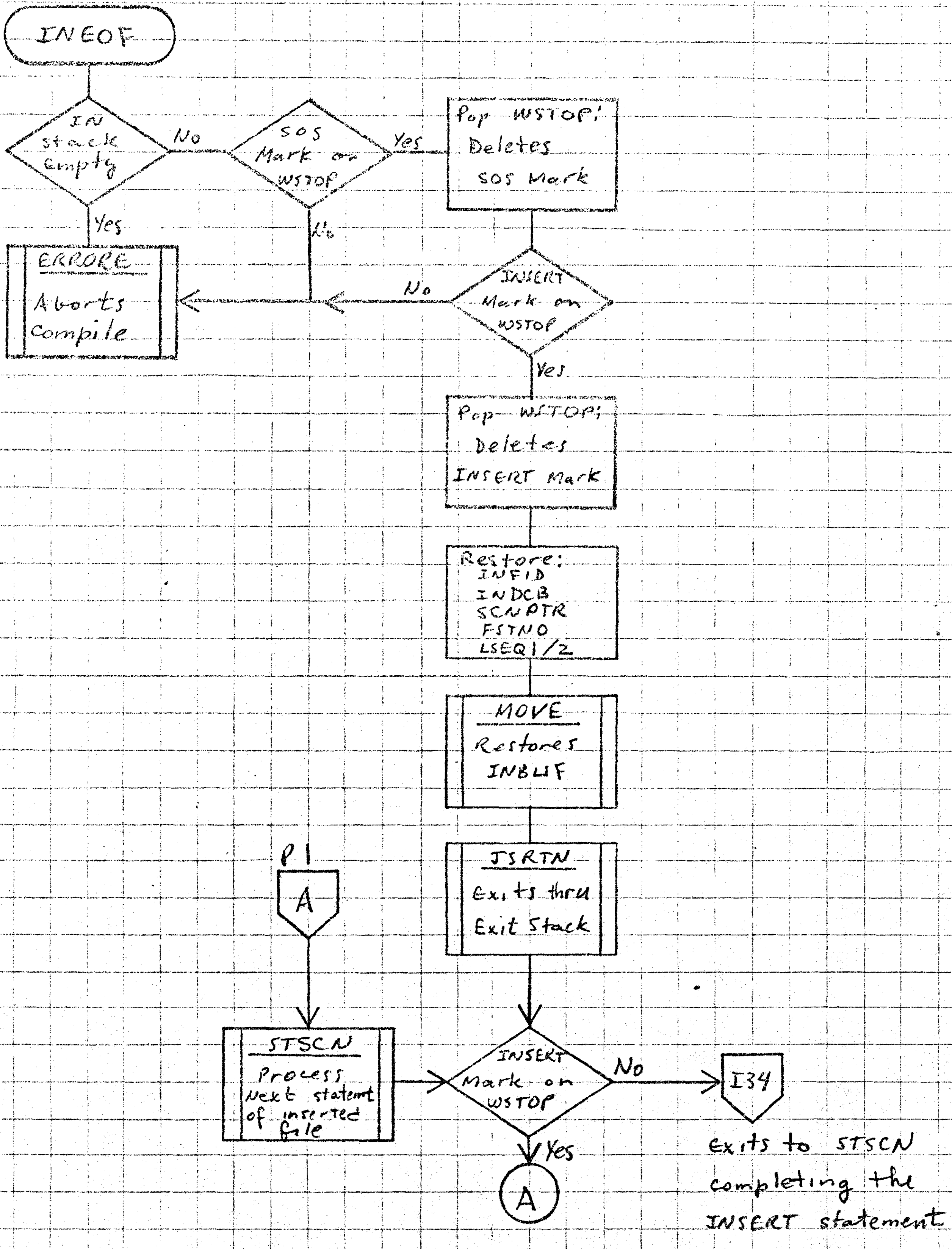
h. Add initialization for INTOP, INBOT to initialization process.

i. Add INSERT to Reserved Word Stack. (INSERT will not cause the statement number to be incremented.)

G. INSERT Process Flow Chart P1


INSERT
Set NOIDF Mark INSERT in Work Stack
SCAN <file-name> → NAME 1/2
Identifier
Yes
No
534
In STSN "Statement Syntax Error"
NAME → Prototype PMF
SRCH Locate file in the Directory
name present
Yes
No
ERR "File Name ERROR"
Restore: INFID INTOP
TADD Allocate IN STACK save space
Save: INFID INDCB SCNPTR FSTNO LSEQ 1/2
MOVE INBUF, PMF Prototyp
OPEN Initialize PMF in the Stack Area
OPEN Error
yes
No
FILE TYPE = DATA, STRING
Yes
No
Restore: INFID INTOP
ERR "FILE TYPE ERROR"
Set New INFID INDCB
Force EOR on INBUF: 177B → SCNPTR
0 → LSEQ1, LSEQ2
A
P2

INSERT Process Flow Chart P2



INEOF
IN stack empty
No
Yes
ERRORE
Aborts Compile
SOS Mark on WSTOP
Yes
No
Pop WSTOP; Deletes SOS Mark
INSERT Mark on WSTOP
No
Yes
Pop WSTOP; Deletes INSERT Mark
Restore: INFID INDCB SCNPTR FSTNO LSEQ1/2
MOVE
Restores INBUF
JSRTN
Exits thru Exit Stack
P1
A
STSCN
Process Next statement of inserted file
INSERT Mark on WSTOP
No
Yes
I34
A
Exits to STSCN completing the INSERT statement

APPENDIX

UPDATE FOR FACTOR PROGRAMMING GUIDE

Ref: FACTOR Programming Guide - July 1969

5.1 INSERT Statement

It is frequently desirable to be able to write a set of code which can be used in many programs. One way to accomplish this is to produce a card copy for each programmer. The drawbacks of this approach are apparent if the logistics of a change to this code is considered. A better way is to create one copy of the common code on direct access storage and then provide a simple means for each programmer to include a copy at the desired point of his text. The INSERT statement is defined to provide for this facility.

Assume, for example, that two files, named TEST2 and TEST3 respectively, exist on disk. Assume further that TEST2 contains the text:

```
        REM TEST2 11/29/70 FILE FOR INSERT TEST;
             A = B+C*D;
```

and that TEST3 contains the text:

```
        REM TEST # 11/29/70 FILE FOR INSERT TEST;
             A = B+1;
             B = C/D;
```

then the following program will illustrate valid usage of the INSERT statement.

```
REM TEST OF INSERT STATEMENT 10/19/70;

DCL   A,B,C,D;

INSERT TEST2;

If A EQ 0

     THEN BEGIN

        INSERT TEST2;
        END

     ELSE BEGIN

        INSERT TEST3;
        END;

END
```

This otherwise meaningless program is offered as a test case. The generated code will be identical to that produced by the following program:

```
REM   TEST PROTOTYPE;

DCL   A,B,C,D;

A = B+C*D;

IF A EQ 0

     THEN BEGIN

        A = B+C*D;
        END

     ELSE BEGIN

        A = B+1;

        B + C/D;
        END;

END
```

This is, in effect, the same program, except that the INSERTed code of the previous program is imbedded in the actual program. Each time it is used, the single statement

```
INSERT  <file name>;
```

is replaced by the set of statements contained in the named file.

There is no restriction on the code that can be INSERTed; the INSERTed code could be a set of DCL statements, a BLOCK, a subroutine, etc.

SECTION IX

INPUT/OUTPUT STATEMENTS

9.1 INTRODUCTION

In Section III, the input of source statements to the compiler and the output of compiled data statements from the compiler was discussed in detail. The compilation of a source program is referred to as compile time. The execution of the compiled program, using TOPSY, is called run-time. This section deals with READ and WRITE statements that make it possible to input or output run-time data.

The statements READ and WRITE described in this section use the following syntax notation.

- Outer parentheses '(',')' are used to enclose items that are optional; at least one item must be selected; items are separated by slashes; '/'.
- A 0 indicates that none of the elements of the set need be chosen. When none are selected, the system assigns the current DOPSY Primary input or output device.
- A file name identifier is shown in lower case letters and is enclosed by double quotation marks.

9.2 READ

The general format of the input statement is:

```
READ((CR)/(EIR)/(TTK)/(TTR)/(MTR)"name"/0)V1, V2,...Vn;
```

The items enclosed by parentheses are peripheral devices defined as follows:

Card Reader	(CR)
Teletype Keyboard	(TTK)
Teletype Reader	(TTR)
Magnetic Tape	(MTR)
External Interface Reg	(EIR) - See Section 11.8

When magnetic tape, MTR, is specified the statement must include a file name which is enclosed by double quotation marks. The file name syntax is defined in the same manner as Identifiers (see Section 4.3.1).

Magnetic tape file "names" are used to uniquely identify data segments on the tape. These names are assigned with the WRITE statement (Appendix E).

The terms V1 through Vn may be any legal variable identifier, including arrays. As the input numerical data is read from the peripheral, it is assigned to the specified variable(s).

The input data is restricted to numbers as defined in Section 4.2.

When values of an array are to be read, they must be separated by at least one space (for TTK, TTR, CR). More than one card can be used to enter these values. All numbers following the last array element number on a card are ignored. Note that the first value for each new variable identifier must start on a new card.

When the input peripheral is the magnetic tape unit, the tape is searched forward until the file "name" is located. The numerical data from this file is read and assigned to the variables VI,...etc. as specified by the READ statement. For magnetic tape, the variables must be arrays which have no less than 7 elements. The maximum array size is limited by the amount of core memory available when the array is declared. It is recommended that arrays be no larger than ≈ 512 elements. Appendix E gives a detailed description of the magnetic tape operation and responses to the READ (MTR) and WRITE (MTW) statements.

9.3 WRITE

The formats for output statements are:

```
WRITE((EIR)/(TTP)/(LP))  (Vi, 'Si', Vj, 'Sj',..,Vn);
WRITE (MTW) "name" V1, V2, V3, V4;
```

where: Vi...Vj...Vn...V1...V4 are legal variable identifiers including arrays which may occur in any sequence. Si, Sj...are strings of alphanumeric characters.

The items enclosed by parentheses are peripheral devices defined as follows:

Teletype Printer/Punch...	(TTP)
Line Printer...	(LP)
Magnetic Tape...	(MTW)
External Interface Register...	(EIR) - See Section 11.8

When magnetic tape (MTW) is specified the statement must include a file segment name which is enclosed by double quotation marks. When writing to magnetic tape, the variables Vi must be arrays which have no less than seven (7) elements and are recommended to be no larger than five hundred and twelve (512) elements as described in Section 9.2. Appendix E gives a detailed description of the magnetic tape operation and responses to the WRITE (MTW) statement.

When the teletype or line printer is specified as the output device there may be one or more strings of alphanumeric characters and one or more variables in a single WRITE statement. All strings must be enclosed by single quotes and must not contain semicolons (;). Multiple variables are separated by commas as are intermixed combinations of strings and variables.

Variables are output in one of two forms. If the numeric value of the variable is an integer whose magnitude is less than one thousand (1000), it will be printed in the following form:

    S999

where S is the sign of the number (+ or -) and the '9' terms are decimal digits. Leading zeros print as spaces.

Non-integers and integers whose magnitude exceed nine hundred and ninety-nine (999), print in the following format:

    S9.999EP99

where S again is the sign of the value and the '9.999' represent the decimal digits of the mantissa, the '99' represent the decimal digits of the exponent of the value and P is the sign (+ or -) of the exponent. The character 'E' prints as shown. For example, 8.979×10^{-6} prints as '8.979E-06'.

Numeric values as described above occupy a field of twelve (12) characters and are left justified within this field.

The maximum number of variables printed per line is five (5) with the first character field left justified.

When a variable is an array, its current values are printed five (5) per line beginning with array element one (1) left justified on the line.

More than five (5) variables can be specified per WRITE statement with the result that five values per line will be printed on all lines including the last, unless there are fewer than five values to fill the last line.

Strings of characters are printed as they appear in the enclosed parentheses. The characters may be any of those in the character set (Section 2.2) excluding single quotation marks and semicolons(;). Leading spaces are printed according to the number of spaces following the single quote of a string. The total number of printed characters will be an integral multiple of four (4). (The restriction is automatically imposed at run-time with the addition of no more than three (3) spaces following the character preceding the trailing quote of a string.)

A single string of seventy-two (72) characters can be printed on a single line when the teletype is the output device. When the line printer is the output device, a string of eighty (80) characters can be printed on one line. Single strings which extend beyond column seventy-two (72) of a punched card can be continued beginning with column one (1) of the next card, etc. When the single string exceeds the character counts described above, the excess characters are printed on the following line. The teletype will ignore characters between column seventy-three (73) and eighty (80).

When variables and strings are intermixed in a single statement, the following output rule holds:

> If the count of characters printed on the current line exceeds fifty-six (56), then the first character of the next entity (either a variable or string) will be printed left justified beginning on the next line. Otherwise, it will be printed beginning on the current line and character position. Overflow to the next line will occur whenever the character count of a string exceeds the number of available characters on the line.

Example:

(FACTOR Code):

```
WRITE 'DATALOG';
WRITE ' ';
WRITE 'TEST#=',N,'    VALUE=',VALUE;
WRITE 'NODE=',PINN, '     EXPECTED VALUE=',EV;
```

(Output Data):

```
DATALOG

TEST#=  +  6           VALUE= +1.200E-06
NODE=   -  0           EXPECTED VALUE= +  2
```

In this example, the variables are N, VALUE, PINN, and EV. At the time the WRITE statements are executed, these variables had the following numeric values; 6, 1.2×10^{-6}, 0, 2, respectively.

SECTION X

NOTATIONAL STATEMENTS

10.1 INTRODUCTION

Notational statements are used to enhance the readability and clarity of programs. The notational statements discussed in this section are NOISE and REM.

10.2 NOISE

The NOISE statement is used to define words which will make the FACTOR statements read like English sentences. Its formats are:

```
NOISE WORD1;
NOISE WORD1, WORD2,...WORDN;
```

The command NOISE is followed by at least one space and the noise word or words which are separated by commas. After defining the noise words they are ignored by the FACTOR compiler. This provides a means for adding clarity to FACTOR statements. An example is shown below:

```
NOISE   VOLTS, AMPS, WATTS;
     V = 10.0 VOLTS;
     I = 1.0E-3 AMPS;
     P = V*I WATTS:
```

Noise words are restricted to the format of identifiers, however, the reserved words listed in Section 4.3.1, as well as user-declared identifiers are disallowed as noise words.

10.3 REM

The REM (remark) statement provides a means for adding commentary to a program listing. It is not executable and it can occur anywhere in the context of a program provided its format rules are followed. The general format is:

```
REM text;
```

where text is anything but END, ELSE or ';'.

For example:

```
          REM   THIS SECTION CALCULATES AVERAGE VALUE;
```

As noted above, anything except END, ELSE or ';' is legal in the REM statement. All other elements of the character set will be positionally listed as located in the REM statement at compile time.

SECTION XI

TEST STATEMENT FORMATS

11.1 INTRODUCTION

Within the following sections the statements which are available for testing digital elements are described. Each statement description consists of a general form which gives the following information; time delay, definition, and in most cases, an example.

The nine major statement types discussed in this section are: Set statements, Enable statements, Disable statements, Force statements, Connect statements, Disconnect statements, Measure statements, On statements and Socket ID statements. The general form of the statement definition uses the syntax notation defined below.

11.1.1 Brackets

Brackets, "[ ]" are used to enclose items that are optional. The choice of options are separated by slashes.

11.1.2 Parentheses

Parentheses "( )" are used to enclose items that are optional. The choice of options are separated by slashes. The underlined option is assumed if none is specified.

11.1.3 Semicolons

Semicolons ";" terminate the statement.

11.1.4 Voltage and Current Ranges

The programmable Voltage and Current modules of the Sentry-400 have from one (1) to four (4) ranges of operation. The four ranges are specified by reserved words as follows:

" RNG0
" RNG1
" RNG2
" RNG3

The full scale value of voltage or current corresponding to each range depends on the statement and module involved. Appendix C summarizes range numbers and their correllation to full scale value, resolution, statement and module.

11.1.5 Time Delay Dependent Instructions

The execution of a time delay dependent instruction is automatically delayed so as to allow the preceding test conditions to stabilize. For example a DC measurement is not executed until all previously programmed power and voltage supplies have stabilized. The value of the time delay is determined by the type of tester instruction previously executed. Time delay dependent instructions wait for the status 'Tester Busy' to be null.

The instructions which are not time delay dependent are executed without waiting for stabilization, even thoush they may initiate a Tester Busy status. This allows a series of programmed responses to essentially stabilize during the same time period, rather than in a sequence. The statements which are time delay dependent are listed in Appendix D.

11.2 SET-UP STATEMENTS

This section describes statements which are typically used to initialize the tester prior to performing actual tests.

11.2.1 Set Delay

General Form:

```
SET DELAY expression (,DC);
```

Time Delay Generated:

0

Description:

A. Without DC Modifier

The value of the expression is loaded into the tester time delay register TD. This presets the delay time for subsequent ex3cutions of functional tests of the form SET F. That is, the expression plus 0.7 microseconds will be the value of the time delay between input stimulus execution and output comparator strobing (a SET F instruction). The 0.7 sec offset is equal to the typical driver response time to 62% of its final value from initial value. The delay used should allow for device under test response time plus 0.4 sec comparator response time. The Set Delay statement has a resolution of 0.35 microseconds and a mazimum value of 5.734 milliseconds.

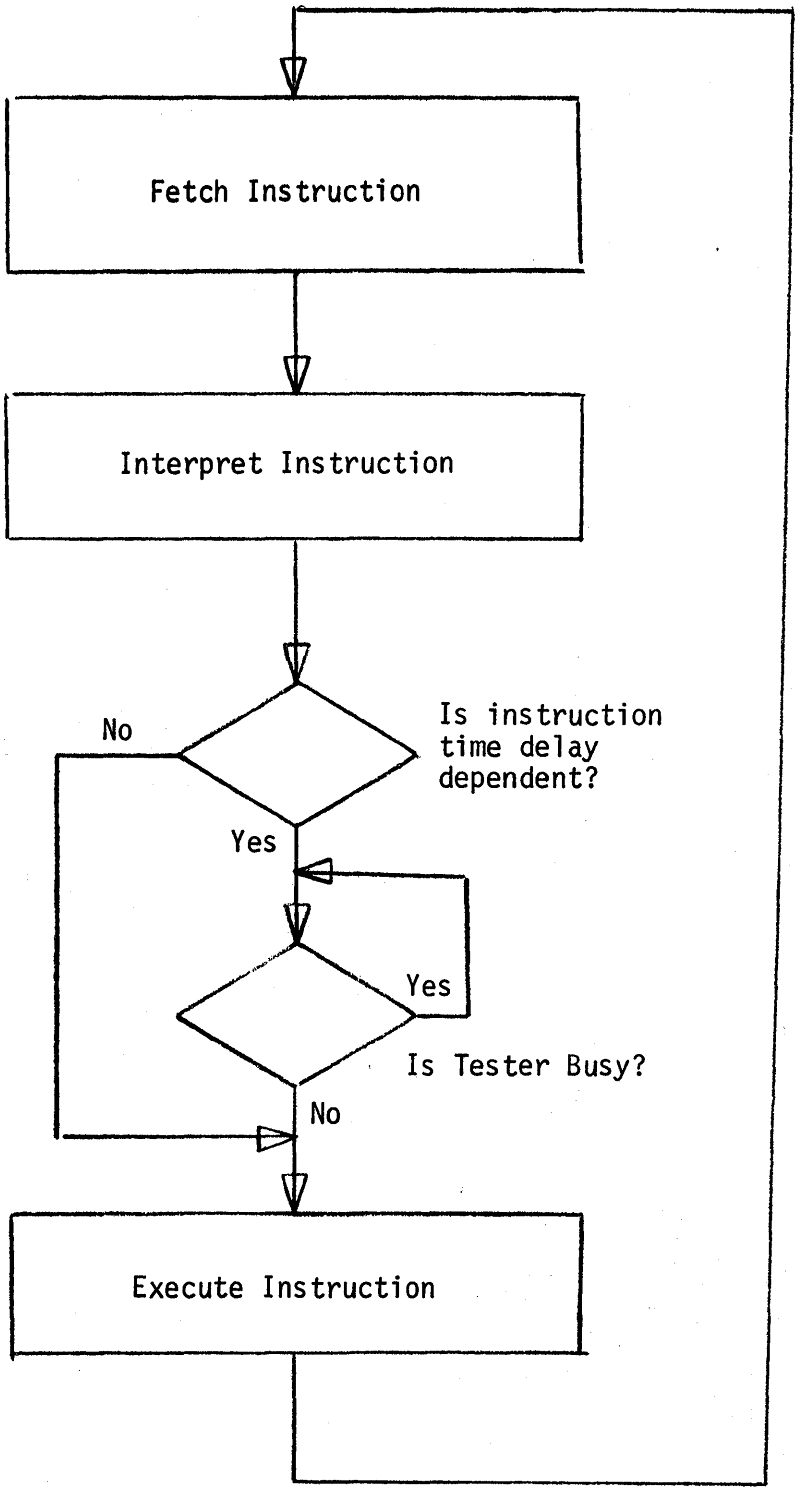


Figure 11.2.1-1

BASIC INTERPRETER FLOWCHART

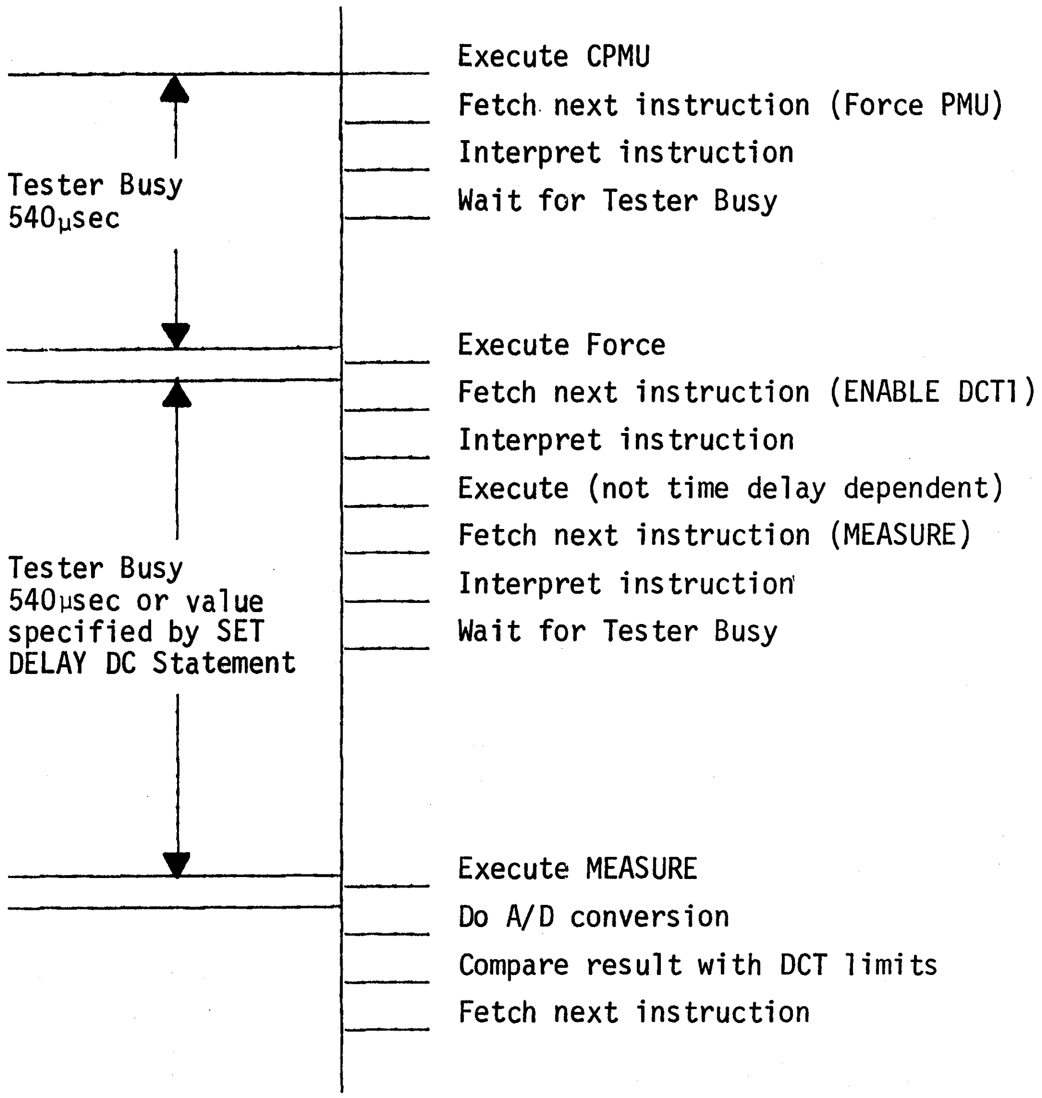

Execute CPMU
Fetch next instruction (Force PMU)
Interpret instruction
Wait for Tester Busy
Tester Busy 540μsec
Execute Force
Fetch next instruction (ENABLE DCT1)
Interpret instruction
Execute (not time delay dependent)
Fetch next instruction (MEASURE)
Interpret instruction
Wait for Tester Busy
Tester Busy 540μsec or value specified by SET DELAY DC Statement
Execute MEASURE
Do A/D conversion
Compare result with DCT limits
Fetch next instruction

Figure 11.2.1-2

Typical DC measurement sequence showing overlap of 'non time delay dependent' instructions with tester busy.

Example: Set the functional test delay for 350 microseconds.

```
SET DELAY 350E-6;
```

B. With DC Modifier

The value of the required time delay is scaled by FACTOR and loaded in the tester time delay register. The resolution is 0.35 milliseconds with a maximum value of 5.734 milliseconds. When a FORCE PMU, FORCE (VOLTAGE/CURRENT) or FORCE DELAY instruction is executed then the 'tester busy' status will remain "on" for the amount of time defined by the Set Delay, DC statement. Even if this instruction is not used a time delay generated by a fixed delay generator of 1.75 msec, if a range is changed, or 0.54 msec if no range is changed, will occur.

Example:

```
SET DELAY 0.005, DC;
```

11.2.2 <u>Set Clamp</u>

General Form:

```
SET CLAMP (POS/NEG/SYM) number
```

Time Delay Generated:

0

Description:

POS means no negative voltages below -0.7 volts will be forced, NEG means no positive voltages greater than +0.7 volts will be forced, SYM means that positive and negative voltages may be forced, and number is the absolute value of the maximum voltage to be forced.

There are 15 values at which the PMU can be clamped (positive or negative). They are: 4, 6, 8, 10, 12, 14, 16, 18, 20, 23, 25, 28, 31, 35, and 39 volts. Any number will be accepted by the SET CLAMP statement and the next higher (if not equal) clamp value will be selected. For example, if SET CLAMP POS 8.2; is given, the allowed voltages are to be between 0.7 and 10 volts. Actual clamp values produced by the hardware will be $\pm$ 1 volt plus or minus 10% the value specified.

11.2.3 <u>Enable</u>

General Form:

```
ENABLE [ILO/IHI/VLO/VHI] (GT/LT) number;
```

Time Delay Generated:

0

Description:

These instructions enable limit comparisons to be made on all programmed current/voltage operands prior to an instruction execution. If the operand fails to be within the LIMIT bounds, a system terminal error is issued. (See Appendix F). Instructions with operands in the LIMIT bounds are executed.

The "pass" regions are established by the LIMIT pairs ILO, IHI for currents and by the LIMIT PAIRS, VLO, VHI for voltages. ~~If the RANGE option is null, range 3 is invoked.~~ The absence of ENABLE LIMIT instructions in a program allows all magnitudes less than the range limits to pass. The function of the limit comparison is to protect the device under test where source forcing parameters are calculated or may be unknown. Once the program is operational and safe parameters are known, there instructions may be removed for execution time efficiency.

Note: Specification of limits with these statements will increase test execution time.

Examples:

> Enable limits to "pass" voltages which are between the values of +5.0 and 0 volts.

11.2.3.a DISABLE

General form:

DISABLE [ILO/IHI/VLO/VHI];

Time delay generated:

> 0

Description:

These instructions nullify the ENABLE limit comparisons invoked by the instructions of 11.2.3.

Time Delay Generated:

> 0

Description:

These statements establish program branch control on DC test failures (DCT), functional test failures (FCT), and current trip failures (TRIP). The label specifies the branch location. The current trip branch has priority in the event a DC or functional failure occurs simultaneously with a current trip failure. The label must be in the outermost block of the test program in all cases.

When a trip, functional or DC failure occurs at statement n and if the corresponding ON statements have not been processed, the program control will resume at statement n+1.

Example:

Set branch points to alter program control when functional, DC and current trip failures are detected.

```
ON DCT, DFAIL:
ON FCT, FFAIL;
ON TRIP, CFAIL;
```

11.2.5 Socket Identification

General Form:

```
SOCKET ID number;
```

Time Delay Generated:

0

Description:

This statement compares the value of the number with the identifier code of the performance board in the test socket. If the values are not equal, a system terminal error is issued and the program aborts. If the values compare, the program execution continues. The maximum Socket ID is 4095 (7777B).

Example:

Test the socket ID for a value of 4095.

```
SOCKET ID 4095;
```

11.3 PROGRAMMABLE POWER SUPPLY STATEMENTS

This section describes the statements which provide programmable control of the Sentry-400 power supplies. The power supplies can force either voltage or current and can detect the resulting voltage or current.

11.3.1 Force DPS Voltage Supplies

General Form:

FORCE [VF1/VF2/VF3] expression (,[RNG2/<u>RNG3</u>]);

Time Delay Generated:

1.75 milliseconds or delay set by the SET DELAY, DC statement.

Description:

The instruction forces the programmable voltage forcing supplies to the value specified by the expression. If the range is not specified, then the highest range is set. This instruction automatically connects the addressed unit to the test station load board.

The VF units will automatically disconnect under the following conditions:

(1) when the magnitude of the supply current, |i|, is greater than 150 milliamps and the trip register is in Range 2.

(2) when the magnitude of the supply current, |k|, is greater than 1.50 amps and the trip register in in Range 3.

Example:

Force units VF1, VF2, VF3 to +8, -5 and -30 volts respectively.

```
FORCE VF1 8, RNG2;
FORCE VF2 -5, RNG2;
FORCE VF3 -30;
```

11.3.2 Force DPS Current

General Form:

ENABLE [TRIPV1/TRIPV2/TRIPV3] [LT/GT] expression (,RNG2/<u>RNG3</u>);
FORCE [IF1/IF2/IF3] expression (,RNG2/<u>RNG3</u>);

Time Delay Generated:

1.75 millisecond

Description:

The ENABLE statement signifies that the DPS unit is to be put into the voltage trip/current force mode. Then the Force Statement specifies the current to be forced.

Examples:

```
ENABLE TRIPV1 LT 5, RNG3;
FORCE IF1 100E-3, RNG3;

ENABLE TRIPV3 GT 4, RNG2;
FORCE IF3 -5E-3, RNG2;

ENABLE TRIPV2 GT 1;
FORCE IF2 30E-2;
```

Note: An ENABLE TRIPV statement must precede the FORCE IF statement or a voltage will be forced rather than a current.

11.3.3 Enable Trip

General Form:

```
ENABLE [TRIP1/TRIP2/TRIP3] [LT/GT] expression [,(RNG2/RNG3)];
```

Time Delay Generated:

1.75 milliseconds or delay set by the SET DELAY, DC. statement.

Description:

These instructions enable the current-trip detector of the corresponding voltage forcing unit VF1, VF2, VF3. If the source/load current of the forcing unit, VF, exceeds the enabled trip value during a test sequence, indicating a DC failure, then program control is transferred to the instruction as specified by the "ON TRIP" instruction, if that instruction was given. If an "ON TRIP" has not been executed prior to the "trip" the program proceeds normally. At a pause or end-of-test, the parameter fail indicator will be lit if a trip occurred. If the TRIP datalogger is set, the value specified by this instruction will be written on the output device.

The VF units will automatically disconnect under the following conditions:

(1) when the magnitude of the current is greater than 150 milliamps and the trip register is in Range 2.

(2) when the magnitude of the current is greater than 1.50 amps and the trip register is in Range 3.

The automatic disconnect is a safety feature which protects both the device-under-test and the DPS units. The trips will be processed, provided they have been enabled, even though the automatic disconnect occurs. The trips are automatically disabled for 1.75 milliseconds, following a programmed instruction to the VF units. If a trip occurs after this time interval, the trip interrupt will occur at the conclusion of the 1.75 milliseconds delay. This feature allows for surge currents when the VF units are driving capacitive loads. Trips are automatically inhibited any time the 'tester busy' status is on. If a highly capacitative load is present, the current trip may be delayed longer by inserting a FORCE DELAY after FORCE VF or by using the ENABLE TRIP several instructions after the FORCE VF.

Example:

Enable the voltage forcing unit VF1 so that it will trip on load currents exceeding 100mA and VF2 to trip on currents more negative than -50mA.

```
ENABLE TRIP1 GT 100E-3, RNG3;
ENABLE TRIP2 LT -0.05, RNG2;
```

Caution:

[illegible] The TRIP
l be enabled

Add 11.3.3a

DISABLE TRIPS

General form:

DISABLE TRIPS;

The purpose of this instruction is to nullify any previous ENABLE TRIP instructions. The FACTOR programmer is thus allowed to selectively enable the "ON TRIP" instruction execution or the TOPSY/PAUSE ON FAIL command.

RNG3);

Time Delay Generated:

1.75 milliseconds

Description:

This statement signifies that the selected DPS unit is to be put into the current forcing mode.

Example: Enable trip interrupts if the voltage on DPS1 is more negative than -10 volts or if the voltage on DPS3 is more positive than 30 volts.

```
ENABLE TRIPV2 LT -10.0, RNG2;
ENABLE TRIPV3 GT +30;
```

11.3.5 Disconnect DPS Unit

General Form:

```
XCON [VF1/VF2/VF3];
```

Time Delay Generated:

1.75 milliseconds

Description:

This statement disconnects the specified voltage forcing unit from the test head. The magnitude of the specified unit is automatically set to 0 in the low range prior to disconnecting. The DPS units will automatically disconnect from the test head at End of Test without this instruction. Also, the trip function for the specified unit is disabled.
Example:

Disconnect all VF units

```
XCON VF1
XCON VF2;
XCON VF3;
```

11.4 SET LOGIC

General Form:

```
SET LOGIC [POS/NEG];
```

Time Delay Generated:

0

Description:

These statements initilize the functional test comparator logic pass conditions for either positive or negative voltage logic for the device under test (DUT). If this instruction is not used, the positive logic condition is assumed.

For positive logic the pass conditions are defined as follows:

(a) F(i) = 1 (expected output function for pin (i) = 1)
Pass = DUT OUTPUT SIGNAL>S1, otherwise fail.

(b) F(i) = 0 (expected output function for pin (i) = 0)
Pass = DUT OUTPUT SIGNAL<S0, otherwise fail.

For negative logic the pass conditions are defined as follows:

(a) F(i) = 1 (expected output function for pin (i) = 1)
Pass = DUT OUTPUT SIGNAL<S1, otherwise fail.

(b) F(i) = 0 (expected output function for pin (i) = 0)
Pass = DUT OUTPUT SIGNAL>S0, otherwise fail.

Note that all voltages, plus or minus, are treated algebraically, thus -10 is less than -1.

Example:

```
SET LOGIC NEG;
```

11.4.1 Force Voltage Conditioner References

General Form:

```
FORCE (E0/E1/EA0/EA1/EB0/EB1/EC0/EC1) expression (,[RNG2/RNG3]);
```

Time Delay Generated:

0.54 milliseconds

Description:

These instructions force the voltage conditioner reference supplies to the programmed values. If the range is not specified, then the lowest range is automatically set.

E0, E1, EA0 and EA1 are voltage conditioner references. The truth table below shows the relationship of F, S and these supplies. The supplies EB0, EB1, EC0 and EC1 are optional and connect to the test station load boards for use as bias supplies.

Example:

Force the standard reference pair to 3.5 and .5 volts respectively for the "1" and "0" levels.

```
FORCE E1 3.5, RNG2;
FORCE E0 .5, RNG2;
```

The following combinations of F and S bits per pin determine which reference voltage is applied to the pin.

F	S	
0	0	E0
1	0	E1
1	1	EA1
0	1	EA0

11.4.2 Set Reference Supplies for Functional Test Comparators

General Form:

SET [S1/S0] expression (,[RNG2/<u>RNG3</u>]);

Time Delay Generated:

0.54 milliseconds

S1 and S0 are reference supplies for the Functional Test Comparators. S1 is the reference level for the expected logic "1" levels and S0 is the reference level for the logic "0" levels. The value of the expression is loaded into the functional test comparator reference voltage supply register.

For testing positive logic, S1>S0. For testing negative logic, as defined by the instruction SET LOGIC NEG, S1<S0. The following table shows pass/fail decisions made by the comparators:

POS LOGIC:		NEG LOGIC:	
S1	$\underline{F}=\underline{1}$ $\underline{\text{pass}}$	S0	$\underline{F}=\underline{0}$ $\underline{\text{pass}}$
	F=1 fail		F=0 fail
S0	$\underline{F}=\underline{0}$ $\underline{\text{fail}}$	S1	$\underline{F}=\underline{1}$ $\underline{\text{fail}}$
	F=0 pass		F=1 pass

Example:

Set S0 comparator reference voltage to -5 volts:

SET S0 -5, RNG2;

11.4.3 Functional Test Statements

SET D/M/R/S/F binary pattern for DEFINITION, MASK, SELECT, RELAY or FUNCTION.

Description:

This statement controls five functions which can be used to control the programmed functional test for each pin. The five functions are: D, M S, R and F. They are each described as follows:

D This sets the DEFINITION pattern for the input and output pins. A 1 defines the pin as an input and the corresponding driver amplifier will be connected. A 0 defines an output pin. In either case, the level detectors are connected to the pin. Time Delay Generated: .54 milliseconds.

M This control condition sets the MASK pattern which defines the pins on which test results will be measured. A 0 means a don't care for that pin. A 1 means a care and the comparators will be enabled for that pin. Time Delay Generated: 0.

S This control condition sets the SELECT pattern which specifies the alternate or standard forcing logic level pair for each pin defined as an input. A 0 selects the standard reference pair E1 or E0. A 1 selects the alternate reference pair EA1 or EA0. Time Delay Generated: 0.

R This control condition sets the utility RELAY pattern which specifies which relays are closed and which relays are open. A 1 closes the corresponding utility relay. Time Delay Generated: .54 milliseconds.

F This control condition sets the binary FUNCTION pattern for the logical input states and the expected output states. For an input pin, a 1 specifies that the input level will be that specified by a 1 reference (E1 or EA1). For an output, a 1 means the output voltage is compared to S1 reference. Time Delay Generated: Value specified by SET DELAY statement.

The general form of the statements for programming the test conditions is:

```
SET (D/M/S/R) binary pin pattern;
SET F binary pattern, binary pin pattern...;
```

The binary pin pattern is defined by a binary value of up to 240 bit which has a one to one correspondence between pin and pattern bit location. Only pins that change from the previous state need be specified by the SET statement. Even if all pins are specified, only codes for the ranks* in which pin data changes are generated for maximum test rate.

The binary pin pattern can be programmed by either specifying each bit of the pattern or else by use of the following operators: [n] = pin origin and (:) = pattern replicator. These two operators are defined as follows:

[n] = pin origin operator
where: n is an integer $1 \leq n \leq 240$

(m: b_p) = binary pattern replicator operator
where: m is an integer $1 \leq m \leq 240$
and b_p is the binary pattern ≤ 240 bits

All non-addressed pins will take on the previously specified values. The absence of a previous specification is interpreted as a binary 0. Furthermore, non-addressed ranks will not be affected, i.e. no code will be generated and hence, at runtime, the non-addressed ranks will not be modified.

A sequence of binary patterns, which are separated by commas ",", represent a series of functional tests. To illustrate the use of these statements, two examples are shown below. The first example uses the origin and replicator operators. The second example yields the same binary pin patterns, but does not make use of the operators.

Example (1):

```
SET F   (3:0) (13:1) (2:0),
        (3:101) 010 (6:1),
        [8] (2:1) (9:0);
```

Example (2):

```
SET F   000  111  111  111  111  100,
        101  101  101  010  111  111,
        101  101  111  000  000  000;
```

* For a description of Ranks - See Appendix B.3

Explanation:

(3:0) means three 0 values specified, (13:1) means thirteen 1 values, (2:0) means two 0 values, (3:101) means 3 sets of 101 values, 010 means set the next three pins to this pattern, etc., etc. The [8] means: preserve the previous pattern and start at pin 8 with the following specifications. (Note pins are numbered from 1).

It should be noted that blanks are ignored within the binary pin pattern.

11.4.4 Set D

General Form:

SET D binary pin pattern;

Time Delay Generated:

0.54 milliseconds

Description:

This instruction loads the D register. A binary 1 in the pattern field specifies the corresponding test pin to be used as an input, thus connecting the pin to a voltage driver via a reed relay. A binary 0 in the pattern field specifies the corresponding test pin to be an output, thus disconnecting this pin from the voltage driver. In both cases, the test pin remains connected to the corresponding pin voltage level detector.

11.4.5 Set F

General Form:

SET F binary pin pattern, binary pattern...;

Time Delay Generated:

Specified by SET Delay instruction.

Description:

This instruction loads the F register. A binary 1 in the binary pin pattern specifies a logic 1 level and a binary 0 specifies a logic 0 level. The bits in the binary pin pattern corresponding to inputs, as specified by (D), are the forcing function. The bits in the binary pin pattern corresponding to outputs, as specified by (D), are the expected outputs. The input forcing analog voltage levels and the expected output comparison thresholds are determined by the contents of the programmable reference supplies S0, S1, E0, E1, EA0, EA1, and by the S register.

11.4.6 Set M

General Form:

SET M binary pattern;

Time Delay Generated:

0

Description:

This instruction loads the M register. A binary 1 in the binary pin pattern enables the associated pin level detector. A binary 0 in the binary pin pattern disables the associated pin level detector. These two states are referred to respectively as the "care" and "don't care" conditions for pins on which functional test results will be measured. The functional test results are strobed from the level detector outputs to the comparison register C, for "care" pins. The strobe is inhibited for "don't care" pins.

11.4.7 Set S

General Form:

SET S binary pin pattern;

Time Delay Generated:

0

Description:

This instruction loads the S register. A binary 0 in the binary pin pattern selects the standard logic level pair E1/E0 as input forcing voltages for the corresponding test pin. A binary 1 selects the alternate logic level pair EA1/EA0.

11.4.8 Set R

General Form:

SET R binary pin pattern;

Time Delay Generated:

0.54 milliseconds

Description:

This statement (instruction provides the control for opening or closing the utility relays. There is only one utility relay which is associated with each tester pin. A 1 in the binary pin pattern will close a utility relay and an 0 will open a relay.

SET (D/M/R/S/F)* binary pattern;

This instruction executes just like the normal long register instructions, however the Compiler will generate data for all pins specified regardless of the previous state of the register.

Example:	Instruction	Compiled Data
	SET F (60:1);	06077777 06177777 06277777 26377777
	SET F (59:1) 0;	26337777;
	CALL X;	
	SET F * (60:1);	06077777 06177777 06277777 26377777

In the example, note that the second SET F generated only one word. Normally, words are generated only for ranks with data that changes. However, the third instruction follows a subroutine call and at run time, the subroutine X may alter the F register. To insure that all 60 pins get set to 1, the programmer may use the asterisk. This asterisk will force the compiler to generate data for all ranks used in the instruction.

11.4.9 Force Strobe

General Form:

```
        FORCE STROBE;
```

Time Delay Generated:

        0

Description:

This instruction forces a single functional test strobe, thus transferring the functional comparator output states to the C register. The strobe is generated, even though the COMPARATORS may be disabled (11.4.11). The detection of a failure is processed in the same manner as is normally done during functional testing.

Example:

```
        FORCE STROBE:
```

11.4.10 Enable Latches

General Form:

        [ENABLE/<u>DISABLE</u>] LATCHES;

Time Delay Generated:

        0

Description:

The DISABLE instruction initializes the functional test control so that the C register is cleared prior to strobing the functional test comparators for each functional test.

If no latch statement is made, the disable latch mode is assumed.

The ENABLE instruction initializes the functional test control so that the C register is not cleared prior to strobing the functional test comparators. In this mode all functional failures are retained in the C register throughout a sequence of tests.

Examples:

Enable the comparison register to retain a history of all functional failures throughout a sequence of functional tests.

ENABLE LATCHES;

Disable the comparison register latches so that the C register contains only the current functional test failures.

DISABLE LATCHES;

11.4.11 Enable Comparators

General Form:

[ENABLE/DISABLE] COMPARATORS;

Time Delay Generated:

0

The ENABLE instruction initializes the functional test control logic so that the comparator outputs will be strobed to the C register for each functional test. The Enable comparator state is assumed if no statement is given.

Also the DISABLE instruction initializes the functional test control logic so that the comparator output will not be strobed to the C register for each functional test.

The disable comparator instruction should be issued when a series of functional patterns are to be executed but the device under test response is not defined.

Example:

Enable the comparators.

ENABLE COMPARATORS;

11.4.12 Enable Strobe

General Form:

ENABLE STROBE binary pattern;

Time Delay Generated:

0

Description:

This instruction enables the functional test comparator strobe to be controlled by the contents of F (1-4) and the "binary Pattern" (or equivalently the 4 bit "value"). The "binary pattern" for this instruction must be a 4-bit binary number and it is defined as follows:

binary pattern = $b_1b_2b_3b_4$,
where the subscripts refer to the tester pin number and b is either a 0 or a 1.

After executing the ENABLE STROBE instruction, all subsequent functional test results will be strobed to the C register according to the following logical condition ('.' AND, '+' OR):

$$\text{STROBE} = b_1 \,.\, F(1) + b_2 \,.\, F(2) + b_3 \,.\, F(3) + b_4 \,.\, F(4)$$

Example:

Enable the strobe to be controlled by F(1).

```
ENABLE STROBE 1000;
```

Note: This statement automatically DISABLES the normal comparator strobe, i.e., DISABLE COMPARATORS

11.5 PRECISION MEASURING UNIT STATEMENTS

This section describes the statements which control the operation of the Sentry-400 Precision Measuring Unit. This unit is used for measuring device DC parameters and for system self check.

11.5.1 Set PMU Ranges

General Form:

```
SET PMU [SENSE/FORCEV/FORCEI], [RNG0/RNG1/RNG2/RNG3/AUTO];
```

Time Delay Generated:

1.75 milliseconds

Description:

This instruction initializes the Precision Measuring Unit (PMU). The force and sense functions are complimentary with respect to voltage and current; i.e. when the unit is set to force voltage (or current) it is automatically initialized to sense current (or voltage). When sensing in Auto range, the first measurement is made in Range 3, then Range 2 if necessary and so on, downward.

Example:

```
SET PMU SENSE, AUTO;
SET PMU FORCEI, RNG3;
```

In this example the statements initialize the PMU to force current in range 3 and sense voltage with auto ranging.

11.5.2 Force Voltage/Current

General Form:

```
FORCE [VOLTAGE/CURRENT] expression (,[RNGO/RNG1/RNG2/RNG3]) (HOLD);
```

Time Delay Generated:

1.75 millisecond with range change, or
0.54 milliseconds with no range change, or
that specified by the SET DELAY DC statement
(whichever is greater).

Description:

This instruction is used to force a programmed voltage or current via the precision measurement unit.

The HOLD form loads the master side of the precision measurement unit forcing register, PPS, and, hence, the output voltage or current does not change from its existing value. The held value is transferred to the slave side of the register (the output of the unit then takes on the held value) whenever an EXECUTE occurs as a result of a SET F instruction.

Caution:

Do not attempt to perform a FORCE and HOLD, followed by another FORCE without an intervening SET F or else the actual value forced will be in error.

The EXECUTE form loads the loads the slave side of PPS and hence the output of the precision measurement unit will begin to slew to the programmed value upon execution of the instruction. If the range is not specified, then the highest range is automatically set.

Example:

Force the output of the precision measurement unit to -1 microamp.

```
FORCE CURRENT -1E-6, RNG1;
```

11.5.3 Force PMU

General Form:

FORCE PMU expression (,HOLD);

Time Delay Generated:

0.54 millisecond or that specified by a SET DELAY DC statement, (whichever is greater).

Description:

The expression of this instruction is scaled according to the mode, V or I, and range as is preset by the SET PMU SENSE/FORCE ....statement. If AUTO ranging has been preset, then the range which gives best resolution will be automatically determined (at run-time) prior to loading the PPS register (if the expression is a variable). This instruction is especially useful when several PMU forcing values in the same range are to be executed, but it has a slightly longer execution time than FORCE VOLTAGE/CURRENT.

This instruction is used in conjunction with the conditions established by:

SET PMU [SENSE/FORCEV/FORCEI], [RNGO/RNG1/RNG2/RNG3/AUTO]

Example:

Force the output of the precision measurement unit to -1 microamps.

SET PMU FORCEI, AUTO;

FORCE PMU -A/B;

11.5.4 Connect PMU

General Form:

CPMU PIN expression;

Time Delay Generated:

0.54 milliseconds

Description:

This statement connects the precision measurement unit to the pin number specified by the expression (or equivalent "value"). Device-under-test (DUT) pins are specified by values in the range 1 to 240_{10}. Calibration nodes are specified by node numbers 254_{10} and 255_{10}. If numbers other than these are programmed the tester will halt with a terminal error. Pin numbers 376B (254_{10}) and 377B (255_{10}) are, respectively, connections to an open circuit and calibration network which is located in the Test-Head. The calibration network consists of fixed precision resistors. The value of the connected resistor is a function of the force and measure ranges as shown in Appendix G.

Example:

> Force 10 volts with a 100K ohm resistor as a load. Connect with 0 volts applied (assume PMU is initially not connected to a load).

```
SET PMU Sense, RNG1;
FORCE VOLTAGE 0, RNG2;
CPMU PIN 377B;
FORCE VOLTAGE 10, RNG2;
```

11.5.5 <u>Measure Value/Node</u>

General Form:

```
MEASURE (VALUE/NODE number) [,LOG];
```

Time Delay Generated:

> 0

Description:

This statement initiates an analog-to-digital conversion within the precision measurement unit.

When the VALUE option is specified, the measurement is made according to the existing conditions. The measured value is appropriately scaled to floating point format and is stored in the system global variable: VALUE. If DC trips are enabled (see Section 11.5.8) VALUE is tested to determine if it falls within the pass window. If it passes, the "DC pass" indicator is set. If it fails, the "fail" indicator is set. If the option [LOG] is specified, then the measured value will be logged according to the MONITOR logging command conditions (see SENTRY-400 TOPSY Manual).

If AUTO ranging has been spedified, the system will automatically determine the measuring range which gives best resolution. Auto-ranging begins with Range 3 and it ranges downward until either the best resolution is obtained or until Range 0 is encountered.

Use of the [NODE number] option provides the means for measuring a parameter at a system internal monitor node point. The node numbers and their descriptions are summarized in Appendix H. Ranging and measuring conditions are automatically invoked.

For internal nodes, the measured value and logging and pass/fail conditions are the same as was previously described. At the conclusion of an internal node measurement cycle, the precision measurement unit is automatically disconnected and it is initialized to force 0 current in Range 1.

Internal nodes 202B to 205B for the E reference supplies are divided by 8. That is, when a reference supply E1, E0, etc. is programmed to V volts, the reference value is actually V/8 volts since the buffer amplifier at each pin has a gain of eight.

Example:

Measure and log the current from VF1

```
MEASURE NODE 217B, LOG
```

Datalogging options specified by the TOPSY monitor are:

1) Datalog MEASURE: All 'MEASURE VALUE' statement are logged.
2) Datalog LOG: 'MEASURE VALUE, LOG'; statements are logged i.e. only those having the 'LOG' modifier.
3) Datalog DCT: All 'MEASURE VALUE'; statements that fail to pass the limits specified by ENABLE DCT statements are logged.

11.5.6 Measure Pin

General Form:

```
MEASURE PIN;
```

Time Delay Generated:

0

Description:

Measure Pin allows fast go/no go d.c. parameter tests. It is similar to MEASURE VALUE except that go/no go comparisons with ENABLE DCT limits are made in tester hardware format (and hence no floating point conversion is made, nor is the result stored in VALUE).

Since comparison is made in hardware format, ENABLE DCT must be executed after the PMU force and sense conditions are specified, so the DCT limit can be scaled properly.

If any datalogging conditions are specified (Datalog DCT or Datalog measure) then the resultant measurement will be converted to floating point as in the MEASURE VALUE instruction.

Example:

```
SET PMU SENSE, RNG1;
CPMU PIN 5;
FORCE VOLTAGE 4.5, RNG2;
ENABLE DCT1 GT 6.0 E-6;
MEASURE PIN;
```

11.5.7 Disconnect PMU

General Form:

```
XPMU PIN;
```

Time Delay Generated:

0.54 millisecond

11.5.8 Enable DC Parameter Limits

General Form:

```
ENABLE [DCTO/DCT1] [LT/GT] expression;
DISABLE [DCTO/DCT1];
```

Time Delay Generated:

0

Description:

The ENABLE forms a pass or fail threshold for level DCT1 and/or DCTO. Either one or both DCT thresholds may be specified; using only one DCT function specifies a level for comparison trips.

Using both DCT functions specified a "pass window" for subsequent DC measurements. The pass region is specified by the operators LT/GT and by the value of the expression in the ENABLE statements. The programmer must be careful not to specify an impossible pass condition.

When a MEASURE VALUE or MEASURE PIN does not pass the level specified by each DCT function, a DC fail is indicated and the program control is transferred to the instruction specified by the "ON DCT" instruction. If an "ON DCT" has not been previously executed, then the next instruction following the MEASURE is executed. At a pause or end-of-test a failure will light the parameter fail indicator.

The DISABLE form disables the comparison limits and inhibits the DC fail regardless of the measured value.

Example:

Enable DC trip limits which will pass all measured values between -2 milliamps and +2 microamps.

```
ENABLE DCT1 GT 2E-6;
ENABLE DCTO LT -2E-3;
```

11.5.9 Enable Relay - Connect PMU to Functional Circuitry

General Form:

```
[ENABLE/DISABLE] RELAY;
```

Time Delay Generated:

0.54 milliseconds

Description:

The DISABLE instruction initialized the pin address control logic such that the voltage conditioner for pin (n) will be automatically disconnected when the precision measurement unit is connected to pin (n). If no relay statement is made, the disable mode is assumed.

The ENABLE instruction initializes the pin address control logic such that the voltage conditioner for pin (n) will remain connected, even though the precision measurement unit is connected to pin (n). After connecting the precision measurement unit to pin (n) the voltage conditioner can be disconnected by executing the instruction: DISABLE RELAY; i.e., allowing a make-before-break sequence to maintain bias on a pin.

Example:

Connect the precision measurement unit to pin (10) with the voltage conditioner connected and then disconnect the voltage conditioner.

```
ENABLE RELAY;
CPMU PIN 10;
DISABLE RELAY;
```

11.6 AUXILIARY CLOCK STATEMENTS

This section describes the statements which control the auxiliary clock functions of the Sentry-400. The clock functions allow a string of up to 255 clock pulses to be generated for each functional test. In addition, these clocks can be selectively enabled or disabled for each functional test.

11.6.1 Set Clock

General Form:

SET CLOCK expression;

Time Delay Generated:

0

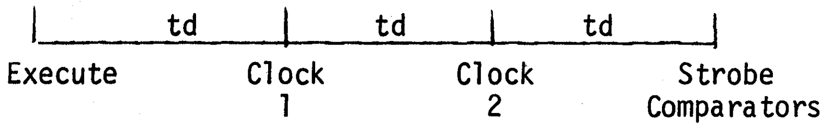
Description:

The value of the expression is loaded into the clock burst count register of the tester. The count specifies the number of clock signals which the tester will output for each functional test. The time delay between clocks is equal to the current contents of the time delay register. The maximum clock count is 255.

Example:

SET CLOCK 2;

The resultant timing is explained in the following diagram:



td = contents of time delay register +0.7 μsec

11.6.2 Enable Clock

General Form:

ENABLE CLOCK binary pattern;

Time Delay Generated:

0

Description:

This instruction enables clock signals to be connected to tester pins 1, 2, 3, 4, according to the "binary pattern" (or equivalently the 4 bit "value"). The binary pattern for this instruction must be a 4 bit binary number and it is defined as follows:

binary pattern = $b_1b_2b_3b_4$,
where the subscripts refer to the tester pin number and b is either a 0 or a 1.

Four clock sync lines for the first four pins are brought to the load board and convenience panel connector. These are used to drive an external clock generator. The clock generator may be as simple as an IC gate mounted on the load board or may be a complex 4-phase clock. The clock may be connected to the device-under-test (DUT) via a utility relay pin on the load board or with a separate relay to the DUT directly. If a separate relay is used, it may be driven by the clock relay (CRLY) signals at the convenience panel connector. Whenever the PMU is addressed to one of the clock pins, the clock relay will automatically open.

Enabled clock signal pins are disconnected from their corresponding functional test circuits. In addition to connecting the clock lines, this instruction enables the sync signals. Sync signals are programmed by the logical "and' condition of the enabled clock pins. The contents of F(1-4) is as follows:

$SYNC_1 = b_1 . F(1)$

$SYNC_2 = b_2 . F(2)$

$SYNC_3 = b_3 . F(3)$

$SYNC_4 = b_4 . F(4)$

The number of sync pulses and their periods are specified by:

SET CLOCK number; (clock burst counter)

Period = 0.7 microseconds + td, where td is the time delay specified by the SET DELAY instruction.

Example:

Enable clock signals to pins 1 and 4.

ENABLE CLOCK 1001;

11.6.3 Force Clock

General Form:

FORCE CLOCK;

Time Delay Generated:

- as specified by the SET DELAY instruction.

Description:

This instruction forces a single clock pulse to occur at each of the 4 SYNC lines. The width of the SYNC pulse is equal to the value specified by the SET DELAY instruction.

Example:

FORCE CLOCK;

11.7 MISCELLANEOUS CONTROL STATEMENTS

This section describes the statements which control the timing as well as initialization of the Sentry-400.

11.7.1 Force Reset

General Form:

```
FORCE RESET;
```

Time Delay Generated:

0

Description:

This instruction forces the test system into the reset state, thus clearing all programmable test conditions.

Example:

Clear the test system.

```
FORCE RESET;
```

Note: This does not cause a 'software' reset. Hence, functional tests succeeding a Force Reset must be preceded with the instructions: SET F (240:0); SET M (240:0), etc. to clear the software.

11.7.2 Force Delay

General Form:

```
FORCE DELAY;
```

Time Delay Generated:

- as specified by SET DELAY, DC statement.

Description:

This instruction forces a time delay to occur, prior to executing the next delay dependent instruction.

Example:

Provide a delay after programming the VF1 unit to provide settling time for the change of programmed voltage prior to executing a forced STROBE.

```
FORCE VF1 10.0;
FORCE DELAY;
FORCE STROBE;
```

11.7.3 <u>Force Wait</u>

General Form:

```
FORCE WAIT;
```

Time Delay Generated:

- the time required for the tester to become not-busy.

Description:

This instruction forces the tester to wait until the tester status is not-busy before processing the next instruction.

Note: The tester goes busy after executing those instructions with a non-zero time delay.

Example:

Provide a delay until the tester is not busy after forcing the E1 reference voltage.

```
FORCE E1 2.5 VOLTS;
FORCE WAIT;
```

11.8 EXTERNAL INTERFACE REGISTER READ/WRITE

General Form:

```
WRITE (EIR) expression;
READ (EIR) variable;
```

Time Delay Generated:

0

Description:

This fifteen bit register is used to display test results and control external handlers. Bits 0-9 are available to the programmer to use in any form, such as to define various pass categories. Bits 10-14 are defined by system software. All bits are read/write. If the user wishes to use some bits to read the status of external equipment, then a simple hardware modification can be made to the register by disconnecting the bit storage device from the register. Consult your field service representative if this is desired.

11.9 READING AND WRITING LONG AND SHORT REGISTERS

The Sentry-400 is capable of reading and writing both the long and short registers of the tester. The formats for porgramming the long and short registers are as follows:

```
WRITE (XXXXB) expression;
where:  XXXX is any register number and
        B is the octal indicator.
```

The format for reading information from a short or long register is:

```
READ (XXXXB) Y;
```

For more detailed coding information regarding the reading and writing of the long and short registers, see Appendix B.

Add 11.9

ENABLE ACCESS

General form:

ENABLE ACCESS;

Time Delay Generated:

30-75ms (one disc access)

This instruction will reallocate the user program in core such that the entire usable test core buffer is available to the instructions following it. It does this by re-reading the user's program from the disc with the next instruction being as close to the lowest available test buffer address as the system will allow.

APPENDIX A

Table A. Character Coding (TRASCII)

Code	Char.	029 Special Character	Code	Char.	029 Special Character
00	SPACE		40	@	
01	!		41	A	
02	"		42	B	
03	#		43	C	
04	$		44	D	
05	%		45	E	
06	&		46	F	
07	'		47	G	
10	(		50	H	
11	)		51	I	
12	*		52	J	
13	+		53	K	
14	'		54	L	
15	-		55	M	
16	.		56	N	
17	/		57	O	
20	0		60	P	
21	1		61	Q	
22	2		62	R	
23	3		63	S	
24	4		64	T	
25	5		65	U	
26	6		66	V	
27	7		67	W	
30	8		70	X	
31	9		71	Y	
32	:	0-8-2	72	Z	
33	;		73	[	<
34	<	12-0	74	\	¬
35	=		75	]	>
36	>	11-0	76	↑	|
37	?		77	←	_

APPENDIX B

READING AND WRITING OF LONG AND SHORT REGISTERS

B.1 INTRODUCTION

The Sentry-400 possesses additional READ and WRITE capabilities which use the system's long and short registers. (The long and short registers consist of one bus each). The following is a description of the operation of the long and short registers.

B.1.1 Long Registers

The long registers are interfaced to the memory interface unit, called the Instruction Register, which sends information to and from the test station. It has a bus which is used primarily for transmitting functional test data to the test station. The S, F, D, M, C and R registers are the only registers which are programmable with functional test data. There are other registers that are essentially like the short register, but since the hardware resides in the test station, they are interfaced to the long register data bus and use rank address bits for identification.

B.1.2 Short Register

The short register is interfaced to the computer accumulator. The register is used for controlling the digital to analog converter sub-systems and for communicating the tester status, mode and interrupt information. Therefore, the E1, E0, EA1, EA0, etc., registers, which contain data for reference supplies are interfaced to the short register data bus.

B.2 ADDRESSING SHORT REGISTERS

Each register can be addressed by the three computer SPU (Select Peripheral Unit) commands: READ, WRITE and SPECIAL. They are each discussed as follows:

READ

To examine the contents of a register, the register is first addressed with an SPU READ command. The contents of the register are then read into the CPU accumulator.

WRITE

To write a bit pattern into a register, the register is first addressed with an SPU WRITE command. The command is followed by the bit assignment for that register.

SPECIAL

An SPU SPECIAL command is defined as an instruction which executes some function, but no read or write data transfer is involved, e.g., Increment IND Counter or Disconnect DPS.

Each register command consists of an 8 digit octal code. The code in effect identifies a specific register in a specific unit and informs this register that it is about to either receive or transmit data. The data will be either read out of the register, written into the register or the register will perform a special function. The command format is shown below.

Consider an example of an SPU command so that its mode and function within the system can be understood. The following table shows the MODE register SPU READ command and its octal and binary equivalents:

Bit Location:	23 22 21	20 19 18	17 16 15	14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
Octal Value:	0	6	6	0	0	5	2	0
Binary Value:	0 0 0	1 1 0	1 1 0	0 0 0	0 0 0	1 0 1	0 1 0	0 0 0
	OP CODE (SPU)		6=Read 4=Write 2=Special 0=No-op	TESTER REGISTER			UNIT ADDRESS	

Starting from the left, the high-order six binary bits (23-18) represent the octal code 06. This octal code is the SPU op code for the READ, WRITE, or SPECIAL command functions. The op code informs the system that it is about to address a register in a unit with either READ, WRITE or SPECIAL information.

The 3-bit value in bits 17--15 defines the command as READ, WRITE, or SPECIAL transfer. Octal 6 = READ; 4 = WRITE; 2 = SPECIAL and 0 = No-op.

The six bits shown under tester register (13-8) specify one of 64 unique registers. The remaining bits (7-0) are used to form the unit address. The tester is unit 120B.

B.2.1 SHORT REGISTER DESCRIPTIONS

The short registers implemented in the Sentry-400 will be described below. The register number is the octal equivalent of bits 13-8 of the SPU command. Table B1 summarizes the short registers, their addresses and special functions.

B.2.1.1 Mode Register (MR)

The mode register controls modal functions affecting the total test system as shown below:

BIT	FUNCTION	Read	Write
0	Reset Tester Short Reg.		✓
1	Reset Tester Long Reg.		✓
2	Monitor Mode	✓	
3	Auto Mode	✓	
4	Negative Logic Mode	✓	✓
5	Latch C Mode	✓	✓
6	Strobe Inhibit Mode	✓	✓
7	Force Strobe		✓
8	Force Sync Pulse		✓
9	DMA Mode		✓
10	Manual Software Flag	✓	✓
11	Function Test Suspended	✓	✓
12	Trip Fail	✓	✓
13	Functional Fail	✓	✓
14	Pass (Cleared by 11 or 12)	✓	✓
15	Spare	✓	✓

NOTE: A SPECIAL command clears the mode register.

B.2.1.2 Status Register (SR)

The status register contains interrupt information as shown below:

BIT	FUNCTION	Read	Write
0	Instruction Number Compare Interrupt Enable	✓	✓
1	Instruction Number Compare Interrupt	✓	✓
2	Delay Complete Int. Enable	✓	✓
3	Delay Complete Int.	✓	✓
4	Trap Interrupt Enable	✓	✓
5	Trap Interrupt	✓	✓
6	Fail Interrupt Enable	✓	✓
7	Fail Interrupt	✓	✓
8	Trip Interrupt Enable	✓	✓
9	DPS #1 Trip (8 must be on)	✓	✓
10	DPS #2 Trip (8 must be on)	✓	✓
11	DPS #3 Trip (8 must be on)	✓	✓
12	Stop Interrupt Enable	✓	✓
13	Stop Interrupt	✓	✓
14	Interrupt in Process	✓	✓
15	Spare	✓	✓

A SPECIAL command clears the status register.

B.2.1.3 Instruction Register (IR)

The instruction register is a buffer between B memory and the long registers via the accumulator. It contains the following information:

BIT(S)	FUNCTION	Read	Write
0	Pin Data - Pin 1	✓	✓
.	. .	.	.
.	. .	.	.
.	. .	.	.
14	Pin 15	✓	✓
15-18	Rank Address		✓
19-21	Register Address		✓
22-23	Long Reg. Read/Write Control		✓

B.2.1.4 Memory Address Register (MAR)

The memory address register contains the memory address for the tester DMA mode. The fourteen bits are all read and write. When the tester is in the DMA mode, phase loop control automatically increments MAR.

B.2.1.5 Test Station Control Register

The test station control register controls four channel multiplexing as follows:

BITS	FUNCTION	Read	Write
0-1	Station Address	✓	✓
2-5	Start Requests from Stations 1 to 4	✓	
6-9	Manual Mode from Stations 1 to 4	✓	
10-13	Reset Request from Stations 1 to 4	✓	

B.2.1.6 Clock Burst Count Register

The clock burst count register consists of eight bits, all read/write, which contain the count of the number of clock syncs generated per function test.

B.2.1.7 Time Delay Register

The time delay register consists of fourteen bits, all read/write, representing the value of a functional or DC time delay to be generated by certain tester instructions. For function tests, the least significant bit represents 0.35 microseconds and full scale is 5.734 milliseconds. The phase loop counter triggers the time delay counter when a SET F instruction is executed. For DC tests, the least significant bit represents 0.35 milliseconds and the full scale value is 5.734 seconds. An SPU SPECIAL command triggers the DC time delay.

B.2.1.8 Instruction Number Compare Register

The instruction number compare register consists of sixteen bits, all read/write, representing the test instruction number at which a pause or external sync pulse occurs.

B.2.1.9 Instruction Number Display Register

The instruction number display register is a sixteen bit register, all sixteen bits read/write, representing the test instruction being executed. An SPU SPECIAL command increments the contents of the register by one.

B.2.1.10 Digitally Programmable Power Supply Registers DPS1, DPS2 and DPS3

Registers DPS1, DPS2 and DPS3 contain the range, polarity and magnitude of the DPS voltage being forced or the voltage trip point.

BIT(S)	FUNCTION	Read	Write
0-9	Voltage Magnitude LSB = 0.01 volt in low range	✓	✓
10	Polarity 0 = Pos 1 = Neg	✓	✓
11	Range 0 = low 1 = high	✓	✓

An SPU SPECIAL command disconnects the corresponding supply.

B.2.1.11 DPS Trip Registers - DPT1, DPT2 and DPT3

Registers DPT1, DPT2 and DPT3 contain the current trip point or the current being forced and the trip greater than or less than control plus the DPS forcing mode control.

BIT(S)	FUNCTION	Read	Write
0-9	Current Magnitude LSB = 0.1mA in low range	✓	✓
10	Polarity 0 = low 1 = high	✓	✓
11	Range 0 = low 1 = high	✓	✓
13	GT or LT:1 = GT 0 = LT	✓	✓
14	Voltage/current 0 = voltage force 1 = current force	✓	✓

B.2.1.12 Reference Voltage Supply Registers S0, S1, E0, E1, EA0, EA1, EB0, EB1, EC0, EC1

The reference voltage supply registers contain the range, polarity and magnitude of the reference voltage supply.

BIT(S)	FUNCTION	Read	Write
0-9	Voltage Magnitude LSB = 0.01 volt in low range LSB = 0.04 volt in high range	✓	✓
10	Polarity 0 = Pos 1 = Neg	✓	✓
11	Range 0 = low 1 = high	✓	✓

B.3 LONG REGISTER DESCRIPTION

The registers which are associated with the long register are divided into two groups. The first group consists of the D, M, S, R, F and C registers. The second group consists of Pin Address,Socket ID,Statement Number Display, Clock and Strobe, Precision Power Source, Precision Sense Level, and External Interface registers.

B.3.1 THE D, M, S, R, F AND C REGISTERS

The six registers of the first group are discussed below.

B.3.1.1 D Register

The D register is termed the Input/Output register. If the D register is programmed as a 1, the associated pin is defined as an input pin. Consequently, the relays are then energized so as to connect the output of the driver to the pin. If the D register is programmed 0, the associated pin is defined as an output pin; as a result, the relays are configured to disconnect the driver.

B.3.1.2 M Register

The M register is the "care/don't care" or "mask" register. The output of the M register is applied as an input to the detector. If the programmer is interested (care) in knowing the output level of a pin, the M register is programmed as a 1. The care or 1 condition enables the detector to make the comparison between the actual output and the expected output. If the programmer is not interested (don't care) in the output level of a pin, the M register is programmed as a "don't care", "mask", or "0". The "don't care" condition inhibits the output of the detector. Logic circuitry at the output of the detector will provide a 0 level to the C register, and the function test output will indicate a pass. An input pin, or an output pin with an undefined state, would normally be programmed as "don't care" to prevent false failure indications on that pin.

B.3.1.3 S Register

The select reference register selects which set of reference supplies are to be used by the functional test driver for each tester pin. A binary 0 selects the E0/E1 reference supplies, while a binary 1 selects the EA0/EA1 reference supplies.

B.3.1.4 R Register

The utility relay register controls the utility relays, one relay per tester pin. A binary 1 indicates a closed relay and a binary 0 indicates an open relay. The utility relays can be used for such functions as connecting a load resistor for an output pin to a programmable power supply.

B.3.1.5 F Register

The F register contains the logic patterns 1 or 0 to be applied to those pins which are defined as input pins by the D register. If the F register is programmed as a high level (1), the F register will cause the high output of the driver to be applied to the associated pin. If the F register is programmed with a low level (0), the low output of the driver will be applied to the pin. The F register also contains the expected logical output of those pins which are defined as output pins.

B.3.1.6 C Register

The C register stores the go/no-go results of a comparison between the actual output of a device and the expected output. A binary 1 represents a comparison failure and a binary 0 represents a pass condition. If the signal is high (fail), the output of the register is displayed at the test console. If the signal is low (pass), no signal will be displayed.

B.3.2 FORMAT OF FUNCTIONAL TEST WORD

The primary difference between the long and short registers is that the short registers consist of fixed bit test words. The long registers are variable length words, with the same format as those registers which are associated with the short register. The format of a 24-bit function test word is discussed below and it is illustrated in the accompanying Figure B1. The format for all function test registers is the same, except that the address for each register is different.

0 1 0 | 1 1 0 | 0 0 0

24 Bit Functional Test Word

23 22	21 20 19	18 17 16 15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Control	Register Address	Rank Address	Pin Data Field

Rank Address

4 Bits = 1 to 16 Ranks
Maximum Register Length = 16 x 15 = 240 Bits

Register Address

Bits: 21	20	19	Meaning
0	0	0	OPEN
0	0	1	D DEFINE I/O PINS
0	1	0	M MASK (CARE/DON'T CARE)
0	1	1	F FUNCTIONAL PATTERN
1	0	0	S SELECT ALTERNATE REFERENCE
1	0	1	C COMPARE (FAIL PATTERN)
1	1	0	R UTILITY RELAY
1	1	1	SPECIAL TEST STATION REGISTERS

Control

Bits: 23	22	Meaning
0	0	WRITE AND HOLD
0	1	WRITE AND EXECUTE
1	0	READ
1	1	TRAP

Register Load Times

1.75 (1 + N) microseconds
where: N = Number of Ranks Changing

Figure B.3.2-1. Test Word Function Format

Starting from the right, the 0 bit represents pin 1, the 1 bit represents pin 2, etc., up to bit 14 which represents pin 15.

Rank Address

Bits 15 through 18 represent the rank to which the first 15 bits have been assigned. The ranks are determined by the normal 8-4-2-1 binary code minus one. 0000 inserted in bits 15 through 18 represent rank 1; 1111 represent rank 16. In this manner, 16 ranks of 15 pins can be programmed, thus providing a capacity of 240 pins.

Register Address

Bits 19, 20 and 21 determine the register to which the rank of 15 bits is to be sent.

Control

Bits 22 and 23 control the read/write function. Assume the device under test is a 15-pin device; all 15 pins will be programmed on one line and will be assigned to rank 1. The correct address code is inserted and bits 22 and 23 are programmed as 01 (write and execute), i.e., there are no more pins to program. If the device is a 45-pin device, the first 15 pins are programmed as described above, except for bits 22 and 23 which are programmed as 00 (write and hold), i.e., there are more pins to program. The second group of 15 pins is assigned to rank 2 and, again, bits 22 and 23 are programmed as 00, i.e., there are more pins to program. The third group of 15 pins is assigned to rank 3 and bits 22 and 23 are programmed as 01 (write and execute), i.e., there are no more pins to program-execute the command. The F and S registers are MASTER/SLAVE so that all bits execute together.

B.3.3 SPECIAL TEST STATION REGISTERS

The seven registers of the second group are discussed below.

B.3.3.1 Pin Address Register (Rank Address = 0)

The pin address register addresses the precision measuring unit to one of the pins of the device-under-test or to an internal node.
The reset state of this register is 377B, the calibration node.

BIT(s)	FUNCTION	Read	Write
0-3	Pin Number 1-15		✓
4-6	Rank Number 1-8 (or Rank 9-16)		✓
7	Internal Node Address		✓
8	Connect Voltage Conditioner (Enable Relay FACTOR Instruction)		✓

B.3.3.2 <u>Socket ID</u> (Rank address = 1)

The socket ID register reads a hard wired address on the load board so that a FACTOR program can compare the load board ID with the program ID.

The register is 12 bits, read only.

B.3.3.3 <u>Statement Number Display Register</u> (Rank address = 2)

The statement number display register contains the statement number to be displayed on the test station control panel. This register is interfaced to the IND register with software in TOPSY. Whenever the test sequence pauses, the software updates SND.

The register is 15 bits - both read and write.

B.3.3.4 <u>Clock and Strobe Register</u> (Rank address = 3)

The clock and strobe register is actually two registers under one address. The first part, clock address, is bits 0-3. The clock address bits are ANDed with the first four bits of the F register to generate clock sync signals, (see Enable Clock Description, 11.6.2). The second part, Enable Strobe, is bits 4-7, (see the Enable Strobe Description, 11.4.12). All eight bits are read/write.

B.3.3.5 <u>Precision Power Source Register/Precision Measurement Unit Forcing Register</u> (Rank address = 4)

This register contains the magnitude, polarity and range information of the PMU forcing value. It also contains the voltage or current force mode bit.

BIT(S)	FUNCTION			Read	Write
0-9	Magnitude			✓	✓
	LSB	Range	Full Scale		
	1mV	1	1.023V		
	10mV	2	10.23V		
	40mV	3	40.92V		
	1nA	0	1.023μA		
	100nA	1	102.3μA		
	10μA	2	10.23mA		
	100μA	3	102.3mA		
10	Polarity	0 = POS		✓	✓
		1 = NEG			
11-12	Forcing Range	00 = Range 0		✓	✓
		01 = Range 1			
		10 = Range 2			
		11 = Range 3			
13	VF/IF	1 = Voltage Force		✓	✓
		0 = Current Force			
14	Execute			✓	✓

B.3.3.6 Precision Sense Level Register (Rank address = 5)

This register serves several functions which are to contain (1) the measuring range, (2) the PMU voltage clamp levels and (3) the results of the analog to digital (A/D) conversion. Some bits (0-10) serve a dual purpose, (a) as read only and (b) as write only.

	BIT(S)	FUNCTION	Read	Write
(a)	0-9	A/D Conversion Magnitude	✓	
	10	A/D Conversion Polarity	✓	
(b)	0-5	Clamp Magnitude		✓
	6-7	Spares		✓
	8	Positive Clamp On		✓
	9	Negative Clamp On		✓
	10	Clamp Range 1 = 40 volt 0 = 10 volt		✓
	12-11	Measuring Range 00 = Range 0 01 = Range 1 10 = Range 2 11 = Range 3	✓	✓
	13	Vm/Im 1 = Voltage Measure 0 = Current Measure	✓	
	14	Start A/D Conversion		✓

B.3.3.7 External Interface Register (Rank address = 6)

This fifteen bit register is used to display test results and control external handlers. Bits 0-9 are available to the programmer to use in any form, such as to define various pass categories. Bits 10-14 are defined by system software. All bits are read/write. If the user wishes to use some bits to read the status of external equipment, then a simple hardware modification can be made to the register by disconnecting the bit storage device from the register. Consult your field service representative if this is desired.

BIT(S)	FUNCTION	Read	Write
0-9	Defined by User Displayed on Station Control Panel	✓	✓
10	D.C. Fail Lamp	✓	✓
11	D.C. Pass Lamp	✓	✓
12	Functional Fail Lamp	✓	✓
13	Functional Pass Lamp	✓	✓
14	End-of-Test	✓	✓

B.3.3.8 Slave Test Stat. Control reg (Rank add 7)

B.4 FORMATTING OF FACTOR WRITE AND READ STATEMENTS

The format for programming the short or long registers is:

```
WRITE (XXXXB) expression;
where:     XXXX is any register number, and
           B is the octal indicator
```

Reading information from a short or long register is:

```
READ (XXXXB) Z;
```

The following tables provide the necessary information for reading from or writing to a SPECIFIC register for a SPECIFIC function on the long and short registers.

Table B.4-1. Short Register Reading and Writing Codes

		A=1= SPECIAL
		A=2= WRITE
		A=3= READ	

Reg. No.	Register	X X X X	SPECIAL Function
0	No-op		
1	MODE	A 0 0 2	Clear Mode Reg
2	STATUS	A 0 0 4	Clear Status Reg
3	Instruction	A 0 0 6	
4	Memory Address	A 0 1 0	
5	TSC	A 0 1 2	
10	Clock Burst Count	A 0 2 0	
11	Time Delay	A 0 2 2	Start D.C. Delay
14	Instruction Number Display Counter	A 0 3 0	Increment Count
15	Instruction Number Compare	A 0 3 2	
21	DPS1	A 0 4 2	Disconnect DPS1
22	DPS2	A 0 4 4	Disconnect DPS2
23	DPT3	A 0 4 6	
24	DPS3	A 0 5 0	Disconnect DPS3
25	DPT2	A 0 5 2	
26	DPT1	A 0 5 4	
32	E1	A 0 6 4	
33	E0	A 0 6 6	
34	S1	A 0 7 0	
35	S0	A 0 7 2	
36	EA1	A 0 7 4	
37	EA0	A 0 7 6	
42	EB1	A 1 0 4	
43	EB0	A 1 0 6	
44	EC1	A 1 1 0	
45	EC0	A 1 1 2	

Table B.4-2. Long Register Reading and Writing Codes

Register (Pins)		Register No.	Write X X X X	Read X X X X
D	1-5	020	0 2 2 0	0 4 2 0
D	16-30	021	0 2 2 1	0 4 2 1
D	31-45	022	0 2 2 2	0 4 2 2
D	46-60	023	0 2 2 3	0 4 2 3
D	61-75	024	0 2 2 4	0 4 2 4
D	76-90	025	0 2 2 5	0 4 2 5
D	91-105	026	0 2 2 6	0 4 2 6
D	106-120	027	0 2 2 7	0 4 2 7
D	226-240	037	0 2 3 7	0 4 3 7
M	1-15	040	0 2 4 0	0 4 4 0
M	16-30	041	0 2 4 1	0 4 4 1
M	31-45	042	0 2 4 2	0 4 4 2
M	46-60	043	0 2 4 3	0 4 4 3
M	61-75	044	0 2 4 4	0 4 4 4
M	76-90	045	0 2 4 5	0 4 4 5
M	91-105	046	0 2 4 6	0 4 4 6
M	106-120	047	0 2 4 7	0 4 4 7
M	226-240	057	0 2 5 7	0 4 5 7
F	1-15	060	0 2 6 0	0 4 6 0
F	16-30	061	0 2 6 1	0 4 6 1
F	31-45	062	0 2 6 2	0 4 6 2
F	46-60	063	0 2 6 3	0 4 6 3
F	61-75	064	0 2 6 4	0 4 6 4
F	76-90	065	0 2 6 5	0 4 6 5
F	91-105	066	0 2 6 6	0 4 6 6
F	106-120	067	0 2 6 7	0 4 6 7
F	226-240	077	0 2 7 7	0 4 7 7
S	1-15	100	0 3 0 0	0 5 0 0
S	16-30	101	0 3 0 1	0 5 0 1
S	31-45	102	0 3 0 2	0 5 0 2
S	46-60	103	0 3 0 3	0 5 0 3
S	61-75	104	0 3 0 4	0 5 0 4
S	76-90	105	0 3 0 5	0 5 0 5
S	91-105	106	0 3 0 6	0 5 0 6
S	106-120	107	0 3 0 7	0 5 0 7
S	226-240	117	0 3 1 7	0 5 1 7

Table B.4-2. Long Register Reading and Writing Codes (Con'd)

Register (Pins)	Register No.	Write X X X X	Read X X X X
C 1-15	120	0 3 2 0	0 5 2 0
C 16-30	121	0 3 2 1	0 5 2 1
C 31-45	122	0 3 2 2	0 5 2 2
C 46-60	123	0 3 2 3	0 5 2 3
C 61-75	124	0 3 2 4	0 5 2 4
C 76-90	125	0 3 2 5	0 5 2 5
C 91-105	126	0 3 2 6	0 5 2 6
C 106-120	127	0 3 2 7	0 5 2 7
C 226-240	137	0 3 3 7	0 5 3 7
R 1-15	140	0 3 4 0	0 5 4 0
R 16-30	141	0 3 4 1	0 5 4 1
R 31-45	142	0 3 4 2	0 5 4 2
R 46-60	143	0 3 4 3	0 5 4 3
R 61-75	144	0 3 4 4	0 5 4 4
R 76-90	145	0 3 4 5	0 5 4 5
R 91-105	146	0 3 4 6	0 5 4 6
R 106-120	147	0 3 4 7	0 5 4 7
R 226-240	157	0 3 5 7	0 5 5 7
Pin Address	160	0 3 6 0	0 5 6 0
Socket ID	161	0 3 6 1	0 5 6 1
Statement Number Display	162	0 3 6 2	0 5 6 2
Clock and Strobe	163	0 3 6 3	0 5 6 3
Precision Power Source	164	0 3 6 4	0 5 6 4
Precision Sense Level	165	0 3 6 5	0 5 6 5
External Interface Register	166	0 3 6 6	0 5 6 6

APPENDIX C
VOLTAGE AND CURRENT RANGE DEFINITIONS

MODULE	STATEMENT	PROGRAMMABLE VALUE/RESOLUTION			
		RANGE 0	RANGE 1	RANGE 2	RANGE 3
PMU	Force Voltage	-	0 to 1.023v/1 mv	0 to ±10.23V/10mV	0 to ±40.92V/40mV
PMU	Set PMU ForceV	-	0 to 1.023v/ 1 mv	0 to ±10.23V/10mV	0 to ±40.92V/40mV
PMU	Force Current	0 to ±1.023μA/1nA	0 to ±.1023mA/.1μA	0 to ±10.23mA/10μA	0 to ±102.3mA/.1mA
PMU	Set PMU ForceI	0 to ±1.023 A/1nA	0 to ±.1023mA/1μA	0 to ±10.23mA/10μA	0 to ±102.3mA/.1mA
PMU	Set PMU Sense Voltage	-	0 to ±1.023V/1mV	0 to ±10.23V/10mV	0 to ±40.92V/40mV
PMU	Set PMU Sense Current	0 to ±1.023μA/1nA	0 to ±.1023mA.1μA	0 to ±10.23mA/10μA	0 to ±102.3mA/.1mA
DPS	Force VF	-	-	0 to ±10.23V/10mV	0 to ±40.92V/40mV
DPS	Force IF	-	-	0 to ±102.3mA/.1mA	0 to ±1.023A/1mA
DPS	Enable Trip	-	-	0 to ±102.3mA/.1mA	0 to ±1.023A/1mA
DPS	Enable TripV	-	-	0 to ±10.23V/10mV	0 to ±40.92V/40mV
RVS	Set (S0/S1)	-	-	0 to ±10.23V/10mV	0 to ±30.00V/40mV
RVS	Force E	-	-	0 to ±10.23V/10mV	0 to ±30.00V/40mV
Software	Enable (ILO/IHI)	0 to ±1.023μA/1nA	0 to ±.1023mA/.1 A	0 to ±10.23mA/10μA	0 to ±102.3mA/.1mA
Software	Enable (VLO/VHI)	-	-	0 to ±10.23V/10mV	0 to ±40.92V/40mV

APPENDIX D
TIME DELAY DEPENDENT STATEMENTS

TIME DELAY DEPENDENT STATEMENT	TIME DELAYS GENERATED (msec.)	
CPMU PIN	SET (S1/SØ)	0.54
ENABLE TRIP	SET PMU	1.75
ENABLE TRIPV	SET CLAMP	1.75
FORCE CLOCK	SET (D/R)	0.54
FORCE CURRENT	ENABLE TRIP	1.75
FORCE PMU	ENABLE TRIPV	1.75
FORCE STROBE	ENABLE RELAY	0.54
FORCE VOLTAGE	FORCE (E1/EØ/...)	0.54
MEASURE VALUE	FORCE VF	1.75
SET DELAY	FORCE IF	1.75
SET D	FORCE (VOLTAGE/CURRENT)	1.75/0.54*
SET F	FORCE DELAY	**
SET M		
SET R		
SET S		
SET PMU		
XCON		
XPMU PIN		
FORCE WAIT		

*1.75 msec with Range Change--0.54 msec with no range change or DC time delay, whichever is longer.

**DC time delay.

APPENDIX E
READ/WRITE MAGNETIC TAPE STATEMENTS

E.1 DEFINITION

The FACTOR READ (MTR) and WRITE (MTW) statements are defined as follows:

```
READ (MTR) "name" V1, V2, V3, V4;

WRITE (MTW) "name" V1, V2, V3, V4;
```

The terms V1 through V4 represent array identifiers which have been declared prior to executing the READ/WRITE statements. There may be one to four arrays per statement. The term "name", enclosed by double quotations specifies the file name of the data to be written on magnetic tape.

Execution of the WRITE (MTW) statement causes the Array Data Segment(s) to be written on magnetic tape at the tape's current position. Figure E.1 gives the format specification of an Array Data Segment.

An EOF (End of File) tape mark is written under the following conditions:

- When End of Test occurs and the tester is in automatic mode, and at least one WRITE (MTW) statement has been executed.
- At the completion of each WRITE (MTW) statement when the tester is in manual mode.
- When the tester pauses as the result of a TOPSY "PAUSE" command or a FACTOR "PAUSE" and at least one WRITE (MTW) statement has been executed.

Only one magnetic tape unit may be used with the SENTRY-400 even though the system may have more than one test station. Any of the four stations which execute programs containing READ (MTR) and/or WRITE (MTW) statements will have access to the magnetic tape unit. To avoid having read/write conflicts which could destroy valid data, only one station of a multiple station system should execute programs which utilize magnetic tape.

FIGURE E.1-1

ARRAY DATA SEGMENT

Physical Record Number	Word Number	Contents
1	1	8 character TRASCII code word
	2	for the file "name".
	3	Data record length = N (integer)
	4	0
	5	0
	6	0
2	1	Words 1 to N are the contents of
	2	one variable length FACTOR array.
	.	(FST-1 floating point)
	.	
	.	The maximum number of words per
		each record is limited to 512,
2	N	(but must not be fewer than 7).

E.2 READ ERRORS

E.2.1 Array Element Count Error

If the word count of the tape data exceeds the number of elements in the specified array(s) or if the declared array has less than seven (7) elements, the system issues terminal error 40. If the array size is less than 7 elements, the tape is not advanced. When the tape data word count exceeds the array size, the tape will have advanced to the end of the excessive tape segment prior to accepting the next station "START".

E.2.2 Data Transfer Error

If a data transfer error is detected, terminal error 31 is issued and the tape is positioned forward to the beginning of the next file. The TOPSY program statement counter is reset such that when station "START" is depressed, the loaded program will begin execution at statement one (1).

E.2.3 End of Tape Error

If the End Of Tape (EOT) mark is encountered before the specified segment is found, the tape is rewound to the Beginning Of Tape (BOT) mark and terminal error 36 is issued. The TOPSY program statement counter is reset such that when station "START" is depressed, the loaded program will begin execution at statement one (1).

E.2.4 Memory Protect

If the memory protect switch located on the tape controller is enabled, the system issues terminal error 37 and the TOPSY program counter is reset such that when station "START" is depressed, the loaded program will begin execution at statement one (1).

E.3 WRITE ERRORS

E.3.1 Data Transfer Error

If a data transfer error is detected, terminal error 33 is issued and the tape is positioned backwards to the start of the current file. The TOPSY program counter is reset so that when station "START" is depressed, the loaded program will begin execution at statement one (1).

E.3.2 End of Tape Error

If the End OF Tape (EOT) mark is encountered prior to completion of a WRITE operation, the tape is rewound and the system issues terminal error 35. The unit is unloaded so that it cannot be restarted by pushing station "START". This avoids accidental writing over good data at the beginning of the tape.

E.3.3 Array Element Count Error

If an array of size less than seven (7) appears in the WRITE statement, terminal error 40 is issued, see below (E.3.4).

E.3.4 Unrecoverable Errors

Any errors other than those described above are considered to be unrecoverable and the system issues terminal error 40. The TOPSY program statement counter is reset.

E.4 STANDARD MAG TAPE OPERATION IN TOPSY

E.4.1 Before executing a program employing mag tape read or write statements, the operator must set the tape at the BOT marker of the tape file the program is to read or write.

E.4.2 The instructions relating to the periodic maintenance of the mag tape should be attended to if error free operation is desired.

E.4.3 Before executing the TOPSY program, the REMOTE switch on the mag tape unit must be enabled.

E.4.4 After steps E.4.1 through E.4.3 it is only necessary to execute the TOPSY program from the tester station. All mag tape controls are performed by TOPSY.

E.5 UNUSUAL MAG TAPE OPERATION IN TOPSY

E.5.1 Catastrophic Errors

If a 'catastrophic' error occurs during mag tape operation and the user desires to make some attempts to recover then the following course of action is recommended as a desparation procedure.

E.5.1.1 Write Operation

Go back to DOPSY manually and execute two tape mark writes, viz:

```
// MTAP TMARK   (twice),
```

followed by:

```
// MTAP SKIP BACK 1 RECS
```

E.5.1.2 Read Operation

Go back to DOPSY manually. Rewind the tape via the tape transport REWIND switch and restart TOPSY.

E.5.1.3 Warning

The user should be aware that these recovery actions bypass the normal TOPSY-DOPSY return and, as consequence, do not update the present state of TOPSY. When TOPSY is reentered, it is initialized to the state prior to the last return to DOPSY.

APPENDIX F

SUMMARY OF TOPSY ERROR MESSAGES

Error Number	Meaning
1	Program Not Loaded
2	Station Disabled (Power Off)
3	Magnitude or Polarity Error in Pin Number, Clock Count, or Time Delay
5	Magnitude Error in Voltage or Current (Exceeds Hardware Limitations)
21	Current Value Not Within Set Limits
22	Voltage Value Not Within Set Limits
23	Improper Pin Address
24	Voltage Value Exceeds 30 Volts
25	Wrong Socket Address on Load Board
26	Undefined OP Code
31	READ (File Skip Forward Executed)
33	WRITE (File Skip Backward Executed)
35	EOT Tape on Write (Catastrophic)
36	EOT Tape on Read
37	Memory Protect on Tape Read
40	Data Count Error Less Than 7 or Greater Than Assigned Array
42	Unrecoverable Error
50	Improper Vector Declaration
51	The Number of Formal and Actual Parameters Do Not Agree
52	Subscript Violation
53	Empty Stack
54	Program Too Big
55	EOF on Test Program
56	Illegal OP Code
57	Improper Vector Initialization
58	I/O Error
59	Improper FOR Loop Constants

On Terminal Error 31, the tape is moved to the next tape file.
On Terminal Error 33, the tape is moved back to the start of the last file.
When START is pressed, the program will continue execution from these tape locations.

CALIBRATION RESISTANCE	VOLTAGE RANGE	CURRENT RANGE
100Ω .02%	RANGE 2 10V FS	RANGE 3 100ma FS
1KΩ .01%	RANGE 2 10V FS	RANGE 2 10ma FS
100KΩ .01%	RANGE 2 10V FS	RANGE 1 0.1ma FS
10MΩ .02%	RANGE 2 10V FS	RANGE 0 1μa FS
400Ω .05%	RANGE 3 40V FS	RANGE 3 100ma FS
4KΩ .01%	RANGE 3 40V FS	RANGE 2 10ma FS
400KΩ .1%	RANGE 3 40V FS	RANGE 1 0.1ma FS
40MΩ .1%	RANGE 3 40V FS	RANGE 0 1μa FS
100Ω .02%	RANGE 1 1V FS	RANGE 2 10ma FS
10Ω .03%	RANGE 1 1V FS	RANGE 3 100ma FS

FS = Full Scale

APPENDIX G
CALIBRATION RESISTOR TABLE

NODE NUMBER			MEASURED PARAMETER
DECIMAL	OCTAL	NAME	DESCRIPTION
128	200	S1	COMPARATOR S1 REF. VOLTAGE
129	201	SO	COMPARATOR SO REF. VOLTAGE
130	202	E1	FORCING LEVEL E1 REF. VOLTAGE
131	203	EO	FORCING LEVEL EO REF. VOLTAGE
132	204	EA1	FORCING LEVEL EA1 REF. VOLTAGE
133	205	EAO	FORCING LEVEL EAO REF. VOLTAGE
134	206	EB1	FORCING LEVEL EB1 REF. VOLTAGE
135	207	EBO	FORCING LEVEL EBO REF. VOLTAGE
136	210	EC1	FORCING LEVEL EC1 REF. VOLTAGE
137	211	ECO	FORCING LEVEL ECO REF. VOLTAGE
140	214	VF1	VOLTAGE FORCING UNIT 1 OUTPUT VOLTAGE
141	215	VF2	VOLTAGE FORCING UNIT 2 OUTPUT VOLTAGE
142	216	VF3	VOLTAGE FORCING UNIT 3 OUTPUT VOLTAGE
143	217	TRIP1	VF1 LOAD CURRENT
144	220	TRIP2	VF2 LOAD CURRENT
145	221	TRIP3	VF3 LOAD CURRENT
254	376	CALOP	PMU-NO LOAD NODE
255	377	CAL	PMU CALIBRATION NODE*

APPENDIX H

INTERNAL NODES

* LOAD RESISTANCE = VOLTAGE RANGE/CURRENT RANGE

APPENDIX I

FACTOR SPECIFICATIONS

1) Maximum Number of Tests: that may be labeled
65535 ~~131071~~$_{10}$ tests.
(IND counter allows up to 177777B tests).

2) Maximum Number of Blocks per Program
Block 0 : used by the system
7 Blocks: Blocks 1-7 for users

3) Maximum Number of Variables per Block
Block 0 : 119 variables
Blocks 1-7: 127 variables per block