RCA
Information
Systems

SPECTRA 70

TIME SHARING OPERATING SYSTEM (TSOS)

Interactive FORTRAN System Reference Manual

70-00-618
January 1969

First Printing: January 1969

FOREWORD

♦ This manual is a programmer's reference document for the Extended Interactive FORTRAN language that is used in writing programs for the TSOS (Time Sharing Operating System) Interactive FORTRAN System.

The Extended Interactive FORTRAN language consists of the following three subsets:

1. FORTRAN IV language (modified)
2. Editing statements
3. Debugging statements

Section 1 describes the objectives of the TSOS Interactive FORTRAN system and the facilities and services provided. Also included are the notation conventions used throughout this manual.

Sections 2, 3, and 4 describe the Extended Interactive FORTRAN language.

Section 2 describes the TSOS FORTRAN IV language that is documented in the TOS/TDOS FORTRAN IV Reference Manual (70-00-604). This section discusses the differences, in statement format and continuation conventions, between these two languages.

Section 3 describes the editing statements available with the system.

Section 4 describes the debugging statements available with the system.

Special programming considerations and examples are included throughout the manual.

CONTENTS

CONTENTS
(Cont'd)

1. GENERAL DESCRIPTION

FEATURES

♦ The TSOS Interactive FORTRAN System combines the features of a conversational FORTRAN compiler, a special FORTRAN text-editing facility, debugging statements, and an immediate execution capability to provide the FORTRAN user of the 70/46 time sharing system with a comprehensive program preparation tool.

The system consists of a single interpreter which accepts FORTRAN programs written using the full FORTRAN IV language, the same language accepted by the TSOS FORTRAN IV background compiler. The system provides to the user the following functions:

1. Complete syntax checking of the FORTRAN IV language.
2. Comprehensive text editing of FORTRAN programs, which takes into account the structure of the FORTRAN language.
3. Comprehensive debugging statements that include facilities for displaying values, checkpointing, and controlled interrupts.
4. A facility for immediate execution for evaluating FORTRAN expressions entered from a remote terminal.

These facilities are obtained by combining the editing statements, the debugging statements, and the FORTRAN language itself into an extended language which is accepted by the interactive FORTRAN system. A user normally constructs, syntax checks, modifies, tests, and debugs his FORTRAN program, all in the interactive mode, and subsequently compiles it using the FORTRAN IV batch compiler.

THE EXTENDED INTERACTIVE LANGUAGE

♦ The extended language of the TSOS Interactive FORTRAN System consists of three components:

1. The FORTRAN language
2. The special FORTRAN editing statements
3. The debugging statements

The system is designed in such a way that editing statements and debugging statements may be embedded in a FORTRAN program if desired, to form a valid interactive FORTRAN program which can then be executed by the interactive system. For processing by the FORTRAN background compiler, however, such statements must be removed, which can be done automatically on command by means of an editing statement.

The three components of the extended interactive language are described in detail in the following sections.

NOTATION CONVENTIONS

♦ The following conventions for writing the elements of the Extended Interactive FORTRAN System are a guide to the programmer in writing his own statements.

1. Words printed in all capital letters and preceded by the symbol # are reserved words. These words have preassigned meanings within the Extended Interactive FORTRAN System. In all formats, these words represent an actual occurrence of a command or of a special variable.

2. All words printed in lower-case letters represent information that must be supplied by the programmer. All lower-case words appearing in formats are defined in the context or in Appendix B.

3. Some lower-case words are hyphenated to facilitate references to them in the text; this modification does not change the syntactical definition of the words.

4. Hyphens appearing outside of syntactical unit definitions serve as indicators of complementation or range. Their use is described in detail for each command where they appear.

5. Square brackets [ ] indicate that the enclosed item may be used or omitted, depending on the requirements of the particular program. When two or more items are stacked within brackets, one or none of them may occur.

6. Braces { } enclosing vertically stacked items indicate that one of the enclosed items is obligatory.

7. The ellipsis (. . .) indicates that the immediately preceding unit may occur once, or any number of times in succession. A *unit* means either a single lower-case word or a group of lower-case words and one or more reserved words enclosed in brackets or braces. If a term is enclosed in brackets or braces, the entire unit of which it is a part must be repeated when repetition s specified.

8. Comments, restrictions, and commentary on the use and meaning of every format are contained in the appropriate sections of the text.

9. The delimiters ⟨ ⟩ are used to enclose the name of a syntactic unit in the description of the syntax of the editing and debugging statements.

10. All other punctuation and special characters, except those mentioned above, represent the actual occurrence of those characters. Punctuation is necessary where it is shown.

11. In the examples, a box is drawn around information typed by the system to distinguish it from that supplied by the user.

2. TSOS FORTRAN IV LANGUAGE

INTRODUCTION

♦ The TSOS FORTRAN IV language is the same as that accepted by the TSOS FORTRAN IV background compiler, described in detail in TOS/TDOS FORTRAN IV Reference Manual 70-00-604. Two differences are: statement format, and continuation conventions.

The format of statements written in the TSOS Interactive FORTRAN System is completely free-form, ignoring the column conventions of standard FORTRAN systems. The sole requirement concerns the character position immediately following the system line number. In the case of a FORTRAN comment, a "C" must be typed in that position; for any other statement, it must be other than "C". When the statement is entered into the system from the terminal, it is automatically reformatted according to standard FORTRAN column conventions, so that in format it is immediately ready for compilation by the background compiler.

The continuation conventions of the Interactive FORTRAN System require that following the last character of the line to be continued, the user type a carriage return (the RETURN button on the teletype). The system then responds on the next and all subsequent continuation lines of that statement with the character "-" (hyphen). It will be typed by the system at the beginning of the line. The line number is typed following the hyphen. When such a continuation line is entered into the system, it will be converted to conform to the standard FORTRAN continuation convention in order that the continued statement be acceptable to the background compiler.

Because of the interactive nature of the system, the FORTRAN processor accepts statements in a prescribed order. So as not to burden the user of the system with the necessity of arranging his program in the required way, an editing command, #ORDER, is included in the system. This command automatically reorders the statements in the prescribed way. However, by writing his program in the required order, the user simplifies the processing of his program and realizes time savings in its execution. The following types of statements must appear in the order given for a program in the TSOS Interactive FORTRAN System:

1. FUNCTION, SUBROUTINE, #DEFINITION, BLOCK DATA, or PROGRAM
2. IMPLICIT
3. ABNORMAL
4. EXTERNAL
5. Explicit type statements
6. DIMENSION
7. COMMON

INTRODUCTION (Cont'd)

8. EQUIVALENCE
9. DATA
10. #PATTERN
11. Statement function definitions
12. Any other statements
13. END

PROGRAMS

♦ A file may contain one or more programs or it may contain data. Each program in a file must be identified by an initial PROGRAM, SUBROUTINE, FUNCTION, #DEFINITION, or BLOCK DATA statement. The retrieval of a file of programs causes the definition of the names of the programs it contains as the names of the lists of all the statements which they each contain (except for the END statements). In general, a program name may be used in editing statements to designate the list of statements that it contains.

If more than one file is in virtual memory and programs with the same name occur in different files, then in editing statements a unique program may be designated by qualifying the program as ⟨file⟩.⟨prog⟩.

STATEMENTS

♦ A statement consists of an initial line and possibly one or more continuation lines. Each line (including continuation lines) must have a line number. In editing statements, a statement may be designated by the line number of the first line of the statement, by its FORTRAN statement number, or by its contents.

Line numbers will be of the form

L.N

where L and N represent a one or four decimal digit number. Leading zeroes in L and trailing zeroes in N will not be typed by the system unless explicitly requested by the user. See below.

Every line number that is typed by the user to indicate a place in a file (at which new text is to be inserted) implies that additional lines of input will follow, that the line number will be incremented, and what the increment is. The implicit increment will be a 1 in the least significant digit typed. Thus, in 101. the increment is 1.; in 4.07, .01; and in 2.030, .001.

The implied increment may always be superseded by explicitly giving the desired one. An explicit increment follows the line number and is enclosed in parentheses. It has the same form as the line number itself, and need not be a fractional number. The number of N digits typed by the system will be the maximum of the number of N digits in the line number and the increment, if given.

STATEMENTS (Cont'd)

Examples:

Typed by User	Increment	Actual or Implied	Line Numbers Typed by System
20.01	.01	I	20.01,20.02, . . .
3.042 (.002)	.002	A	3.042,3.044, . . .
4.713	.03	A	4.713,4.743,4.773, . . .
5.1 (2.01)	2.01	A	5.10,7.11,9.12, . . .

MODES

♦ The system is assumed to be always in one of two modes: Command or Text. In the Command mode, which is equivalent to the Desk Calculator mode, lines entered are extended FORTRAN statements. They are executed immediately, and, unless the instruction explicitly alters the mode, a new Command mode line is required as the next line.

In the Text mode, the system is expecting a line of text to be entered into a file. This fact is indicated by the system typing a line number. This line number becomes the line of text's number in the file if a line of text is entered. If a line of text is not entered, the typed line number is ignored and no change is made to the file because of it.

ENTERING TEXT FROM A TERMINAL

♦ Whenever a user requests to insert new text into a file (whether it is a new or existing file), the system attempts to preset the line number and increment for him.

In the case of a new file, after the system responds with

FILE IS EMPTY

the line number and increment are set to 1, the mode set to text, and the line number (1.) typed in anticipation of the first line.

If a place is specified between two lines of an already existing file, the message

BETWEEN LINES ⟨line 1⟩ AND ⟨line 2⟩

is typed and the line number and increment computed from ⟨line 1⟩ and ⟨line 2⟩, if possible. The increment will be a 1 in the least significant non-zero N digit of ⟨line 1⟩ or the whole number 1, if all N digits are zero. If this increment is such that ⟨line 1⟩ + ⟨increment⟩ ≥ ⟨line 2⟩, the increment will be shifted one place right and tested again. The line number typed by the system in expectation of the first line of new text is then computed to be ⟨line 1⟩ + ⟨increment⟩.

If no increment can be found, no line number is typed but an appropriate message is typed.

ENTERING TEXT FROM A TERMINAL (Cont'd)

Examples

Line 1	Line 2	Increment	Typed Line Number
2.0000	2.0100	.001	2.001
2.	3.	.1	2.1
4.999	4.9992	.0001	4.9991
7.9999	8.	——	——

If new text is to be inserted at the end of an existing file, the message

LAST LINE IS ⟨line 1⟩

is typed. The increment is a "1" in the least significant non-zero N digit, or a "1." if all N digits are zero and if the beginning line number is ⟨line 1⟩ + ⟨increment⟩.

LINE NUMBER COMMANDS

◆ Whenever text is being inserted from a remote terminal, several commands are available for altering the sequence of statements stored into the file. These include the ability: (1) to change the current line number and increment; and (2) to temporarily change it, but have it remembered so that it may be returned to later. This second capability permits one to go back and insert text between two previously entered lines and then resume at the old place with a minimum of line number typing.

All commands which pertain to line numbers are identified by an "@" symbol. The permissible commands are shown below, and are followed by an explanation of their meaning and use.

1. @SET ⟨line-no⟩ [(⟨increment⟩)]

   Examples:

   @SET 1.79
   @SET 1.24(.01)
   @SET 100(10)

2. @⟨line-no⟩ [(⟨increment⟩)]

   Examples:

   @100(10)
   @100

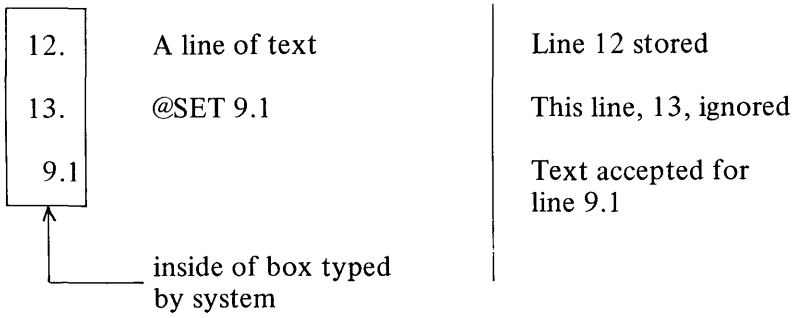
3. @

Command 1: @SET

♦ This command specifies that the current line number and increment are to be forgotten, being replaced by the specified line number and increment, and that new text is to be entered at this new place. No text is typed on this line, because the system will type back the new line number before accepting the text for the line.

When this command is given in place of a line of text (i.e., on a line for which the system has typed a line number), it is not taken as text, the line number of this line is ignored, and the new line number is typed in expectation of a line of new text.

Example:



This command is valid only if an insert is being made to a file and if a line number has been typed out by the system.

Command 2: @n

♦ This instruction specifies that the current line number and increment are to be saved (pushed) in a push-down stack for later retrieval, and then replaced by the specified line number and increment. As with the @SET command, the line on which it is typed is not entered as text, and an extra line is skipped before the number is typed.

This command is used in conjunction with command 3, and examples follow its description.

Command 3: @

♦ When typed in place of a line of text, this command specifies that the current line number and increment are to be replaced by the most recently pushed line number and increment, and the stack "popped". The stack has a maximum depth of 3, and if an attempt is made to pop it more times than it has been pushed, an error message is given, no action is taken, and the current line number is preserved and then retyped for the next line of text. It should be noted, however, that after the stack has been pushed the fourth time it becomes circular and the fourth pushed line number replaces the first. In this case, it can be popped indefinitely, with the line numbers repeating every fourth time, and no error is detected.

Command 3:

@

(Cont'd)

Examples:

```
.
.
.
71.     'text'      Text entered as line 71
72.     @9.1        Save line number 72, Go to 9.1
                        } Skipped line
 9.1    'text'      Text for line 9.1
 9.2    'text'      Text for line 9.2
 9.3    @7.01       Save 9.3, Set to 7.01
                        } Skipped line
 7.01   'text'      Text for line 7.01
 7.02   'text'
 7.03   @           Return to last previously saved line.
                        } Skipped line
 9.3    'text'      Text for line 9.3
 9.4    'text'      Text for line 9.4
 9.5    @SET22.03(.02)  Start entering at line 22.03
                        } Skipped line
22.03   'text'      Text for line 22.03
22.05   'text'      Text for line 22.05
22.07   @           Return to first previously saved line.
                        } Skipped line
72.     'text'      Back to original line number. Text
                    for line 72.
73.     'text'      Text for line 73
74.     @           Attempt to perform procedure too
                    many times.
TOO MANY POPS
74.                 Text mode continues
```

The @n procedure can be used in place of the @SET command, simply by never issuing the corresponding @.

IMMEDIATE EXECUTION

♦ A statement typed in with no line number is a request for immediate execution of that statement.

Example

```
10.0    PROGRAM NASA003
10.1    DIMENSION X(30)
10.2    ETX     < NULL>
*       Y = SIN (22,379)
*       Z = 3.14 * Y
*       #DISPLAY (/NASA003/Y,Z)
```

Because there is no line number on the statements following 10.2, they would be executed immediately. As a result, their values are displayed at the terminal.

3. EDITING STATEMENTS

◆ The types of entities processed by the FORTRAN editing statements are files, programs, statements and lists of consecutive statements, syllables (elementary constituents of FORTRAN statements), lists of consecutive syllables, and strings of consecutive characters. Each of these is discussed separately below.

FILES

◆ Before a file can be executed or manipulated in any way by the interpretive system, it must be retrieved from the disc or typed in at the terminal. On the disc, a file is an indexed sequential file, sorted by line number. The same conventions are used for naming files as are used throughout the TSOS System. A file is never cataloged or stored back on the disc unless an explicit command requesting such storage (#SAVE) is executed. Otherwise, the file remains in the user's virtual memory until the end of the session. In general, a file name may be used in editing statements to designate the list of statements which it contains, and the retrieval of a file causes the definition of the file name as the name of that list of statements.

In certain of the editing and debugging statements, in which the name of a file is required and is not specified, one of two actions will take place:

1. If there is only one file in virtual memory at the time, that file (referred to in the sequel as the "only" file) is referenced.

2. If there is no file in virtual memory at the time, or if there is more than one file, omitting a file reference causes creation of a new file (or reference to it, if it has been previously created in this way). This file is referred to in the sequel as the "unnamed" file, and is a temporary or work file which is not saved at the end of the user's session.

Designating Statements and Places

◆ In some editing statements, it is necessary to identify a point within a file where information is to be inserted. This is always identified as *before* or *after* a statement or list of statements. In the description of the syntax of the editing statements the use of a designator of a statement or list of statements in this way is called a ⟨place⟩, which may take the following forms:

1. ⟨list-name⟩

2. ⟨list-name⟩ (⟨end-pt⟩)

3. (⟨end-pt⟩)

The ⟨list-name⟩ may be a file name, program name, or the name of a list of statements defined by a #NAME statement. The ⟨end-pt⟩ selects a single statement out of the list; its forms are:

1. ⟨line-no⟩

2. &⟨stmt-no⟩

3. ⟨list-name⟩

4. #S⟨integer⟩ [(⟨exp⟩)]

Designating Statements and Places *(Cont'd)*

where:

⟨line-no⟩ represents the line number of the first line of a statement and is an integer (1-4 digits) possibly followed by a decimal point followed by a fractional part (0-4 digits).

⟨stmt-no⟩ represents a (1-5 digit) statement number; and the form 4 denotes a statement located previously by a #FIND, #EDIT, or #PATTERN statement (described below; see also the section on the "Lists of Syllables" below). Each form must represent a statement which is actually present in the list being qualified.

LISTS OF CONSECUTIVE STATEMENTS

◆ A set of consecutive statements of a program may be treated as a list by the editing statements and designated by a name (with the #NAME statement). Such a named list of statements may still retain its position in a program or in another list.

A sublist of a list of statements may be designated by specifying the first and last statements in the sublist; and these, in turn, may be designated by their initial line numbers, by their statement numbers, or by specifying a list containing them.

More exactly, a ⟨stmt-list⟩, used in editing statements to designate a list of statements, has the following forms:

1. ⟨list-name⟩
2. ⟨list-name⟩ (⟨range⟩)
3. (⟨range⟩)

where:

⟨range⟩ serves to extract a list of consecutive statements from the ⟨list-name⟩ (the unnamed or only file in case 3) and has one of the following forms:

1. ⟨end-pt⟩
2. ⟨end-pt⟩ – ⟨end-pt⟩
3. ⟨end-pt⟩ –
4. – ⟨end-pt⟩

Form 3 means: "from the beginning of ⟨end-pt⟩ through the end of the ⟨list-name⟩". Form 4 means: "from the beginning of ⟨list-name⟩ through the end of ⟨end-pt⟩"

STATEMENT SETS

♦ Many of the editing statements operate on specified sets of statements. If these are the entire contents of some named list, they may be referred to by the list name alone. However, ranges of statements within a named list may be specified by giving the line numbers, statement numbers, or sublist names of the beginning and end of each range.

Specifically, such a reference will be called a ⟨stmt-set⟩ and has the following possible forms:

1. ⟨list-name⟩
2. ⟨list-name⟩ (⟨range⟩ , ⟨range⟩, . . .)
3. ⟨list-name⟩ – (⟨range⟩ , ⟨range⟩, . . .)
4. (⟨range⟩, ⟨range⟩, . . .)
5. –(⟨range⟩, ⟨range⟩, . . .)

In case 2, the ⟨range⟩'s here are selected from the ⟨list-name⟩. In case 3, the complement of the ⟨range⟩'s is selected from the ⟨list-name⟩. Cases 4 and 5 are like 2 and 3, except that the single unnamed file (or the only file in virtual memory) is being designated.

Examples of ⟨stmt-set⟩'s

ARCSIN
MULT (101)
MULT- (101-110.3)
HARP (–&15, &20, &30, &40–)
PROGA (10–LISTA)
PROGB (LISTB–)
PROG (#S2–)
– (&20–&40)

The mere mentioning of a statement set does not affect its existence in virtual memory nor change the relationships which the set or any pa. the set bears to any existing file, program, or list of statements.

SYLLABLES

♦ When analyzed, a statement is broken down into *syllables*, the smallest syntactic units which are recognized by the lexical scanner. For example, the following are syllables:

1. Names
2. Numbers
3. Statement Numbers
4. Arithmetic or Logical Operators
5. Relations
6. Punctuation
7. Statement Keywords: GO TO, READ, etc.

SYLLABLES (Cont'd)

Some editing statements permit one to search within statements for patterns composed of syllables and lists of syllables. For this purpose, a number of elementary search procedures are provided to specify specific syllables or certain classes of syllables. Having located a syllable or a list of syllables within a given statement, it may be named, copied, moved, deleted, etc. by the syllable editing statements. The syllable editing statements (#FIND and syllable list assignment statements) are distinct from the editing statements used for statements and lists of statements.

LISTS OF SYLLABLES

◆ A sublist of consecutive syllables within a statement may be located by the #FIND statement. In order to be able to distinguish the names for these sublists from other names with which they may be combined, the *special names* #1, #2, #3, . . . are used.

The special names #S1, #S2, #S3, . . . are used to designate the statements containing the sublists #1, #2, #3, . . . , respectively.

Notice that the syllable lists #1, #2, . . . have definite positions within the specific statements #S1, #S2, . . . which, in turn, represent specific statements of some file or program.

RELATIVE SYLLABLE AND STATEMENT NOTATION

◆ The syllable which is a certain number of syllables after (or before) the syllable list #n may be designated by the notation

#n(e)

Here N is a positive integer and e is any integer FORTRAN expression. If the value of e is positive, say 7, then #n(7) represents the 7^{th} syllable after the syllable list #n. If the value of e is negative, say 7, then #n(7) represents the 7^{th} syllable before the syllable list #n. The syllable #n(e) has a definite position within a statement. If the value of e is zero, then #n(e) represents the syllable list #n.

Similarly, the statement which is a certain number of statements after (or before) the statement #Sn may be designated by the notation #Sn(e), where n is again a positive integer and e is an integer-valued FORTRAN expression. If the absolute value of the expression e is 7, then #Sn(e) represents the 7^{th} statement after the statement #Sn, or the 7^{th} statement before #Sn, according to whether the value of e is positive or negative, respectively. If the value of e is zero, then #Sn(e) represents the statement #Sn.

Examples of Relative Syllable Notation

```
#1  (1)      #3(I**2+6)
#2  (-1)     #62(-J+K-2)
```

Examples of Relative Statement Notation

```
#S1 (1)      #S3(J)
#S2 (-1)     #S4(-K+J*2)
```

SPECIAL SYLLABLE LISTS

◆ The special names #1L, #2L, . . . are used to designate special lists of syllables or characters which are never sublists. That is, they are never part of a statement or program. They are like list accumulators and are used for intermediate list computations.

SEARCH EXPRESSIONS

Elementary Search Expressions

◆ Certain strings of concatenated syllables (constants, identifiers, operators, and punctuation characters) form legal elementary search expressions, ⟨search-exp⟩. These legal search expressions include the following:

1. Any legal FORTRAN arithmetic or logical expression.
2. Any legal assignment statement or arithmetic statement function definition.
3. Any arithmetic or logical operator or relation.
4. Any actual argument or actual argument list with or without the enclosing parentheses.

Except within literal constants, blanks are ignored in these elementary ⟨search-exp⟩'s.

These elementary search expressions are matched with lists of syllables (rather than with strings of characters) in the statements being searched. Therefore, not all occurrences of the same set of characters will be regarded as a match. For example, an asterisk represents the operation of multiplication and thus an asterisk in a search expression will not match the asterisk in the following statement:

```
REAL*4 I
```

Table 3-1 shows a list of various search syllables and their respective matching syllables.

Table 3-1. Search Syllables and Respective Matching Syllables

Search Syllable	Matching Syllable
identifier	Same identifier; e.g., variable, array, function name, etc.
numerical constant	Any equal number used numerically, but not repeat count, data length, or statement number.
+ - * / **	Same symbol used as an operator.
.AND. .OR. .NOT.	Same symbol used as logical operator.
.GT. .GE. .LT. .LE. .EQ. .NE.	Same symbol used as a logical relation.
& integer	Statement number as actual argument.
literal constant	Any equal literal constant.
(period)	Any occurrence not part of constant, operator, or relation.

Elementary Search Expressions *(Cont'd)*

Table 3-1. Search Syllables and Respective Matching Syllables *(Cont'd)*

Search Syllable	Matching Syllable
:	Beginning of statement itself, after continuation column.
;	End of statement.
, ( ) @ % ? =	Any occurrence.
blank	Ignored, except in literal constants.

Any ⟨search-exp⟩ of the types shown in Table 3-1 may be enclosed in the characters <> without changing its meaning. It must be so enclosed if it is used to form a compound search expression.

Examples of Elementary Search Expressions

```
A + B              :A = B + 1;
(A+B) * 4.16
<C,10,D>           <I + 2>
```

Searches for Classes of Syllables

◆ Table 3-2 gives the format of elementary search expressions which are satisfied, in general, by classes of syllables.

Table 3-2. Elementary Search Expressions

Form of ⟨search-exp⟩	Matching Syllable
#I [⟨type⟩]	any identifier [with type ⟨type⟩]*
#V [⟨type⟩]	any variable [with type ⟨type⟩]*
#A [⟨type⟩]	any array name [with type ⟨type⟩]*
#F [⟨type⟩]	any function name [with type ⟨type⟩]*
#C [⟨type⟩]	any constant [with type ⟨type⟩]*
#S	any subroutine name
#B	any block name
#N [(e)]	any statement number [with value of expression e]
#L [(e)]	any statement label [with value of expression e]
#ANY	any string of syllables, including null string
#ANY ( )	any string of syllables, with balanced parentheses
#ST (⟨kind⟩)	any syllable in statement of kind ⟨kind⟩**
" string "	any matching character string

*See Table 3-3, Values for ⟨type⟩ in search expressions.
**See Table 3-4, Values for ⟨kind⟩ of statement.

Searches for Classes of Syllables *(Cont'd)*

Table 3-3. Values for ⟨type⟩ in Search Expressions

Type	Meaning
L	Logical
L1	Logical, 1 byte
L4	Logical, 4 byte
I	Integer
I2	Integer, 2 byte
I4	Integer, 4 byte
R	Real
R4	Real, 4 byte
R8	Real, 8 byte
C	Complex
C8	Complex, 8 byte
C16	Complex, 16 byte

Table 3-4. Values for ⟨kind⟩ of Statement

Kind	Meaning
X	Any executable statement
S	any specification statement
A	any assignment statement
C	any control statement
I	any I/O statement
O	any organizational statement
T	any explicit type statement
E	any extended language statement, editing or debugging
GO	GO TO
IF	arithmetic IF
LIF	logical IF
DO	DO
CON	CONTINUE
PAU	PAUSE
STOP	STOP
CALL	CALL
RET	RETURN
READ	READ
WRI	WRITE
EF	END FILE
BS	BACKSPACE
PUN	PUNCH
PRI	PRINT
IMP	IMPLICIT
ABN	ABNORMAL
EXT	EXTERNAL
DIM	DIMENSION
COM	COMMON
EQU	EQUIVALANCE
PRO	PROGRAM
REW	REWIND
END	END
FUN	FUNCTION
SUB	SUBROUTINE
ENT	ENTRY
BD	BLOCK DATA
DATA	DATA
NAM	NAMELIST
FOR	FORMAT

Compound Search Expressions

◆ Elementary search expressions may be combined by the following operators, which are listed in order of binding strength from strongest to weakest:

Representation	Meaning: i.e., what list satisfies compound expression (S_1 and S_2 are search expressions)
.NOT. S_1	list satisfies if it doesn't satisfy S_1
S_1 .AND. S_2	list satisfies if it satisfies S_1 and S_2
S_1 .OR. S_2	list satisfies if it satisfies S_1 or S_2
S_1 S_2	list satisfies if it consists of list satisfying S_1 followed by list satisfying S_2; i.e., concatenation.

The delimiters <> serve to denote the scope of the character string defining the search expression. In addition, parentheses may be used to group subexpressions in the normal way.

Examples

<A> .OR. <B>
<A .OR. B>
A .OR. B
<A> .AND. #AC16 .OR. <B>
#B .AND. (<XX> .OR. <YY>)
#ST(GO) .AND. #N(10) <;>

Given a portion of a program containing the statements:

```
220.  10  IF      (A .OR. B)  GO TO 17
225.  20  A=3
230.  30  B=7
235.  45  AB=14
```

The following table indicates both the line number and the statement number of those statements for which the given search expression is satisfied when using editing statements.

Search Expression	Statement Number	Line Number
<A>.OR.<B>	10, 20, 30	220, 225, 230
<A .OR. B> A .OR. B	10	220
A <A>	10, 20	220, 225
<AB>.OR.<A>	10, 20, 45	220, 225, 235
AB .OR. A	None	None

Compound Search Expressions *(Cont'd)*

All or portions of lists found by a search expression may be given one of the special list names #1, #2, . . . by enclosing that portion in parentheses and preceding it by #n: . In the absence of parentheses this "naming operator" has a scope extending up to the next concatenation operation; however, the insertion of the parentheses must not change the meaning.

Example

#2: <A> .OR. <B>

#2:(<A>).OR. <B>

} These are different.

Special List Names in Search Expressions

♦ If a special list name of the form #n or #n(e) or #nL appears in a search expression it has the same meaning as if the contents of that list surrounded by the delimiters <> were inserted in the search expression at that point. By convention, any compound search expression may be surrounded by these delimiters without changing the meaning.

Example

#1:(#V) <*>#1

This search expression will be satisfied, for example, by A*A or B*B, if A and B are simple variables.

LIST EXPRESSIONS

♦ List expressions are much simpler than search expressions. They consist of concatenations of the following types of quantities:

1. Literal strings surrounded by double quotes.
2. Special syllable list names (#n, #n(e), #nL).
3. Certain functions; namely, the functions

   #SV(e)

   #CH(m,n,'list-exp')

   #DM(e_1,#n[(e_2)])

The function #SV yields the source code representation of the value of an ordinary numerical or logical expression.

The function #CH yields the character string which is the n characters from the m^{th} through the $(m+n-1)^{th}$ character of the list expression 'list-exp'.

The function #DM(e_1,#n[(e_2)]) yields a character string which is the source code representation of the $e_1{}^{th}$ dimension of the array whose name must be the contents of #n[(e_2)].

Syllable List Assignment Statements

♦ The syllable list assignment statement provides a means of creating a syllable list, or of changing the contents of an already existing one. The syntax of the syllable list assignment statement is as follows:

$$\left\{\begin{array}{l}\#n\\ \#n(e)\\ \#nL\end{array}\right\} \quad = = \quad \left[\ {}^{\prime}\text{list-exp}{}^{\prime}\ \right]$$

where n is a positive integer and e is any FORTRAN integer expression.

The syllable list specified by the left side of the assignment statement is replaced by the list specified by the list expression, 'list-exp', on the right side. If the list expression is omitted, the syllable list is defined to be a null (empty) list; that is, the previous contents of the list are deleted.

LOGICAL EXPRESSIONS

♦ The relational operators are extended for TSOS Interactive FORTRAN to include list expressions.

Logical expressions for the TSOS Interactive FORTRAN are expressed as listed in the TOS/TDOS FORTRAN IV Manual (No. 70-00-604) with the following changes:

Relational operators combined with:

a. Arithmetic expressions whose mode is integer or real; or

b. List expressions.

Comparison of List Expressions

♦ A list expression may be compared only with another list expression. List expressions are evaluated before they are compared (see Evaluation of List Expressions, on the next page).

List Expressions of Equal Length

♦ If list expressions are of equal length, comparison proceeds by comparing characters in corresponding positions starting from the left-hand end and continuing until either a pair of unequal characters is encountered or the right-hand end of the two list expressions is reached, whichever comes first. The list expressions are determined to be equal when the right-hand end is reached without encountering any difference.

The first encountered unequal pair of characters is compared for relative position in the EBCDIC character set. The list expression that contains the character position higher in this collating sequence is determined to be the greater list expression.

List Expressions of Unequal Length

♦ If the list expressions are of unequal length, comparison proceeds as described above. If this process exhausts the characters of the shorter list expression, then that list expression is determined to be less than the longer list expression.

THE SYNTAX OF LIST EXPRESSIONS

◆ List expressions consist of concatenations of the following types of quantities:

1. Literal strings surrounded by quotation marks (")
2. Syllable sublist names (#n[(e)])
3. Independent syllable list names (#nL)
4. The following functions:

a.	#SV(e)	source value
b.	#DM(e_1, #n [(e_2)])	dimension
c.	#CH(m,n, ⌜list-exp⌝)	characters
d.	#PB(#n [(e)]) or #PB (#nL)	preserve blanks

In case the list expression appears after a == operator or in response to a syntax error report, and, if the list expression consists solely of a literal string not containing quotation marks, leading blanks, or trailing blanks, the quotation marks may be omitted. If the quotation mark character is to be included in the literal string, then two quotation marks must be indicated.

#SV

◆ The function #SV yields the source code character string representation of the value of an ordinary arithmetic or logical expression.

#DM

◆ The function #DM yields a character string which is the source code representation of the $e_1{}^{th}$ dimension of the array whose name must be the contents of #n [(e_2)].

#CH

◆ The function #CH yields a character string of length n which includes the m^{th} through the $(m+n-1)^{th}$ characters of the value of the list expression ⌜list-exp⌝.

#PB

◆ The function #PB yields a character string which is equivalent to the argument syllable list with blanks preserved.

A more detailed discussion of these functions is included in the description of the evaluation of list expressions.

EVALUATION OF LIST EXPRESSIONS

◆ A list expression is evaluated by the juxtaposition of its component parts (i.e., without introducing any blanks between the component parts). The resultant character string is the value of the list expression.

The rules for the evaluation of the component parts are given below.

Literal Strings Surrounded by Quotation Marks

◆ The result string consists of the entire literal string which is contained within the quotation marks. Two consecutive quotation mark characters within the literal string is interpreted as a single quotation mark character in the result string at that point.

Syllable Sublist Names (#n[(e)])

♦ The result string consists of the contents of that list, with blanks squeezed out. If the syllable sublist name contents is null the result is also null.

Independent Syllable List Names (#nL)

♦ Same as for syllable sublist names above.

#SV(e)

♦ The arithmetic or logical expression e, is evaluated. The result string is the source code character string representation of the value of that expression. The result string will be output in logical, integer, floating point, or complex notation as appropriate.

#DM (e_1, #n [(e_2)])

♦ The result string is the source code representation of the $e_1{}^{th}$ dimension of the array whose name must be the contents of #n [(e_2)].

#CH(m,n, ⟨list-exp⟩)

♦ Let c = number of characters in the value of ⟨list-exp⟩

d = min(c,m+n-1)

The result is the m through the d^{th} characters of the value of ⟨list-exp⟩, or is null if m is greater than c.

#PB (#n [(e)] or #PB (#nL)

♦ The result string consists of the contents of the argument syllable list including all blanks. Note that the evaluation of syllable sublist names and independent syllable list names will always result in the blanks being squeezed out if the #PB function is not used.

DEFINITIONAL PROGRAMS

♦ Definitional programs effectively permit arbitrary extensions to the FORTRAN language in a way which is analogous to the way programmer-defined macros permit an extension to an assembler language.

A definitional program consists of a #DEFINITION statement, possibly followed by specification statements, followed by one or more #PATTERN statements, followed by an editing program, followed by an END statement.

The following schema illustrates this arrangement:

```
#DEFINITION . . .

[Specification statements]

#PATTERN

[#PATTERN statements]

Editing program

RETURN
  .
  .
  .

END
```

DESCRIPTION OF EDITING STATEMENTS

◆ Following is a detailed description of each of the FORTRAN editing statements, including its syntax, a description of its function, and examples of its use where appropriate.

#GET

◆ *Format:*

$$\#GET \quad \left[\begin{Bmatrix} C'n' \\ X'h' \end{Bmatrix}\right] \quad \langle file\rangle, \quad \left[\begin{Bmatrix} C'n' \\ X'h' \end{Bmatrix}\right] \quad \langle file\rangle, \ \ldots$$

This statement finds (by using the catalog) the named files on the disc and reads them into virtual memory. Each file name becomes defined in the symbol table as a file name and may be referenced subsequently in editing statements. All names of programs within a file which are read into virtual memory by the #GET become defined as program names within their respective files and may be used subsequently in editing statements to designate the list of statements which they contain.

C'n' or X'h' are password options. In the case that a password is required to access the named file, it is specified either by C'n' where n is a 1-4 alphanumeric character password, or by X'h' where h is a 1-8 digit hexadecimal password, depending upon the type of password associated with the particular file.

If more than one file is in virtual memory and programs with the same name occur in different files, a unique program may be designated in editing statements by qualifying the program as ⟨file⟩.⟨prog⟩.

Examples

```
#GET QUIRK
#GET QUIRK, QUARK
#GET C'A4D' PAY
#GET FILE1.PROGA
```

When #GET is issued, an OPEN is given, and the file is copied to virtual memory and then closed.

#SAVE

◆ *Format:*

$$\#SAVE \left[\text{AND DELETE}\right] \left[\begin{Bmatrix} C'n' \\ X'h' \end{Bmatrix}\right] \langle file\rangle \left[= (\langle stmt\text{-}set\rangle, \langle stmt\text{-}set\rangle, \ldots)\right], \left[\begin{Bmatrix} C'n' \\ X'h' \end{Bmatrix}\right] \langle file\rangle \left[= (\langle stmt\text{-}set\rangle, \ldots)\right], \ldots$$

This statement stores each file named ⟨file⟩ back on the disc, or enters its name in the catalog and then stores it on the disc if it is not already a catalog entry. If a ⟨stmt-set⟩ is given, the data to be stored in the file is the concatenation of the contents of ⟨stmt-set⟩, ⟨stmt-set⟩, . . . If there was no file previously cataloged by the name ⟨file⟩, and no ⟨stmt-set⟩ is given, then a new file is cataloged and the entire contents of the current session are #SAVE'd in the file. The #SAVE statement does *not* serve to name or create a file in virtual memory. The password options C'n' or X'h' are as described in #GET.

#SAVE *(Cont'd)*

The file (or the statement sets) remains unaltered in virtual memory unless the AND DELETE option is written. In this case statement sets are deleted from virtual memory as if they had been specified by a #DELETE statement.

Examples

```
#SAVE     TIME,HIGH
#SAVE     GLOB = (TIME (&10-10.12,16.1 ),QUE)
```

#UNSAVE

◆ *Format:*

#UNSAVE ⟨file⟩,⟨file⟩, . . .

This statement causes the files named to be deleted from the catalog and from the disc. It does not affect virtual memory.

#PREFIX

◆ *Format:*

#PREFIX [⟨file prefix⟩]

This statement causes the interpreter to remember the ⟨file prefix⟩ and to prefix it to any file names mentioned subsequently in editing statements.

The ⟨file prefix⟩ has one of the following forms:

⟨identifier⟩

⟨identifier⟩.⟨identifier⟩ . . . . . ⟨identifier⟩

For example, if the two editing statements

```
#PREFIX A
#GET B
```

were executed, then the effect would be to get the file A.B.

A #PREFIX with no ⟨file prefix⟩ serves to cancel the last previous #PREFIX statement.

#DELETE

◆ *Format:*

#DELETE [⟨stmt-set⟩, ⟨stmt-set⟩, . . .]

This statement causes the specified statement sets to be deleted from virtual memory. The symbol tables of any program which are *completely* deleted by the statement are forgotten. (See the #FORGET statement.) The names of any lists, programs, or files which are completely deleted are also forgotten.

The line numbers of material which has not been deleted are not affected.

A #DELETE with no statement sets means delete the unnamed or only file.

#MOVE

◆ *Format:*

#MOVE [BEFORE] ⟨place⟩ = (⟨stmt-set⟩, ⟨stmt-set⟩, . . .),

[BEFORE] ⟨place⟩ = (⟨stmt-set⟩, . . .) . . .

This statement causes the information contained in the statement sets on each right-hand side to be copied after (or before, if the optional BEFORE is written) the information contained in the corresponding statement or list designated by ⟨place⟩, after which the original copies of the statement sets are deleted as described under the #DELETE statement.

#INSERT

◆ This statement may be used at a terminal to enter a new file or to insert or change statements in an existing file (for making changes or corrections). It may also be executed as part of a stored program. These uses are discussed separately below.

Entering a New File from the Terminal

◆ When the interpreter is expecting to execute a statement from the terminal (i.e., is in the immediate execution mode), the #INSERT statement may be used to name and prepare for the entering, checking, and storing of a program file from the terminal. The simplest form of the #INSERT to use in this case is:

#INSERT [⟨file⟩]

If the file (named or unnamed, as the case may be) is empty, the system will then print the line:

```
FILE IS EMPTY
```

and the first line number:

```
1.
```

If the file is not empty, the system responds with the line:

```
LAST LINE IS 336
```

and the next available line number:

```
337.
```

A previously typed line may be corrected by retyping the line number and the new line. Lines are inserted in the file in order by line numbers. Thus, an insertion between two previously stored line numbers may be made by simply entering a line number intermediate between the line numbers of the previously typed lines. A previously typed line may be deleted from the file by typing only the line number followed by an end-of-transmission character *(ETX)*.

If the automatic line number and increment must be changed temporarily, an @ symbol followed by another line number and parenthesized increment will save the old line number, and start prompting with the new one. To use the old automatic line number and increment, an @ symbol followed by an *ETX* is given.

Entering a New File from the Terminal (Cont'd)

If a new line number and increment is to be used and the old one not retained, the statement is given as:

@SET ⟨line-no⟩ [(increment)]

For a complete description of these commands, see "Line Number Commands" on page 2-4. A line number may be rejected by sending a null line (i.e., sending only an end-of transmission character). If this is done, automatic line number generation is suppressed and the system enters the immediate execution mode; i.e., it executes immediately any statement given to it *without a line number*. Thus, side computations or editing operations may be performed while constructing the program. In order to return to INSERT mode, a null line is again sent.

The end of the statements to be stored in the file may be indicated by an #END statement given *without a line number*, in the immediate execution mode.

The following example illustrates the procedure. The lower case phrases and sentences are comments on the procedure and would not appear on the terminal listing.

Example

```
*       #INSERT
FILE IS EMPTY
1.      @ SET 100 (1)
100.    X = 2.5
101.    Y = 3.7
102.    A = A + X
103.    B =                          RETURN character sent
                                     for continuation.
- 104.      A + Y
105.      C = 3.7
106.    ETX     <NULL>               No spaces before ETX.
                                     ETX sends user to desk
*       #DELETE 102                  calculator mode. Second
                                     ETX sends user back to
*       ETX     <NULL>               insert mode.
106.    C    THIS IS A COMMENT.
```

↑
Printed by system not user.

Changing an Existing File from the Terminal

◆ The procedure for making insertions or changes to a file that already exists in virtual memory is very similar to entering a new file. The main differences are that one may specify a ⟨place⟩ where an insertion is to be made and that the system responds to the #INSERT by informing the user of the existing line numbers in the vicinity of the designated ⟨place⟩.

Example

```
*   #INSERT BEFORE LACK (&10)
    BETWEEN LINES 210.2 AND 211      <- Printed by system not user
    210.3   X = 22.2
    210.4   Y = 26.1
    210.5   Z = 20.
    210.6   ETX
*   #END
```

Example

```
*   #INSERT   ALPHA
    LAST LINE IS   662.01      <-- Printed by system not user.
    662.02   STOP
    662.03   END
    662.04   ETX   <NULL>
*   #END
```

Syntax Checking and Statement Reformatting

◆ When a program is initially being entered at the terminal, it is normally checked for executability by the interactive system. This check is carried out by also building a symbol table and checking for consistency in the use of names. In order to change this normal mode, the following control options are allowed on the #INSERT statement:

```
[ { AND [COMPILER] [SYNTAX] } CHECK ]  [BEFORE] [⟨place⟩]
  { WITHOUT                 }
    DATA
```

Syntax Checking and Statement Reformatting (Cont'd)

The checking options are the same as the options available with the following statement:

#[COMPILER] [SYNTAX] CHECK . . .

If corrections to an already existing program are being inserted from the terminal, the symbol table is not normally completed for that program and, in that case, the checking is done without reference to a symbol table (corresponding to the SYNTAX CHECK option).

Syntax Checking

♦ When a statement is entered in the desk calculator mode, or in the insert mode with a specific request for syntax checking, such checking is done by the system and any errors detected are reported, one at a time. That is, if one or more errors are detected in a statement, the user is notified of the first and given a chance to correct it, then he is notified of the next error, and so on. A number of possible responses are acceptable to the system and will be described below.

Error Reporting

♦ If a syntax error is detected by the TSOS Interactive FORTRAN System when a statement is entered, a question mark is typed at the extreme left of the next line, and the syllable list name #1 becomes defined to be that syllable or character which was being examined when the error was detected (as if it has been located by a #FIND statement).

If the user cannot determine what the error is, he may respond with a ? followed by ETX, whereupon the system types on the next line another ? directly under (or as close as possible to) the syllable in error and then skips to the next line.

For further information, the user may again respond with ?*ETX* in which case the system types a diagnostic message on the next line followed by another ? on the following line. Further responses of ? by the user merely result in a repetition of the error message and final ? .

Example

```
      27.       READ(97,10)A,
?  ?ETX
                                  ?
   ?ETX
IDENTIFIER EXPECTED
?
```

Error Reporting (Cont'd)

Corrective procedures may be invoked at any time immediately after the system responds with any of its question marks, as described in the next section below.

In some cases, the line containing the error is reprinted before the second ? is typed by the system. This occurs when the line containing the error is not the last one typed (e.g., the error occurs in the second of three continued lines), or if there is more than one error in the line and a correction changes its original content. When the line is reprinted it contains a * in place of the decimal point in the line number, to indicate to the user that this is a line typed back by the system, and not the original line typed by the user.

Example

```
        29.     A=B++(C+D
? extra plus sign deleted by user
? ?ETX
        29*0000 A=B+(C+D
                      ?
```

The provisions for making the deletion indicated in the example and other facilities for correcting errors are described next.

Error Correction

◆ When a syntax error is reported as described above, the user has four options which he may exercise at any stage immediately following the receipt of a ? from the system.

1. Ignoring the error

   If for any reason the user wishes to ignore the error and continue processing he simply types # (followed by *ETX*). The system responds with the next line number (in insert mode, or with * in desk calculator mode). Note that in this case if there is more than one error in the line of text, all remaining ones are also ignored and are not even reported to the user.

2. Deleting the erroneous syllable

   To delete the syllable in question, the user types the backspace character _ (shift-O on the teletype) followed by *ETX*.

Error Correction (Cont'd)

3. Replacing the erroneous syllable

   To replace the syllable in question with a correct one the user simply types in the new syllable (followed by *ETX*) immediately following the system's ? . Note that in this case the backspace character, if it appears, is considered part of the replacement text and does not serve to delete the preceding character. Alternatively, a syllable list assignment statement may be used to change a syllable in the vicinity of the syllable in question (relative to #1).

4. Making side computations

   If the error occurs in the insert mode, the user may enter the desk calculator mode in order to make side computations before applying the correction, by typing a null message (*ETX* alone). Another null message, in the desk calculator mode, returns him to the insert mode so that he may then make the correction based on this side computation.

In cases 2 and 3 above, it should be noted that the system always applies the correction to the syllable to which the second ? printed by the system is pointing. In a few cases this is not necessarily the syllable which the user expects. Thus caution is in order in applying such corrections to avoid unexpected results.

Example

```
        5.          A=B++C
? ?ETX
                        ?
#ETX
        6.          IF (A .GT. B)) 46
? ?ETX
                              ?
?ETX
TOO MANY RIGHT PARENTHESES
? __ETX
? ?ETX
            6*0000      IF (A .GT. B) 46
                                  ?
?ETX
BAD 'THEN' CLAUSE IN LOGICAL 'IF'
? GO TO 46 ETX
        7.
```

Reformatting of Statements and Entering Data

◆ When program statements are entered with the #INSERT, they are automatically reformatted to obey the column conventions required by the background compiler so that an indexed sequential file entered from a terminal can be compiled; i.e., statement numbers are put in column positions 1-5, continuation marks in column 6, and END statements in columns 7-9. If this reformatting is to be avoided, then the DATA option of the #INSERT must be used.

Use of #INSERT to Copy Statements

◆ The information being inserted may also be contained in a file in virtual memory. There are two cases: (1) the statements (to be inserted) immediately follow the #INSERT, or (2) they exist elsewhere in the files in virtual memory. In the first case, the statements to be inserted are followed by an #END statement which marks their extent. The following example shows how such an #INSERT might itself be inserted in a program file:

```
* #INSERT     PROG (&10)
  BETWEEN     271 AND 272
  271.1       #INSERT PROG (&40)
  271.2       X = 2.
  271.3       Y = 3.
  271.4       #END
  271.5       ETX
* #END
```

This is stored.

This is executed.

In the case that the statements to be inserted exist elsewhere in virtual memory, the format of #INSERT to be used is as follows:

#INSERT [⟨check⟩] [BEFORE] ⟨place⟩ = (⟨stmt-set⟩, . . .) ,
[BEFORE] ⟨place⟩ = (⟨stmt-set⟩, . . .) . . .

The effect of this statement is to copy the statements on each right-hand side after (or before) the statement(s) designated by the corresponding ⟨place⟩.

In these cases, no checking occurs unless it is explicitly asked for by including some form of the ⟨check⟩ which has the following syntax:

AND [COMPILER] [SYNTAX] CHECK

#REPLACE

◆ This statement has the same syntax as #INSERT except that wherever ⟨place⟩ is legal in #INSERT, ⟨stmt-list⟩ is legal in #REPLACE and the optional BEFORE is not allowed.

The effect is that the ⟨stmt-list⟩ is replaced by the new information.

#NUMBER

- *Format:*

#NUMBER ⟨stmt-list⟩ [=(⟨number⟩ [,⟨increment⟩])] ,
⟨stmt-list⟩ [=(⟨number⟩ [,⟨increment⟩])] , . . .

This statement causes new line numbers to be formed for each ⟨stmt-list⟩ mentioned. If a line number is given, the line numbers of the statements are:

⟨number⟩

⟨number⟩ + ⟨increment⟩

⟨number⟩ + 2(⟨increment⟩)

.
.
.

Both ⟨number⟩ and ⟨increment⟩ may be arbitrary arithmetic expressions. If the ⟨increment⟩ is omitted, it is assumed to be 1.

If the line number is omitted and the ⟨stmt-list⟩ is a sublist of some file, then the line numbers are spread uniformly over the available gap in line numbers previously occupied by the ⟨stmt-list⟩; (i.e., as if it had just been inserted in the file). If, however, the ⟨stmt-list⟩ is a complete file, then it is numbered starting at 100 with an increment of one.

#ORDER

- *Format:*

#ORDER ⟨name⟩, ⟨name⟩, . . .

This statement rearranges certain non-executable statements in one or more programs into the required sequence for execution. The ⟨name⟩'s may be program names or file names. If a file name is specified, then each program in the file will be rearranged separately.

The required order of statements for execution is:

1. FUNCTION, SUBROUTINE, BLOCK DATA, #DEFINITION, or PROGRAM.
2. IMPLICIT
3. ABNORMAL
4. EXTERNAL
5. Explicit type
6. DIMENSION
7. COMMON

#ORDER *(Cont'd)*

8. EQUIVALENCE
9. DATA
10. #PATTERN
11. Statement function definitions
12. Any other statements
13. END

#PREP

◆ *Format:*

#PREP ⟨stmt-set⟩, ⟨stmt-set⟩ , . . .

This command searches the specified ⟨stmt-set⟩'s and deletes any editing or debugging statements which they contain.

#CHECK

◆ *Format:*

#[COMPILER] [SYNTAX] CHECK [AND PRINT] ⟨stmt-set⟩ ,
⟨stmt-set⟩ , . . .

This statement causes the statements in the designated statement sets to be checked for legality for the Interactive FORTRAN System or for the background compiler (if optional COMPILER is written).

If the optional SYNTAX is written, then each statement is checked by itself without reference to a symbol table; otherwise, the appropriate symbol table is used for each statement to check for consistent use of identifiers and statement numbers.

Normally, the errors are reported conversationally at the terminal in the same manner as described for the #INSERT statement, except that the line containing the error must first be typed before the mark can be typed. If the optional AND PRINT is written, however, full error comments are printed off-line.

#NAME

◆ *Format:*

#NAME { ⟨ident⟩ [:⟨stmt-list⟩] , ⟨ident⟩ [:⟨stmt-list⟩] , . . . / ⟨file⟩ , ⟨file⟩ , . . . } , . . .

The first form of this statement assigns each ⟨ident⟩ as the name of the corresponding ⟨stmt-list⟩ (the list itself is not moved or copied). If there is no ⟨stmt-list⟩, then the ⟨ident⟩ is defined to be the name of an empty file in virtual memory. An ⟨ident⟩ is a 1-6 character FORTRAN identifier.

The second form of this statement causes an empty file to be created in virtual memory having the name ⟨file⟩, where ⟨file⟩ is a file name conforming to the standard TSOS naming conventions.

#FORGET

- *Format:*

#FORGET { ⟨qualifier⟩ | ⟨qualifier⟩ (⟨ident⟩ , . . .) } , . . .

This statement is used to remove all entries in the symbol table for a ⟨qualifier⟩ (a program, subprogram, or COMMON block), or only those identifiers listed in parentheses after the ⟨qualifier⟩.

#REFER

- *Format:*

#REFER [⟨qualifier⟩ [(⟨ident⟩ , ⟨ident⟩ , . . .)]]
, [⟨qualifier⟩ [(⟨ident⟩ , ⟨ident⟩ , . . .)]] , . . .

This statement changes (usually temporarily) the symbol table which is used to interpret references to variables in subsequent statements. If #TABLE (with no further specifications) is executed, it means "use the symbol table of this program"; i.e., revert to the symbol table of this program for interpretation of all identifiers.

If some additional specification is given, it means interpret all or some of the identifiers in following statements according to the symbol table of some other program or COMMON block. The ⟨qualifier⟩ is either a program or subprogram name or the name of a COMMON block. If specific identifiers are listed in parentheses, it means to interpret only those variables as belonging to ⟨qualifier⟩.

A frequent use for this statement occurs when it is desired to refer to variables within some program or subprogram from the direct program; i.e., from the desk calculator program which is immediately executed from the terminal.

#IN

- *Format:*

#IN ⟨stmt-set⟩ , ⟨stmt-set⟩ , . . .

This phrase directs the attention of the interpreter to the particular sets of statements specified by the ⟨stmt-set⟩'s. It is usually used as an introductory phrase to the #FIND, #EDIT, or #EXPAND statements, to select those statements to be found, edited, or expanded. This phrase may also appear as a statement by itself.

#FIND

- *Format:*

[⟨#IN-stmt⟩,] #FIND ⟨search-exp⟩ [==⟨list-exp⟩]
[#AT ⟨rel-place⟩,]

#FIND *(Cont'd)*

The #FIND statement locates the next instance of a specified list of syllables within the statements specified and optionally changes that string of syllables. The #FIND statement causes the statement sets specified by the #IN statement (if an #IN statement is prefixed or just previously executed) to be searched starting at the beginning; or at a place determined by a previous #FIND; or at ⟨rel-place⟩, which is a relative syllable or relative statement position, i.e., is #n [(e)] or #Sn [(e)]. Each statement is examined, left to right, to find the first syllable list within it which satisfies the conditions specified by the *search expression,* ⟨search-exp⟩. When the first matching syllable list is found, it is replaced by a list which is specified by the *list expression,* ⟨list-exp⟩, on the right of the double equal sign. If there is no double equal sign, then the only effect of the #FIND is to possibly define one or more of the special syllable list variables #1, #2, . . . and to advance the current position in the statement sets defined by the previous #IN.

If a #FIND statement has located at least one occurrence of the ⟨search-exp⟩, then the special logical variable #FOUND is set to .TRUE.; otherwise, it is set to .FALSE.. Thus, the success of the search can be subsequently tested with a FORTRAN logical IF; e.g.,

```
IF (#FOUND) GO TO 20
```

#EDIT

◆ *Format:*

$$\left[\begin{array}{l}\text{⟨\#IN-stmt⟩,}\\ \text{\#AT ⟨rel-place⟩,}\end{array}\right] \text{\#EDIT⟨search-exp⟩==⟨list-exp⟩}$$

The #EDIT statement is very similar to #FIND except that it finds and changes *all* instances of a specified string of syllables within the specified statements. The search procedure described under #FIND is similar except that it continues in #EDIT immediately following the (modified) ⟨search-exp⟩ to search for another matching syllable list. This search continues until the end of the statement sets has been reached. The special logical variable #FOUND is set for #EDIT exactly as described under #FIND.

#DEFINITION

◆ *Format:*

```
#DEFINITION ⟨def-name⟩ [(⟨arg⟩ , ⟨arg⟩ , . . . )]
```

This statement introduces a definitional program. The ⟨def-name⟩ is the name of the definitional program and may be used to reference it in #EXPAND statements in other programs (or from the terminal). The list of ⟨arg⟩'s contains dummy arguments which must be passed by the #EXPAND statements. There need not be any arguments; in which case, the parentheses are also omitted.

#PATTERN

- *Format:*

  #PATTERN ⟨search-exp⟩ / / ⟨stmt-no⟩

The #PATTERN statement defines a special type of program, a *definitional* program which, when referenced by an #EXPAND statement, may cause extensive systematic editing of statement sets. #PATTERN defines a pattern which is to be searched for in the statement sets being expanded by the just previously executed #EXPAND statement. When the pattern is found, control passes to the statement whose number is ⟨stmt-no⟩. Syllable list variables may be defined within the ⟨search-exp⟩ as if it had occurred within a #FIND statement. The program starting at ⟨stmt-no⟩ is normally an editing program which will make some change in the program being expanded. Execution of a RETURN statement will cause the searching process to continue immediately after the pattern which was located by the ⟨search-exp⟩.

If it is desired to change the place of continuation of the search, then the special pointer variable, #SEARCH is set equal to some special list or relative list; i.e.,

#SEARCH = #n[(e)]

just before the RETURN.

This causes the search to continue the expansion starting at the syllable designated by #n[(e)].

#EXPAND

- *Format:*

  [⟨#IN-stmt⟩,] #EXPAND ⟨def-name⟩ [(⟨arg⟩ , . . . , ⟨arg⟩)]

This statement causes the statement sets designated by the attached (or just previously executed) #IN statement to be expanded by the definitional program ⟨def-name⟩. The actual arguments ⟨arg⟩ , . . . , ⟨arg⟩ are passed to the definitional program by the #EXPAND. The definitional program must be in virtual memory at the time the #EXPAND is executed.

4. DEBUGGING STATEMENTS

INTRODUCTION

◆ The statements described in this section provide the debugging tools of the TSOS Interactive FORTRAN System. They provide means for:

1. Displaying the values of variables of the program.
2. Displaying selected variables when they are changed.
3. Checkpointing the program and continuing and then later resuming at the previous checkpoint.
4. Branching to another program whenever certain variables are changed.
5. Printing a list of all statements of a subprogram which have (have not) been executed during the test.
6. Displaying the present call structure of the programs.

These debugging statements may be written into a program which is to be subsequently executed under the TSOS Interactive FORTRAN System, or they may be issued from the terminal in the desk calculator mode. There is an editing statement, #PREP, which assists the user in preparing a program or file for the TSOS background compiler. One of the functions of this command is to remove all debugging and editing statements.

The TSOS Interactive FORTRAN System provides a special identifier #STEP The value of this integer variable reflects the current computational step. This value is incremented *before* each statement is interpreted.

The computational step number applies to the entire program and is never reset except at the start of a session. The following statements do *not* affect the step number:

FORMAT
NAMELIST
DATA
BLOCK DATA
ENTRY
SUBROUTINE
FUNCTION
END
PROGRAM
Specification statements
CONTINUE

All *other* statements cause the step number to be incremented by 1.

INTRODUCTION (Cont'd)

In the following discussion, ⟨spec-list⟩ is a string of characters of the following form:

/x/a,b, . . .

where:

x is a subprogram, program, or COMMON name,

and

a, b, . . . are variable names within x.

DESCRIPTIONS OF DEBUGGING STATEMENTS

♦ This section describes each of the debugging statements of the TSOS Interactive FORTRAN System, including its syntax and a description of its function.

#DISPLAY

♦ *Format:*

#DISPLAY ⟨what to print⟩

This statement displays, at the terminal, information about programs in virtual memory that are being executed. The formats of ⟨what to print⟩ are listed below.

Note:

In the description of the forms,

the b's represent block names, program names, or subprogram names;

c,d,e,f, . . . represent identifiers or statement numbers, whichever is appropriate, within the specified COMMON block or program; and

"exp" represents an arithmetic expression.

Form 1

♦ VAR $\left[\text{[NOT] CHANGED [SINCE (exp)]}\right]$ [(/b/ [c,d, . . .]

/b/ [e,f . . .] . . .)$\left.\right]$

This statement prints the values of variables or arrays. Either the changed (since computational step "exp") or unchanged variables can be printed. In any case, the quantities to be printed are selected from the list supplied within parentheses. If no identifiers are listed for a given qualifier (block-name, program name, etc.) it means all the variables associated with that qualifier.

Form 2

♦ STMT [ID] $\left[\text{[NOT]} \left\{\begin{matrix}\text{CHANGED}\\ \text{EXECUTED}\end{matrix}\right\} \text{[SINCE (exp)]}\right]$

$\left[\text{⟨stmt-set⟩ , . . . }\right]$

This statement prints statement sets or only the line numbers and statement numbers of selected statements within the statement sets. It can be specified that only statements which have been changed by editing statements or only those not changed, or only those statements (not) executed are to be printed. The SINCE clause permits a more specific meaning to be given to "CHANGED" or "EXECUTED".

Form 3

◆ VAR XREF [[NOT] CHANGED [SINCE (exp)]] [(/b/ [c,d, . . .] /b/ [e,f . . .] . . .)]

This statement generates cross reference tables for identifiers; that is, lists of every statement that mentions each selected identifier.

Form 4

◆ STMT XREF [[NOT] {EXECUTED / CHANGED} [SINCE (exp)]] [(/b/ [c,d, . . .] /b/ [e,f . . .] . . .)]

This statement causes the generation of cross reference tables for statement numbers; i.e., lists of line numbers and statement labels that contain references to each selected statement number.

Form 5

◆ {VAR / STMT} TABLE [(b,b . . .)]

This statement causes the printing of either the variables or the statement numbers present in selected programs.

Form 6

◆ BRANCHES [[NOT] EXECUTED [SINCE (exp)]] [(b,b, . . .)]

This statement generates selected lists of the branches of each program along with indications of which alternatives have actually been exercised (since computational step "exp").

Form 7

◆ LINKAGE

This statement displays the current subroutine linkage. This includes:

1. The name of the current subprogram and the line number of the current line being executed.
2. The name of the calling program and the line number from which the CALL or reference was invoked.
3. Similarly for all other calling programs to the top level program.

Form 8

◆ CHANGES [OFF] [(/b/ [c,d, . . .] /b/ [e,f . . .] . . .)

This statement prints the value of a selected variable each time it changes during execution. (A variable is considered to have changed whenever a value is stored in it, even if the new value is identical to the old.) The statement remains in effect until cancelled by a corresponding statement using the OFF option.

Form 9

◆ PROGRAM NAMES ['file-name' , 'file-name' , . . .]

This statement prints the name of each program in the files named, and the line numbers of the first and last statements in each program. If no 'file-name' is specified, the unnamed or only file is assumed.

#PRINT

- *Format:*

  #PRINT ⟨what to print⟩

This statement prints off-line, information about programs in virtual memory that are being executed. The specification ⟨what to print⟩ is identical to that of #DISPLAY including all of the same forms and options as described in detail under the #DISPLAY description.

#WHEN

- *Format:*

  #WHEN (e) S

  where:

  e is a logical expression and S is an executable statement (except DO, a logical IF, or another #WHEN).

Execution of the #WHEN statement causes the logical expression e to be evaluated between each statement which is processed by the interpreter. The statement S is then executed if the logical expression e yields a TRUE value.

The scope of a #WHEN statement is that program or subprogram within which it is written. That is, all #WHEN statements which are active in a program are temporarily disabled during the execution of any subordinate subprograms and re-enabled when control is returned to the calling program.

If the statement S causes a transfer of control (other than CALL or a function reference) then that transfer point must be within the same subprogram and control must be returned to the interpreter by means of a #RETURN statement. All #WHEN statements are disabled until after the #RETURN has been executed.

Example

Line No.	Stmt. No.	Statement
100.		A=3.
101.		CALL B(A)
102.		STOP
103.		END
104.		SUBROUTINE B(A)
105.		#WHEN (A=5.) GO TO 30
106.		C=D
107.		#WHEN (I=2) GO TO 40
108.		D=E
109.		A=5.
110.		B=6.
111.		RETURN
112.	30	#DISPLAY LINKAGE
113.	40	A=0.
114.		#RETURN
115.		END

#WHEN ***(Cont'd)***

Explanation

The scope of the first #WHEN is lines 106-111; that of the second, lines 108-111. At line 101 the subroutine B is called, causing the execution of line 105 which enables the first #WHEN. At line 107 the second #WHEN is enabled. At line 109 the first #WHEN is satisfied, whereupon control goes to statement 30 (line 112) as specified by that #WHEN. Since satisfaction of the #WHEN causes a transfer of control (to statement 30), both #WHEN's are temporarily disabled during the execution of lines 112-114. The #RETURN at line 114 re-enables the #WHEN's and returns control to line 110. The RETURN at line 111 disables both #WHEN's (permanently in this example, since there are no further calls on B) and returns control to the main program at line 102.

#RETURN

◆ This statement returns control to the interpreter, after a transfer to a subprogram from a #WHEN statement, reactivating the #WHEN.

#TURN OFF

◆ *Format:*

#TURN OFF ⟨end-pt⟩

where:

⟨end-pt⟩ is as described on pages 3-1 and 3-2, and specifies a #WHEN statement. A diagnostic results otherwise.

This statement disables the specified #WHEN statement. The #WHEN statement remains disabled until it is executed again.

#TRACE

◆ This statement turns the trace mode on. The output of the trace mode consists of the line number of each statement which causes a transfer of control and the statement number to which the control is given, in the program or subprogram in which the trace is active.

The scope of the #TRACE statement is the same as for the #WHEN statement.

#NO TRACE

◆ This statement turns off the trace mode for the program or subprogram in which the statement is executed.

#CHECKPOINT

◆ *Format:*

#CHECKPOINT (f)

where:

f is a file name.

This statement causes a checkpoint to be taken on the file designated by f.

#RESTART

- *Format:*

  #RESTART (f)

  where:

  f is the name of a file which contains a checkpoint.

This statement restarts the program after the point of the checkpoint contained on file f.

#EXECUTE

- *Format:*

  #EXECUTE ⟨stmt-list⟩

This statement causes the statements in ⟨stmt-list⟩ to be executed; that is, it effectively causes a transfer of control to the first statement of the ⟨stmt-list⟩ and terminates the execution after the last statement in the ⟨stmt-list⟩ has been executed for the first time.

This statement may be used to execute a main program, or a BLOCK DATA subprogram. The latter causes initialization of named COMMON's. Then, if the file starts with a main program, that main program is executed.

During debugging, a small section of any program or subprogram may be executed by the use of this statement.

Notice that there already exists a way of causing a subroutine or function subprogram to be called while passing parameters; i.e., the CALL statement and the function reference.

#PROCEED

- This statement is executed only at the terminal. It causes the Interactive FORTRAN System to resume computations at the point where it was when interrupted by a "break sequence", in order to examine variables, to make side computations, or to make changes by executing direct statements at the terminal.

#HALT

- This statement terminates an individual user's current session of the TSOS Interactive FORTRAN System and returns control to the TSOS Executive System.

The system prompts the user in order to notify him that the file(s) currently in virtual memory must be #SAVE'd. The files must be saved if he is to retain their content (*perhaps* edited or updated) before executing the #HALT which would erase them from virtual memory.

APPENDIX A

SUMMARY OF EDITING COMMANDS

Commands	Entities	Specials
#INSERT #REPLACE #NUMBER #GET #SAVE #PREP #NAME #PREFIX #UNSAVE #CHECK #ORDER #MOVE #DELETE	files programs COMMON blocks statement lists statement sets	
#FIND #EDIT	syllables syllable sets syllable lists consecutive character strings search expressions	#FOUND #SEARCH
#AT	relative statement or syllable position	
#IN	statement sets	
#FORGET #REFER	symbol programs subprograms COMMON blocks	
#DEFINITION #PATTERN #EXPAND	definition name statement numbers syllable and others [e.g., search expressions (see #FIND)]	

Note:

All commands in a group do not necessarily apply to all entities in a corresponding group.

APPENDIX B

DESCRIPTION OF SYNTACTICAL UNITS

Entity	Description
name	program or file name
⟨file⟩	file name
⟨list-name⟩	name
	list of statements defined by a #NAME
⟨integer⟩	a 1-4 digit unsigned integer
⟨line-no⟩	line number of 1st line of statement
⟨stmt-no⟩	a 1-5 digit FORTRAN statement number
⟨end-pt⟩	selects a single statement from ⟨list-name⟩ ⟨line-no⟩ & ⟨stmt-no⟩ ⟨list-name⟩
	a statement located by a #FIND #S ⟨integer⟩ [(⟨exp⟩)]
⟨place⟩	⟨list-name⟩
	⟨list-name⟩ (⟨end-pt⟩)
	(⟨end-pt⟩)
⟨range⟩	extracts a list of consecutive statements ⟨end-pt⟩ ⟨end-pt⟩ – ⟨end-pt⟩ ⟨end-pt⟩ – – ⟨end-pt⟩
⟨stmt-set⟩	⟨list-name⟩
	⟨list-name⟩ (⟨range⟩, ⟨range⟩, . . . , ⟨range⟩)
	⟨list-name⟩ – (⟨range⟩, ⟨range⟩, . . . , ⟨range⟩)
	(⟨range⟩, ⟨range⟩, . . . , ⟨range⟩)
	– (⟨range⟩, ⟨range⟩, . . . , ⟨range⟩)
⟨stmt-list⟩	⟨list-name⟩
	⟨list-name⟩ (⟨range⟩)
	(⟨range⟩)

LIST NAMES

◆ A ⟨list-name⟩ is a mnemonic name which may be used to address either a file or possibly one or more statements contained in a file. A ⟨list-name⟩ is either a global list name or a local list name. A local list name may only be referenced within the program unit* in which it has been previously defined, while a global list name may be referenced by any program unit in the task.

Global List Name

◆ A global list name is a file name or a program name, where a program name is the name of a main program, the name of a subprogram (including #DEFINITION), or an ENTRY name.

File Name

◆ A file name is unique among all global list names in the task. The file name addresses all statements which that file contains. A file name may only be introduced into the task as the result of the execution of one of the following statements:

1. #GET
2. #INSERT
3. #NAME A,C
   (i.e., the #NAME statement without the optional ⟨stmt-list⟩)
4. #SAVE

An explicit file name does not exist for the unnamed file.

Program Name

◆ A program name is unique within the file in which it is stored. However, program names, within the above mentioned constraint, need not be unique. The program name addresses all statements in that program including the initial statement (PROGRAM, SUBROUTINE, FUNCTION, #DEFINITION) and the END statement. In the case of an ENTRY name the program name addresses only the ENTRY statement. A program name may only be introduced into the task as the result of the execution of one of the following statements:

1. #GET
2. #INSERT

An explicit program name does not exist in the source code representation for a BLOCK DATA subprogram; however, the interpreter assigns a name of the form #Bnn to each BLOCK DATA subprogram, where nn is 01 for the first BLOCK DATA subprogram encountered, 02 for the second, etc.

Global List Name Ambiguities

◆ If more than one named file is in virtual memory and the same program name appears in different files, then in editing statements, a unique program name may be designated by qualifying the program name with the file name as ⟨file⟩.⟨prog⟩. If a prefix is active at the time this sequence is interpreted (see #PREFIX statement), then that prefix is applied to ⟨file⟩.

The syntax of several statements specifies, among other things, either a file name or a program name. In these cases ambiguities may result.

*A program unit will refer to a main program, a subprogram, or to a sequence of statements being executed in the desk calculator mode.

Global List Name Ambiguities (Cont'd)

Example

```
#PREFIX A     Establish the active prefix as A.
#GET    B     Read file A.B into virtual memory (suppose file A.B
              contains programs B and C).
#INSERT B
```

In the above example, it is not clear whether the file A.B or the program B which is contained in file A.B is to be referenced. In all such cases, the ambiguities are resolved by the following convention. The specified name, prefixed where appropriate, is first checked as a file name, then as a ⟨file⟩.⟨prog⟩ name, and finally as a program name. The item which is first matched under this scheme is considered to be the referenced item. (Thus, the reference in the above example is to the file A.B.) The following algorithm specifies this procedure more precisely.

Let B be the specified name (e.g., A or A.B).

Let C be the specified name preceded by the active prefix, if any (e.g., P.A or P.A.B).

then,

1. If C is the name of a file name entry in the symbol table, then that file is addressed; otherwise

2. If B contains at least one period and, in addition,

2.1 If D is a program name which is contained in file E, then that program (D) is addressed, where D is the rightmost name in C and E is the string of characters in C up to, but not including, the period which immediately precedes D.

2.2 If D is not a program name which is contained in file E, then the reference is undefined.

3. If B does not contain at least one period, and

3.1 If B is a unique program name, then that program (B) is addressed.

3.2 If B is not a unique program name, either because it is undefined or because it is multiply defined, then that reference is either undefined or multiply defined, respectively.

Local List Name

◆ A local list name may only be defined in a program unit as the result of the execution of a #NAME statement. The local list name addresses those statements which are contained in the ⟨stmt-list⟩ (see #NAME). The local list name may only be referenced in the program unit in which it is defined. This is not to say that the lines, so addressed, cannot be referenced from a different program unit since they may also be referenced via some other local name, by a global name, or a line number, etc.

Using ⟨list-name⟩ to Address Statements

♦ It is possible to address a list of consecutive statements which perhaps may not have been explicitly named or to address a place in a file or a set of (not necessarily consecutive) statements. This is accomplished by the specification of certain combinations of list names, line numbers, or statement numbers. Explicit syntax rules for these constructs are given in the following sections.

Preliminary Definitions

⟨end-pt⟩

♦ The ⟨end-pt⟩ selects one or more statements out of the ⟨list-name⟩ with which it is associated (see ⟨place⟩, ⟨stmt-list⟩, ⟨stmt-set⟩). The forms for ⟨end-pt⟩ are:

1. ⟨line-no⟩
2. & ⟨stmt-no⟩
3. ⟨list-name⟩
4. #S⟨integer⟩ [(exp)]

where:

⟨line-no⟩ represents the line number of the first line of a statement and is an integer (1-4 digits) possibly followed by a fractional part (0-4 digits preceded by decimal point);
⟨stmt-no⟩ represents a (1-5 digit) FORTRAN statement number;
Form 3 may be any (list-name) except a file name; and
Form 4 addresses a single statement.

In all cases, the statement or statements which are addressed by ⟨end-pt⟩ must be completely contained in the associated ⟨list-name⟩. A diagnostic will be given otherwise.

Forms 1, 2, and 4 are used to address a single statement while form 3 may address more than one statement.

⟨range⟩

♦ The ⟨range⟩ is similar to ⟨end-pt⟩ in the sense that it is used to extract one or more consecutive statements from an associated ⟨list-name⟩. (See ⟨stmt-list⟩, ⟨stmt-set⟩. However, ⟨range⟩ is more general. ⟨range⟩ has the following forms:

1. ⟨end-pt⟩
2. ⟨end-pt-1⟩ – ⟨end-pt-2⟩
3. ⟨end-pt⟩ –
4. – ⟨end-pt⟩

Form 1 addresses all statements contained in ⟨end-pt⟩;

Form 2 addresses the statements from the first statement in ⟨end-pt-1⟩ through the last statement in ⟨end-pt-2⟩;

Form 3 addresses the statements from the first statement in ⟨end-pt⟩ through the last statement in the associated ⟨list-name⟩; and

Form 4 addresses the statements from the first statement in ⟨list-name⟩ through the last statement in ⟨end-pt⟩.

⟨range⟩ (Cont'd)

Since ⟨range⟩ is specified in terms of ⟨end-pt⟩, the statement or statements which are addressed by ⟨range⟩ must be completely contained in the associated ⟨list-name⟩.

⟨place⟩

◆ ⟨place⟩ selects a list of consecutive lines in a file. ⟨place⟩ is always used in a context which permits the optional word BEFORE to be specified:

[BEFORE] ⟨place⟩

◆ If the optional BEFORE is omitted, the place specifies the position in the file which immediately follows the last (or only) selected line.

If BEFORE is written, ⟨place⟩ specifies the position in the file which immediately precedes the first (or only) selected line.

⟨place⟩ may take one of the following forms;

1. ⟨list-name⟩
2. [⟨list-name⟩](⟨end-pt⟩)

In form 2, if the optional ⟨list-name⟩ is omitted, then a ⟨list-name⟩ is assumed according to the following rules. If there is more than one file in virtual memory, then the unnamed file is assumed. If there is only one file in virtual memory, then that file is assumed. In either case the line(s) specified by the ⟨end-pt⟩ must be a subset of the line(s) specified by the ⟨list-name⟩.

⟨stmt-list⟩

◆ ⟨stmt-list⟩ is used to select a list of one or more consecutive statements and has the following forms:

1. ⟨list-name⟩
2. [⟨list-name⟩](⟨range⟩)

If the optional ⟨list-name⟩ is omitted from form 2 then a ⟨list-name⟩ is assumed according to the rules given above (see "[BEFORE] ⟨place⟩").

Form 1 addresses the statements contained in ⟨list-name⟩ and Form 2 extracts that list of consecutive statements which are addressed by ⟨range⟩ from the list of statements addressed by ⟨list-name⟩.

⟨stmt-set⟩

◆ Many of the editing statements operate on specified sets of statements which are not necessarily consecutive. The ⟨stmt-set⟩ is the construct which is used to address such sets of statements. Its forms are:

1. ⟨list-name⟩
2. [⟨list-name⟩] (⟨range⟩[,⟨range⟩]...)
3. [⟨list-name⟩]-(⟨range⟩[,⟨range⟩]...)

Form 1 addresses the statements contained in ⟨list-name⟩.

In forms 2 and 3, if the optional ⟨list-name⟩ is omitted, then a ⟨list-name⟩ is assumed according to the rules above (see "[BEFORE] ⟨place⟩").

⟨stmt-set⟩ (Cont'd)

Form 2 extracts the lists of consecutive statements which are addressed by the ⟨range⟩'s from the associated ⟨list-name⟩. The ⟨range⟩'s must be contained in the ⟨list-name⟩. The ⟨range⟩'s are processed from left to right in the order in which they are specified.

Form 3 extracts the complement of the statements which are addressed by the ⟨range⟩'s from the statements contained in the ⟨list-name⟩. The ⟨range⟩'s are processed from left to right and must be specified in ascending, non-overlapping, order (by line number). Any ⟨range⟩ which is out of order will result in an error message at execution time.

APPENDIX C

COMMAND ABBREVIATION SCHEME

♦ Most editing and debugging commands in the TSOS Interactive FORTRAN system may be abbreviated if the user of the system so desires. This is a scheme for determining the unique acceptable abbreviation knowing the command in its entirety.

ALGORITHM USED

♦ 1. If the command is a single word of three letters, there is no abbreviation.

2. If the command is a single word but not three letters in length, the abbreviation is the first and last letter of the word.

3. If the command is two words or more, the abbreviation is the first letter of each word, ignoring the word AND if it appears in the command.

Table C-1. TSOS Interactive FORTRAN System Command Abbreviations

Command	Abbreviation
#AT	#AT
#CHECK	#CK
#CHECK AND PRINT	#CP
#CHECKPOINT	#CT
#COMPILER CHECK	#CC
#COMPILER CHECK AND PRINT	#CCP
#COMPILER SYNTAX CHECK	#CSC
#COMPILER SYNTAX CHECK AND PRINT	#CSCP
#DEFINITION	#DN
#DELETE	#DE
#DISPLAY VAR	#DV
#DISPLAY VAR NOT CHANGED	#DVNC
#DISPLAY VAR CHANGED	#DVC
#DISPLAY VAR NOT CHANGED SINCE	#DVNCS
#DISPLAY VAR CHANGED SINCE	#DVCS
#DISPLAY STMT	#DS
#DISPLAY STMT CHANGED	#DSC
#DISPLAY STMT NOT CHANGED	#DSNC
#DISPLAY STMT EXECUTED	#DSE
#DISPLAY STMT NOT EXECUTED	#DSNE
#DISPLAY STMT CHANGED SINCE	#DSCS
#DISPLAY STMT NOT CHANGED SINCE	#DSNCS
#DISPLAY STMT EXECUTED SINCE	#DSES
#DISPLAY STMT NOT EXECUTED SINCE	#DSNES
#DISPLAY STMT ID	#DSI
#DISPLAY STMT ID CHANGED	#DSIC
#DISPLAY STMT ID NOT CHANGED	#DSINC
#DISPLAY STMT ID EXECUTED	#DSIE
#DISPLAY STMT ID NOT EXECUTED	#DSINE
#DISPLAY STMT ID CHANGED SINCE	#DSICS
#DISPLAY STMT ID NOT CHANGED SINCE	#DSINCS
#DISPLAY STMT ID EXECUTED SINCE	#DSIES
#DISPLAY STMT ID NOT EXECUTED SINCE	#DSINES
#DISPLAY VAR XREF	#DVX
#DISPLAY VAR XREF CHANGED	#DVXC
#DISPLAY VAR XREF NOT CHANGED	#DVXNC

Table C-1. TSOS Interactive FORTRAN System Command Abbreviations *(Cont'd)*

Command	Abbreviation
#DISPLAY VAR XREF CHANGED SINCE	#DVXCS
#DISPLAY VAR XREF NOT CHANGED SINCE	#DVXNCS
#DISPLAY STMT XREF	#DSX
#DISPLAY STMT XREF CHANGED	#DSXC
#DISPLAY STMT XREF NOT CHANGED	#DSXNC
#DISPLAY STMT XREF EXECUTED	#DSXE
#DISPLAY STMT XREF NOT EXECUTED	#DSXNE
#DISPLAY STMT XREF CHANGED SINCE	#DSXCS
#DISPLAY STMT XREF NOT CHANGED SINCE	#DSXNCS
#DISPLAY STMT XREF EXECUTED SINCE	#DSXES
#DISPLAY STMT XREF NOT EXECUTED SINCE	#DSXNES
#DISPLAY VAR TABLE	#DVT
#DISPLAY STMT TABLE	#DST
#DISPLAY BRANCHES	#DB
#DISPLAY BRANCHES EXECUTED	#DBE
#DISPLAY BRANCHES NOT EXECUTED	#DBNE
#DISPLAY BRANCHES EXECUTED SINCE	#DBES
#DISPLAY BRANCHES NOT EXECUTED SINCE	#DBNES
#DISPLAY LINKAGE	#DL
#DISPLAY CHANGES	#DC
#DISPLAY CHANGES OFF	#DCO
#DISPLAY PROGRAM NAMES	#DPN
#EDIT	#ET
#END	#END
#EXECUTE	#EE
#EXPAND	#ED
#FIND	#FD
#FORGET	#FT
#GET	#GET
#HALT	#HT
#IN	#IN
#INSERT	#IT
#INSERT AND CHECK	#IC
#INSERT AND COMPILER CHECK	#ICC
#INSERT AND SYNTAX CHECK	#ISC
#INSERT AND COMPILER SYNTAX CHECK	#ICSC
#INSERT WITHOUT CHECK	#IWC
#INSERT DATA	#ID
#MOVE	#ME
#NAME	#NE
#NO TRACE	#NT
#NUMBER	#NR
#ORDER	#OR
#PATTERN	#PN
#PREFIX	#PX
#PREP	#PP
#PRINT VAR	#PV
#PRINT VAR NOT CHANGED	#PVNC
#PRINT VAR CHANGED	#PVC
#PRINT VAR NOT CHANGED SINCE	#PVNCS
#PRINT VAR CHANGED SINCE	#PVCS
#PRINT STMT	#PS
#PRINT STMT CHANGED	#PSC
#PRINT STMT NOT CHANGED	#PSNC
#PRINT STMT EXECUTED	#PSE

Table C-1. TSOS Interactive FORTRAN System Command Abbreviations *(Cont'd)*

Command	Abbreviation
#PRINT STMT NOT EXECUTED	#PSNE
#PRINT STMT CHANGED SINCE	#PSCS
#PRINT STMT NOT CHANGED SINCE	#PSNCS
#PRINT STMT EXECUTED SINCE	#PSES
#PRINT STMT NOT EXECUTED SINCE	#PSNES
#PRINT STMT ID	#PSI
#PRINT STMT ID CHANGED	#PSIC
#PRINT STMT ID NOT CHANGED	#PSINC
#PRINT STMT ID EXECUTED	#PSIE
#PRINT STMT ID NOT EXECUTED	#PSINE
#PRINT STMT ID CHANGED SINCE	#PSICS
#PRINT STMT ID NOT CHANGED SINCE	#PSINCS
#PRINT STMT ID EXECUTED SINCE	#PSIES
#PRINT STMT ID NOT EXECUTED SINCE	#PSINES
#PRINT VAR XREF	#PVX
#PRINT VAR XREF CHANGED	#PVXC
#PRINT VAR XREF NOT CHANGED	#PVXNC
#PRINT VAR XREF CHANGED SINCE	#PVXCS
#PRINT VAR XREF NOT CHANGED SINCE	#PVXNCS
#PRINT STMT XREF	#PSX
#PRINT STMT XREF CHANGED	#PSXC
#PRINT STMT XREF NOT CHANGED	#PSXNC
#PRINT STMT XREF EXECUTED	#PSXE
#PRINT STMT XREF NOT EXECUTED	#PSXNE
#PRINT STMT XREF CHANGED SINCE	#PSXCS
#PRINT STMT XREF NOT CHANGED SINCE	#PSXNCS
#PRINT STMT XREF EXECUTED SINCE	#PSXES
#PRINT STMT XREF NOT EXECUTED SINCE	#PSXNES
#PRINT VAR TABLE	#PVT
#PRINT STMT TABLE	#PST
#PRINT BRANCHES	#PB
#PRINT BRANCHES EXECUTED	#PBE
#PRINT BRANCHES NOT EXECUTED	#PBNE
#PRINT BRANCHES EXECUTED SINCE	#PBES
#PRINT BRANCHES NOT EXECUTED SINCE	#PBNES
#PRINT LINKAGE	#PL
#PRINT CHANGES	#PC
#PRINT CHANGES OFF	#PCO
#PRINT PROGRAM NAMES	#PPN
#PROCEED	#PD
#REFER	#RR
#REPLACE	#RE
#REPLACE AND CHECK	#RC
#REPLACE AND COMPILER CHECK	#RCC
#REPLACE AND SYNTAX CHECK	#RSC
#REPLACE AND COMPILER SYNTAX CHECK	#RCSC
#REPLACE WITHOUT CHECK	#RWC
#REPLACE DATA	#RD
#RESTART	#RT
#RETURN	#RN
#SAVE	#SE
#SAVE AND DELETE	#SD
#SYNTAX CHECK	#SC
#SYNTAX CHECK AND PRINT	#SCP

Table C-1. TSOS Interactive FORTRAN System Command Abbreviations *(Cont'd)*

Command	Abbreviation
#TRACE	#TE
#TURN OFF	#TO
#UNSAVE	#UE
#WHEN	#WN

APPENDIX D

PROGRAMMING AIDS

◆ There are a number of useful programming aids available to the user while in the insert mode.

REPLACING STATEMENTS

◆ If while entering statements, the user decides he would like to change certain statements entered previously, he may do so as follows:

1. To replace one or more lines then return to the present line number and continue:

   a. When the system types the next line number, the user types @n*ETX* where n is the line number of the first line to be changed.

   b. The system responds by skipping a line and typing out n as the next line number, and the user types the new line.

   c. He continues replacing, the system incrementing line numbers by 1, until he has replaced as many lines as desired.

   d. When the next line number is typed out the user responds with *@ETX* which returns him to the line at which he typed @n*ETX*.

   Example

```
5.   READ(97,20) A,B
6.   D=A+B
7.   WRITE (99,25) A,B,D
8.   @6
6.   D=A*B
7.   @
8.
```

   Note:

   At this point line 7 still contains the WRITE statement typed in originally; it is not affected by "7. @"

2. To replace all previous lines beginning with a given line number when the user does not wish to return afterwards to his current place (or to change line numbers altogether in case the user wants to leave space for later insertions, for example):

   a. When the system types the next line number the user types @SET n *ETX* where n is the line number of the first line to be replaced.

REPLACING STATEMENTS (Cont'd)

b. The system responds by skipping a line and typing out n as the next line number, and the user proceeds as usual (the system continues to increment line numbers by 1 as usual).

Example

```
5.      READ(97,20) A,B
6.      D=A+B
7.      WRITE (99,25) A,B,D,E
8.      @ SET 6
6.      D=A*B
7.      E=D+A
8.      WRITE (99,25) A,B,D,E
9.
```

INSERTING STATEMENTS

♦ To insert statements between previously entered lines of the current file, the user follows procedure 1 outlined above but this time specifying a line number between the two in question (and a suitable increment in parentheses following the line number if he wants to insert more than one line).

The increment may be any number of the user's choosing up to a maximum of four digits followed by a decimal point followed by four more digits.

Example

```
5.      A=B+C
6.      D=A*F
7.      WRITE (99,20) A,D
8.      @5.1 (.2)
5.1     D=0
5.3     F=G+H
5.5     @
8.
```

DELETING STATEMENTS

♦ If the user decides at some point in the insert mode to delete one or more previously entered lines he proceeds as follows:

When the system types the next line number the user responds with

@D ⟨list⟩

where ⟨list⟩ may be a line number, a range of line numbers, or any combination of these separated by commas. A range of line numbers is two line numbers separated by a hyphen and means those two lines and all lines in between.

DELETING STATEMENTS (Cont'd)

If the user neglects to furnish a ⟨list⟩ or if the ⟨list⟩ contains an error, he gets the error message, "@ PARAMETER IN ERROR" and the system retypes the line number.

Example

Assume the file into which the user is inserting contains lines 1 through 6, 6.1, 6.2, and 7 through 15, and that the system has just typed line number 16:

16. @D10,3,5-8,14

After this statement is entered, the file contains lines 1, 2, 4, 9, 11, 12, 13, and 15, and the system responds with line number *16 again.*

DISPLAYING STATEMENTS

- Two options are available to the user if he wishes to display selected lines while in the insert mode.

Display to the Teletype

- When the system types the next line number, the user types

@P⟨list⟩

where ⟨list⟩ is the list of lines to be printed and is defined above (in "Deleting Statements").

The system responds by typing the specified lines followed by the line number at which the @P command was given. In displaying the lines the line numbers are typed out in full; e.g., line number 8. is typed as 8.000.

Print on the On-Line Printer

- Instead of typing @P⟨list⟩, if the user types

@L⟨list⟩

the specified list of statements is spooled out and will be printed on the on-line printer connected to the computer when the user logs off.

Example

```
 5.      A=1.
 6.      B=2.
 7.      D=3.
 8.      E=4.
 9.      F=5.
10.      G=6.
11.      H=7.
12.      @P8-10,6
 8.0000      E=4.
 9.0000      F=5.
10.0000      G=6.
 6.0000      B=2.
12.
```

APPENDIX Z

INDEX

INDEX (Cont'd)

INDEX (Cont'd)

INDEX (Cont'd)

INDEX (Cont'd)

INDEX (Cont'd)

INDEX (Cont'd)

INDEX (Cont'd)