# DEFINITION AND STANDARD FOR TRAC® T-64 LANGUAGE

Calvin N. Mooers

This volume constitutes Appendix A of
The Beginner's Manual for TRAC® Language

**TRAC STANDARD NO. 1**
**RR-284**
**First Edition 1972**

ROCKFORD RESEARCH, INC.
140½ Mount Auburn Street
Cambridge, Massachusetts 02138

First Edition.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -



- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

TABLE OF CONTENTS

TRAC® STANDARD NO. 1, October 1972

# DEFINITION AND STANDARD FOR TRAC T-64 LANGUAGE

by

Calvin N. Mooers

## 1. INTRODUCTION

### 1.1 ABOUT THIS STANDARD

The designation T-64(1) distinguishes the particular TRAC(2) language described here. This "Definition and Standard for TRAC T-64 Language" shall be referred to as "TRAC® Standard No. 1, October 1972". This Standard is an updating, with removal of certain ambiguities and defects, of the previous defining document(3) which it supersedes.

Standards for TRAC language are developed and maintained by Rockford Research, Inc., and are put forth on the authority of Rockford Research, Inc., where TRAC language was developed.

All TRAC standards and documentation are copyrighted by Rockford

---

(1) "T-64" is a trademark of Rockford Research, Inc.

(2) "TRAC" is registered in the U. S. Patent Office as a service mark of Rockford Research, Inc. It is also a trademark of Rockford Research, Inc.

(3) Calvin N. Mooers, "TRAC, A Procedure-Describing Language for the Reactive Typewriter", Communications of the ACM, v. 9, n. 3, pp. 215-219 (March 1966), copyright © 1966 Rockford Research, Inc.

This Standard defines TRAC language in terms of the actions of the TRAC processor on character sequences written in the language. It is not the purpose of this Standard to describe the particular technical means by which these actions are carried out by the processor.

'reactive typewriter'
'characters'
'function characters'
'format characters'

## 2. TEXT AND THE PROCESSOR

### 2.1 CHARACTERS, STRINGS AND SCRIPTS

TRAC language is a standardized interactive text manipulating language, based on macrogenerator principles, for use with a reactive typewriter. By a 'reactive typewriter'[4] is meant an automatic typewriter, or an equivalent graphic display screen and keyboard device, which operates in conjunction with a computer.

TRAC language is defined in terms of 'characters', where a character is the result of a particular action performed at the keyboard. A character produces a printed or indicated graphic symbol, or causes other effects, such as moving the location of the printing action. The class of characters includes the letters of the alphabet, the numerical digits from ø to 9, the punctuation marks and special signs, the space character, and the various 'function characters' which produce assorted machine or control actions.

Two of the function characters are called 'format characters' in this Standard. These are the carriage return character, indicated by $\overline{\text{CR}}$, and the line feed character, indicated by $\overline{\text{LF}}$.

For some purposes, the ASCII character code, as standardized by the American National Standards Institute, is a useful reference for considering the characters and their codes. However, for purposes of

---

(4) Whenever a new term is introduced in the text of this Standard, it is placed in single quotation marks and, for convenient reference, it is also listed at the top corner of the page.

'reserved characters'
'processor'
'strings'
'substring'
'text'
'script'



this Standard and Definition for TRAC language, the actual codes used to represent the characters play no part.

TRAC language has no 'reserved characters' (i.e., it has no special characters which may not be used in text), although certain characters in an appropriate context have special meanings or effects. The most important group of such characters in TRAC language are the five characters:

# ( , ) '

In addition to these, the eight characters:

$\overline{\text{CR}}$ $\overline{\text{LF}}$ \ @ + - / *

have particular special roles.

Actions specified in TRAC language, when typed at the keyboard, are carried into effect by a TRAC 'processor'. By processor is meant the program running inside the computer connected to the reactive typewriter.

TRAC language deals with sequences of characters called 'strings'. A part of a string is called a 'substring'. Some strings, when typed at the keyboard, specify actions to be taken by the TRAC processor. Other strings are acted upon. The same string, at different times within the processor, may fill either role. Strings in general are called 'text'. A text whose primary purpose is the control of the TRAC processor is called a 'script'.

Thus the TRAC processor, acting in accordance with a TRAC language script, carries out actions on text inside the computer, and from time to time it displays the results of these actions by printing out strings or performing control or format actions at the reactive typewriter.

'expression'
'primitive'
'execution'
'arguments'
'print string'
'mnemonic'

## 2.2 EXPRESSIONS AND PRIMITIVES

Any part of a script which specifies some sort of action is called an 'expression'. One simple form of expression in TRAC language is called a 'primitive'. An example of a primitive is

#(PS,CAT)

When the action specified by such a primitive takes place in the processor, the primitive is said to be carried into 'execution'. Execution of the primitive #(PS,CAT) causes the processor to print out the string CAT at the reactive typewriter.

This primitive has two 'arguments'. which are PS and CAT . These arguments specify the action to be taken. The kind of action is specified by the first argument PS , which stands for 'print string'. The second argument is the actual string of characters to be printed out. The arguments are separated by commas.

The kind of action to be performed by a TRAC primitive is always specified by the first argument, which is a two-letter string called a 'mnemonic'.(5) TRAC T-64 language has a total of 34 different primitives, each having a characteristic action and its own unique mnemonic.

The different TRAC primitives require various numbers of arguments in addition to the mnemonic. These additional arguments provide the additional information needed to carry out the action of the primitive.

If the first argument of an expression is not one of the two-letter

---

(5) "Mnemonic" is pronounced "nee-mon-ick", and it means that the letters in the symbol have been chosen to assist one in remembering what the symbol stands for.

'left parenthesis'
'begin parenthesis'
'right parenthesis'
'end parenthesis'
'corresponding parentheses'
'parenthesis count'



mnemonics, the expression may still prescribe a valid action, as will be defined later.

TRAC language expressions have two modifications, represented by

#( ::: )  and  ##( ::: )

where ::: represents any appropriate sequence of argument strings. As will be defined later, these two modifications cause slightly different kinds of behavior to take place in the TRAC processor.

## 2.3 CORRESPONDING PARENTHESES

Parentheses are the dominant characters determining the syntax of TRAC language. The two parentheses are denoted 'left parenthesis' or 'begin parenthesis' for the character ( , and 'right parenthesis' or 'end parenthesis' for the character ) .

An important concept is that of pairs of 'corresponding parentheses' in a string of characters. For example, the underlined pair of parentheses in the string ABC(DE(FC))XYZ are corresponding parentheses. A pair of corresponding parentheses in an arbitrary string is located by means of the following rules:

1. The string is examined, from left to right, character by character, and a 'parenthesis count' is computed.
2. Initially, the parenthesis count is set to zero. Characters which are not parentheses do not change the parenthesis count.
3. For each left parenthesis encountered, the parenthesis count is increased by one.

'sharp sign character'
'protective parentheses'
'protected string'

4. For each right parenthesis enccountered, the parenthesis count is decreased by one.

5. If the parenthesis count becomes negative, the examination is terminated, and the string does not have a pair of corresponding parentheses.

6. The parenthesis which first causes the parenthesis count to go from zero to +1 is taken as the first member of the pair.

7. The parenthesis which first causes the parenthesis count to go from +1 to zero is taken as the second member of the pair, and the examination is terminated.

The pair of corresponding parentheses of a string which is located by this rule depends upon the starting point taken in the string. By starting at the character A of the example string, the underlined parentheses are found as a corresponding pair. By taking the substring beginning at character D in the example, another pair of corresponding parentheses are found, namely the pair which surrounds the substring FG .

## 2.4 PROTECTIVE PARENTHESES

In a given string with a pair of corresponding parentheses, if the left parenthesis of the pair is not preceded by the 'sharp sign character' # , then the two parentheses are called 'protective parentheses'. Such parentheses have a protective function in the syntax. The substring which is contained within a pair of protected parentheses is called a 'protected string'. Thus in the example string ABC(DE(FG))XYZ , the string DE(FG) is a protected string. On the other hand, if the character C were replaced by the character # , then DE(FG) would not be a protected string.

'workspace'
'scanning pointer'



# 3. SCANNING AND EXECUTION

## 3.1 THE WORKSPACE AND SCANNING POINTER

The manipulations on TRAC language scripts take place in a 'workspace' of the processor. The actions taking place in the workspace can be diagrammed by means of character strings together with various symbolic markers.

The workspace is linear. The left end of the workspace is marked by the symbol =/ while the right end is marked by the symbol /= . In the workspace, there is a 'scanning pointer' represented by the symbol // . The empty workspace, without any characters, but containing the scanning pointer, is represented by:

=//=

The size of the workspace is elastic, and therefore it can expand and hold various character strings between the end markers and the scanning pointer. Thus a workspace holding characters is illustrated by the diagram:

=/ABC//XY#(PS,DOG)/=

Characters in the workspace are analyzed one by one, from left to right. As each character is analyzed at the scanning pointer, an appropriate action is taken according to a set of TRAC language rules. Some characters are simply moved to the left of the scanning pointer. Other characters which have an active syntactic role are moved onto an upper line to the left of the scanning pointer:

=/ABCXY#( PS, //DOG)/=

'syntactic markers'
'scanning diagram'
'neutral strings'
'active string'

These "upstairs" characters provide the 'syntactic markers' which control execution of the primitives.

Other kinds of transformations take place. In some cases, strings are brought into the workspace, or are moved out of the workspace, or are made to disappear. These various actions are all controlled by the primitives as they are encountered by the scanning action in the workspace.

The actions in the workspace can be shown and precisely specified by a 'scanning diagram' which shows at every stage the active and neutral strings, the syntactic markers, and the location of the scanning pointer.

For example, the process of scanning may begin with the diagram:

=//#(PS,THIS IS TEXT)/=

At a later stage of the scanning action, the scanning diagram will be:

#( ,
=/ PS THIS I//S TEXT)/=

In this second scanning diagram, the syntactic markers which were found during the scanning process are shown placed on the upstairs line and to the left of the scanning pointer. The character group #( and the comma , are the two syntactic markers shown. These syntactic markers separate into substrings the characters standing on the "downstairs" line.

Two substrings are shown. They are PS and THIS I . These substrings are called 'neutral strings', because they are to the left of the scanning pointer and have already been scanned and "neutralized".

The single string standing to the right of the scanning pointer, and extending up to the right-hand end marker, is S TEXT) . This string is called the 'active string' because it may produce various actions in the processor when it is scanned.

'transformation rules'



## 3.2 THE SCANNING TRANSFORMATIONS

The action of a TRAC processor depends upon analysis of the strings in the workspace. The analysis takes place at the location of the scanning pointer according to a definite list of TRAC language 'transformation rules' which are given below.

At any given position of the scanning pointer, the processor makes use of the list of rules to determine what action to take. In effect, the processor runs down the list of rules to find the first rule in the list which is applicable. It then applies this rule to make a transformation, or to take some other action. Usually the scanning pointer will be moved one character forward.

Then, unless otherwise specified, the scanning action continues, character by character, with the processor going to the head of the list for each character to find the particular rule to apply. All the actions of the processor and of the scanning action are determined by the following transformation rules.

Rule 1. //( ::: ) becomes ::: //

where ::: stands for any protected string.

Rule 2. //( with no corresponding parenthesis standing to the right becomes =//=

By this rule, when no corresponding parenthesis can be found to the right of the scanning pointer, the entire workspace is cleared.

Rule 3. $/\!/\#\#($ becomes $\overset{\#\#(}{/\!/}$

The syntax group ##( goes to the upstairs line of the diagram.

Rule 4. $/\!/\#($ becomes $\overset{\#(}{/\!/}$

Rule 5. $/\!/\#$ becomes $\#/\!/$

An isolated sharp sign # is treated as an ordinary character.

Rule 6. $/\!/,$ becomes $'/\!/$

Rule 7. $/\!/\overline{\text{CR}}$ becomes $/\!/$

The carriage return format character disappears.

Rule 8. $/\!/\overline{\text{LF}}$ becomes $/\!/$

The line feed format character disappears.

Rule 9. $/\!/)$ Go to Rule 13 to determine the action.

Evaluation of a primitive usually occurs from this rule.

Rule 10. $/\!/\text{A}$ becomes $\text{A}/\!/$
where A stands for any character not treated by the preceding rules.

Rule 11. $/\!/=$ becomes $=/\!/=$

Rule 12. $=/\!/=$ becomes $=/\!/\#(\text{PS},(\overline{\text{CRLF}}))\#(\text{PS},\#(\text{RS}))/=$

The remaining rules, Rules 13 through 20, will be found in the two following sections.

The script which is placed in the empty workspace by Rule 12 is called the 'idling program'. Rule 12 permits the processor to begin a new

sequence of actions after it has executed all the expressions in the workspace. The idling program causes the processor to print out the format characters $\overline{\text{CRLF}}$ . This immediate printing action provides a 'typewriter signal' to the person at the keyboard, and it indicates that keyboard typing can resume.

## 3.3 EXECUTION OF EXPRESSIONS

Rule 9, where the scanning situation is *//)* , results in the prescription, "Go to Rule 13 to determine the action." Rule 9 in actuality is an indication that some TRAC language primitive may be ready for execution.

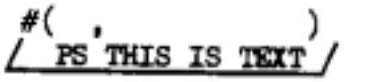
To execute an expression, the processor first looks to the left of the scanning pointer, in the upstairs line of the scanning diagram, to find the nearest (right-most) instance of one or the other of the syntactic groups *#(* or *##(* . If an instance of either group can be found, the right-hand parenthesis in front of the scanning pointer is moved upstairs, and the scanning pointer temporarily disappears. The processor now marks out the 'scope of execution' of the expression.. The scope extends at the left from the nearest upstairs *#(* or *##(* group, to the right-hand parenthesis which has just been moved upstairs at the right. To illustrate, the scope of execution in the previous example is:

$$=/\overbrace{{}^{\#(}\ ,\ {}^{)}\ \text{PS THIS IS TEXT}}^{\text{scope of execution}}/=$$

In order to diagram the action of the processor, as it is executing

'execution diagram'
'default'

an expression, the scope of execution is marked out by means of slant characters, with one slant character being placed at the beginning of the scope, and another being placed at the end of the scope, and with the two slant characters being connected by an underline:




This diagram is called the 'execution diagram' of the primitive. In this notation, one can consider that the scanning pointer has been "enlarged" so as to indicate the scope of the expression to be executed.

In this representation, with the execution diagram, the syntactic markers which are effective for purposes of execution of the primitive are now all in the upper line. They are:

#( , )

These syntactic markers separate the argument strings of the primitive, of which there are two in this example:

PS THIS IS TEXT

The execution diagram displays all the information available in the workspace to be used in specifying the action of the processor when it executes an expression or a primitive. It should be noted that other strings will ordinarily be present in the workspace to be left and to the right of the scope marker /_____/ of the execution diagram.

In some cases, neither syntactic group #( nor ##( can be found upstairs to the left of the scanning pointer. In such a case, there is no actual expression to be executed, and so a 'default' action must take place. The default action is to clear the workspace entirely.

If the default action is not required, then an execution diagram can be formed, and some sort of defined action can always take place.

The default action, and the formation of the execution diagram, take place in accordance with the following two rules:

Rule 13. //) with no syntactic marker upstairs and to the left becomes =///=

Rule 14. #( or ##( ::: //) becomes "execution diagram"
where ::: stands for any string and any upstairs-comma syntactic markers standing between the nearest syntactic group and the scanning pointer, and "execution diagram" stands for the complete execution diagram.

## 3.4 THE VALUE STRING FROM A PRIMITIVE

The execution diagram, with its complete marking out of the arguments of an expression or primitive, provides the information necessary for the processor to act on the expression. In most cases, the expression is a TRAC language primitive. A default case, the 'default call', is defined later in this Standard to cover the cases where the first argument is not a mnemonic.

Execution of a primitive according to the execution diagram will sometimes produce a character string called a 'value string' which is the 'value' of the primitive. With certain other primitives, no character string is produced. In such cases, the value string can be thought of as

'null string'
'null value primitives'
'resume marker'
'default call'

consisting of a string with no characters, the 'null string', which is then taken as the value string of the primitive. Such primitives are called 'null value primitives'. Null value primitives are employed because of the actions or side effects they produce, rather than for their value strings.

The manner in which the value string is handled in the processor is completely represented by further transformations that are made in the scanning diagram. At the time of execution of a primitive, the complete content of the execution diagram is deleted from the scanning diagram, and the value string from execution of the primitive is inserted in its place. Since the workspace is "elastic", the surrounding strings in the workspace are moved to the left or right to make room, or to close up blank spaces, as may be necessary.

For diagrammatic purposes, the scope of the value string is now temporarily marked out by slant characters placed at the head and tail of the inserted value string. Furthermore, a 'resume marker' consisting of an underline is placed at the foot of either the left-hand or the right-hand slant character: / .

Placement of the resume marker depends upon whether the execution diagram used the syntactic group #( or ##( . If the syntactic group of the execution diagram was #( , then the resume marker is placed under the left-hand slant character in the scanning diagram. If the group was ##( , then the resume marker is placed under the right-hand slant character.

Two exceptions to this rule take place. The first exception occurs with the 'default call' in which the first argument is not one of the

two-letter mnemonics for a TRAC language primitive. With the default call, the resume marker is always placed under the left-hand slant marker irrespective of the #( or ##( syntactic group.

The second exception occurs with the primitives which have a 'default argument'. When the default argument furnishes the value string of the primitive, the resume marker is also always placed under the left-hand marker.

The purpose of the resume marker is to indicate where the scanning pointer will resume scanning. The purpose of the right- and left-hand slant characters of the scanning diagram at this stage is merely to indicate clearly the place where the value string has been inserted into the workspace.

In the next stage of the process, the scanning pointer replaces the resume marker and its associated slant character. The other slant character is simply deleted. Scanning now resumes in the workspace at the scanning pointer according to the transformation rules.

As is evident from this discussion of the resume marker, the value string usually resulting from a primitive with the #( syntactic group will always be scanned. On the other hand, the value string resulting from the primitive with ##( will usually not be scanned.

The handling of the value strings produced by the primitives, the resume marker, and the scanning pointer is described exactly by the following transformation rules:

Rule 15. "execution diagram
for default call" becomes / "value string" /

Rule 16. "execution diagram in which the default argument of a primitive furnishes the value string" becomes ∠ "value string" /

Rule 17. "execution diagram with #( " becomes ∠ "value string" /

Rule 18. "execution diagram with ##( " becomes / "value string" ∠

The scanning pointer re-appears according to the rules:

Rule 19. ∠ "value string" / becomes // "value string"

Rule 20. / "value string" ∠ becomes "value string" //

The scanning action with the scanning pointer can now resume with Rule 1.

It should be noted that the six rules, Rules 15 through 20, could be reduced to four rules. However, for descriptive and diagrammatic purposes, the rules as they are presented are believed to be the most helpful for understanding TRAC language.

The following example illustrates the diagramming of the scanning process, the execution of an expression with the formation of a value string, and the other matters described:

=//#(PS,THE #(CL,XY) DOG)/=

=/ #( , PS THE //#(CL,XY) DOG)/=

=/ #( , #( , PS THE CL XY//) DOG)/=

$$=/^{\#(\ ,\ }\ \text{PS THE}\ \underline{/^{\#(\ ,\ )}\ \text{CL XY}\ /}\ \text{DOG})/=$$

At this point, the execution diagram for the CL primitive is fully formed. Assume that the value string produced by the CL primitive being executed is the string BROWN :

$$=/^{\#(\ ,\ }\ \text{PS THE}\ /\text{BROWN/}\ \text{DOG})/=$$

$$=/^{\#(\ ,\ }\ \text{PS THE}\ //\text{BROWN DOG})/=$$

At this stage, scanning is resumed by the transformation rules, beginning with Rule 1.

In the scanning diagram, any auxiliary information about the action taking place, such as any strings which are typed in or printed out by the processor, can be placed as comments at the right in the diagrammatic representation:

Printed out: THE BROWN DOG

Typed in: CAT'

Value string: BROWN

# 4. OTHER MATTERS

## 4.1 DEFAULT CASES FOR ARGUMENTS

Most of the 34 primitives in TRAC language are defined with respect to some specific number of argument strings. For example, one of the primitives, the 'define string' primitive, requires three arguments. Some primitives take a variable number of arguments. For the primitives which are defined for a specific number of arguments, there are default rules for situations in which there are too many or too few argument strings in the execution diagram.

When the execution diagram shows too many arguments according to the definition of a primitive, the excess arguments at the right-hand end of the argument sequence are ignored at the time of execution.

It is important to note that although these excess argument strings are ignored at the time of execution, they are scanned prior to the formation of the execution diagram. Accordingly, all expressions inside a primitive, even those which appear to be in excess argument positions, must be considered in determining the total action of the TRAC processor.

When the execution diagram does not show as many arguments as are required by the action of the particular primitive, another default action is defined. In this case, at the time of execution, the missing arguments are taken to be null strings, unless it is otherwise specified by the definition of the particular primitive. For example, in the expression #(DS) , the definition requires that there be three arguments. Therefore,

at the time of execution, the two missing arguments are automatically supplied as null strings.

## 4.2 MNEMONIC DESIGNATIONS OF THE PRIMITIVES

Each of the 34 TRAC language primitives has a full name like 'define string'. Each primitive also has a two-letter 'mnemonic', such as DS in this case, which is used in TRAC language scripts and expressions to designate the desired action of the primitive. (See Table on page 82.)

The two-letter TRAC mnemonic always occurs as the first argument in the execution diagram for each of the TRAC language primitives.

Where the reactive typewriter keyboard has an upper and lower case capability, the TRAC processor will accept any combination of upper or lower case letters for the mnemonic of a primitive. Thus DS , Ds , dS , and ds all have the same action. This kind of disregard of the case distinction for the alphabetic characters takes place only for the mnemonics of the 34 TRAC primitives, and only when the mnemonics are in the first argument location. In all other situations, the case distinction of the characters is retained. In particular, if the first argument of an expression has only two characters, but it is not a mnemonic for a primitive, the case distinction of the characters is retained.

## 4.3 OVERFLOW OF THE CAPACITY OF THE PROCESSOR

Although the workspace of the processor is elastic, and can handle strings of varying lengths, it will in general have an upper limit to its capacity. Too many long strings will cause the processor to 'overflow'. This can happen, for instance, when new strings are typed in from the

<SCE> 'string capacity exceeded'
'forms'
<SCA> 'string capacity alert'
'panic stop'

keyboard, or if strings are duplicated in the course of the action of the processor. Whenever all the areas of the processor devoted to the workspace and to the storage of strings become filled, then the ordinary action of the processor cannot continue and an emergency action of the processor must occur. What happens is that the processor erases all the strings in the workspace and then prints out the emergency diagnostic <SCE> , meaning 'string capacity exceeded'. However, strings held in storage (TRAC language 'forms') are not affected by this erasing action.

Because any work in the workspace is erased by this emergency action, it is desirable to avoid such an overflow situation. The processor gives a warning of an impending overflow situation by printing out the warning <SCA> , meaning 'string capacity alert', whenever the free capacity of the processor becomes less than 1ØØ characters. After the string capacity alert diagnostic has been given, it is not again printed out until after the idling program is again reloaded and another warning is required.

When the <SCA> diagnostic appears, it is usually possible to take some action at the keyboard which will remedy or avoid the potential overflow situation.

## 4.4 THE PANIC STOP

Through inadvertence, it sometimes happens that an endless sequence of actions is caused to take place in the TRAC processor. In such a situation, the processor is not responsive to any ordinary activity at the keyboard. No further work can be done until the endlessly continuing actions can be made to stop.

Therefore, there is a keyboard action which can be taken to override the normal action of the processor and to cause the endless action to stop. This emergency action is called the 'panic stop'.

Unfortunately, because of technical reasons having to do with the great variability in physical hardware of typewriters, graphic displays, computers, and communication lines, it is not possible in this Standard to designate a standard panic stop key action. The user of this Standard must refer to the technical material dealing with his particular TRAC processor and its computer to learn how to make a panic stop.

After the panic stop action has been taken, the processor stops the scanning and execution of the strings currently in the workspace. It clears the workspace, and reloads the idling program. In particular:

1. The output printing of the print string PS or print form PF primitives, or of the trace action, is halted immediately.
2. Any read-in action of a read string RS or read character RC primitive is set to the state of completion.
3. All other actions of the particular primitive currently being executed are completed.
4. The scanning and execution action then stops.
5. The workspace is cleared.
6. The processor action resumes with the reloading of the idling program.

Strings in storage (TRAC language forms) are not disturbed. The processor is now again available for normal use, and the circumstances which caused the endless activity should be avoided.

# 5. DEFINITIONS OF THE PRIMITIVES

## 5.1 CONVENTIONS USED IN THE REPRESENTATION OF THE ARGUMENTS

In the definition of the 34 TRAC T-64 language primitives which follow, certain conventions are used in showing the argument strings. The definitions are based upon the arguments as they appear in the execution diagram of the primitive. The syntactic markers of the upstairs line of the execution diagram are omitted in the definitions. Each primitive generally has a number of different arguments, and these arguments are indicated by 'argument designators', separated by commas. In actual use, the argument designators are replaced by some specific character strings of the appropriate kind.

The first argument shown in each definition is always the specific two-letter mnemonic for the primitive.

The second and succeeding arguments, if there are any, are indicated by argument designators such as N1 or T3 and so on. Several different kinds of argument strings are distinguished in these definitions, and for each kind of argument, a different kind of argument designator is used.

The first kind of argument string is a string of characters which is the name string for some stored string. Such name strings have argument designators N1 , N2 , etc. for the different names required in a primitive.

If an indefinite number of strings of the same kind is allowable in the primitive, the ellipsis symbol consisting of three dots ... is used to indicate that there may be additional strings of the same kind. For

example, the symbolism N1,N2, ... indicates that an indefinite number of name strings is allowed at this point. By "indefinite number" is meant that the number may be as few as no strings at all, one string, or some large number of strings.

The second kind of argument string is the kind which contains decimal numerical digits. Such strings are used in the arithmetic primitives. These strings are designated by D1 , D2 , and so on.

Another kind of string is used in the 'Boolean' primitives, which will be explained later. These strings are designated by B1 , B2 , etc.

The 'default argument' string, which will also be explained, is designated by Z .

Argument strings which do not fall in any of the previous categories, and thus are ordinary text strings, are designated by T1 , T2 , etc.

The arguments of a primitive are designated sequentially as the first, second, third, etc., arguments. The first argument of a primitive is always the two-letter TRAC language mnemonic for the primitive.

## 5.2 PRIMITIVES FOR INPUT AND OUTPUT CONTROL (PS, RS, CM, RC)

Four primitives provide for the passage of text back and forth between the reactive typewriter (the keyboard and display device) and the TRAC processor. Of these, the two primitives 'print string' and 'read string' with mnemonics PS and RS are the most important. These primitives are now defined.

'print string'
'read string'
'meta character'
'full duplex'

Print String PS,T1

Two arguments. Null value string. Execution of this primitive prints out at the reactive typewriter the string of characters of the T1 argument.

- - -

The third argument of this primitive is unassigned, and so it may be used as a convenient place to get rid of unwanted character strings.

Read String RS

One argument. Execution of this primitive produces a value string consisting of the sequence of characters typed at the reactive typewriter keyboard up to the point of occurrence in the sequence of the current meta character. The meta character is deleted from the value string and is discarded.

- - -

The 'meta character' is usually taken as the apostrophe ' , but other characters may be used in its place. When execution of the read string primitive begins, all further action of the processor is deferred, with the exception of the actions required to transfer characters from the keyboard to the workspace. Upon the receipt of the meta character, the transfer is ended, the value string is completely transferred, and the scanning action of the processor resumes.

In a computer and keyboard-display system in which simultaneous keyboard and printing operations do not interfere (i.e., those which provide 'full duplex' operation), the input string for each RS primitive begins with the first keyboard character not previously read in, and the input string extends to the next occurrence of the meta character. Because

'half duplex'
'single-character correction'
\ 'reverse slant'
'restart input correction'
@ 'commercial at'



there may be simultaneous output printing (from a PS ) and input typing (for a RS ), the characters which are being typed at the keyboard are stored and are not displayed until the RS primitive comes into execution.

In a computer and keyboard-display system in which simultaneous keyboard and printing operations do interfere (i.e., those which provide 'half duplex' operation), the input string begins with the first character typed after the processor begins execution of the RS primitive and extends up to the next meta character. Characters typed prior to the time that the processor begins execution of the RS primitive are ignored by the processor.

TRAC processors provide two correction characters which are active during the input process with the RS primitive. The first is the 'single-character correction' character. This correction character causes deletion of any previously-typed character -- other than another correction character or the meta character. For typewriters having the ASCII graphic character set, the 'reverse slant' character \ is used for single-character correction.

Two adjacent reverse slant characters have the effect of deleting two immediately preceeding ordinary characters, and so on. If more reverse slant correction characters are typed in than are applicable to the input string, the extra correction characters are ignored.

The second correction character is the 'restart input correction' character. With keyboards using the ASCII code, the 'commercial at' character @ is used for the restart input correction character. It

causes deletion of all the preceding characters of the input string of the RS primitive.

Both of the correction characters act only on the string characters which precede the correction characters. After a correction character has been used, the desired correct replacement character or string is then typed in. Any combination of single-character and restart input correction characters may be typed in the input sequence. Then when the meta character is finally typed, all the corrections are made (from left to right) and the final corrected string becomes the value string for the RS primitive.

The 'change meta' primitive with the mnemonic CM provides the capability for changing the assignment of the meta character:

Change Meta CM,T1

> Two arguments. Null value string. Execution of this primitive changes the meta character to the first character of the T1 argument string. If the T1 argument is a null string, the meta character is not changed. The meta character is also not changed if the first character of the T1 argument is one of the following characters: sharp sign # , begin parenthesis ( , or comma , .

- - -

The final proviso of the definition prevents the user from inadvertently "locking himself out" of the processor, since any of these three characters used as the meta character would prevent any primitive whatsoever from being entered from the keyboard.

It should be noted that the "first character" of the T1 argument

string is the first character after the T1 argument string has been completely scanned and formed into a string in the execution diagram. Consequently, in order to change the meta character to the carriage return character $\overline{CR}$, the following expression with protective parentheses must be executed: #(CM,($\overline{CR}$))' .

The 'read character' primitive with mnemonic RC reads a single character from the keyboard, and it does not require the meta character for the processor to resume its action.

Read Character RC

One argument. Execution of this primitive produces a single-character value string consisting of the next character typed at the reactive typewriter.

- - -

This primitive will accept any character, including the two correction characters and the meta character. When execution of this primitive begins, further action of the processor is deferred until a character is received from the reactive typewriter keyboard, and then the scanning action of the processor resumes.

The meaning of "next character" for this primitive is taken to be consistent with the meanings for the RS primitive in regard to the half- and full-duplex distinctions. The character typed at the keyboard is printed or displayed when RC is executed.

## 5.3 COMPUTERS REQUIRING AN ENTER CHARACTER

Some computers and keyboard-display systems require the use of an

extra 'enter character' or keyboard signal in order to transfer the string typed at the keyboard to the processor. Unfortunately, the requirement for an enter character seriously degrades the entire input behavior of a TRAC processor, and makes the read character primitive RC almost useless.

The utility of the RC primitive is substantially destroyed by the requirement for the enter character because one of the uses of the RC primitive is to cause the processor to resume its action with the receipt of any single character from the keyboard. For example, in an instructional use, either Y (for "yes") or N (for "no") may be used as a keyboard response. The RC primitive would accept either single character as a response, and the processor would resume its action. When there is a requirement for the enter character, the additional enter character must be provided along with the Y or N character in order to cause the processor action to resume.

The utility of the read string primitive RS is also seriously degraded by those computers and keyboard-display systems which require the use of an enter character. In order for the TRAC language meta character to get to the processor from the keyboard, so the processor can act upon it, the enter character must also be typed as the final action at the keyboard. Such a redundant action is a nuisance at best, and should be technically unnecessary.

The degrading effect of the enter character is particularly insidious in those computers or keyboard-display systems which have a permanent assignment of the RETURN key (the $\overline{\text{CR}}$ character) or the NEW LINE key (the pair of characters $\overline{\text{CR}}$ and $\overline{\text{LF}}$ ) as the enter character. It is

possible to change the assignment of the TRAC processor meta character to be the same as such an enter character by means of the change meta primitive CM . Superficially such a change would appear to make it unnecessary to press two keys to terminate a string and to send it to the processor. Unfortunately, it does not solve the problem. Instead, it creates a new problem with a very undesirable logical effect. The reason for this is that the meta character controls the time at which the TRAC processor resumes its scanning action. Therefore when a executable TRAC script consisting of several lines of text is to be typed in, each line of the script would then go into immediate execution as the RETURN or NEW LINE enter character is typed in order to go to a new line. Such piecemeal execution of scripts, line by line in this manner, would generally lead to completely erroneous results.

Therefore, with computers which are limited to the necessity of an enter character, there is no satisfactory alternative to the use of both the TRAC language meta character and the computer system's enter character at the final end of each input string.

A further complication with many computer systems occurs as a consequence of the "editing" actions which take place with regard to the RETURN or NEW LINE when they are used as the enter character. Some computer-keyboard systems both delete and add line feed $\overline{\text{LF}}$ characters, depending upon the context, as the strings are passed to the processor. In addition, the enter character may move the point of printing at the reactive reactive typewriter or display unit, but since the enter character is actuated following the TRAC language meta character, it may not be a part

'form'
'text string'
'name string'
'forms storage'
'define string'

of the character string received at the processor. This may cause a discrepancy between what is displayed at the reactive typewriter and what actually has been delivered to the TRAC processor. Because there are a baffling variety of conventions used in different computers and keyboard-display systems in regard to the enter characters and the editing actions, no standard prescription for corrective action can be made in this TRAC standard.

## 5.4 PRIMITIVES FOR STRING STORAGE AND DELETION (DS, DD, DA)

The TRAC processor provides for the storage of strings in the processor but outside the workspace area. In order that such strings may be retrieved, each is given a name at the time it is stored.

The complete object which is stored is called a 'form', and it consists of the 'text string', its 'name string', and certain internal markers and pointers which will be defined. The storage region in the processor which is used for such storage, together with its collection of forms, is called the 'forms storage' of the processor. Depending upon the size of the computer in use, the total storage capacity of the forms storage area is generally greater than several thousand characters.

The first primitive in this group is used for the creation of TRAC forms. Because it attaches a name to a string (i.e., because the meaning of the supplied name is defined to be the stored string), the primitive is called 'define string', and it has the mnemonic DS :

'define string'
'form pointer'
'delete definition'
'delete all'



Define String DS,N1,T1

Three arguments. Null value string. Execution of this primitive creates a form in forms storage whose name string is the N1 argument string, and whose text string is the T1 argument string. At the time of creation of the form, the form pointer is placed at the head of the text of the string, preceding the first character. If a form with the same name is already in forms storage, this previous form is deleted when the new form is set up.

- - -

The manner of use of the 'form pointer' will be defined in subsequent primitives.

With the DS primitive, a valid form is created even though one or both of the N1 or T1 arguments strings are null strings. The null string is a perfectly valid name string. A form may also have a non-null name string, but have a null text string.

If execution of the DS primitive with the supplied argument strings would cause an overflow situation, the string capacity exceeded <SCE> diagnostic is printed out at the reactive typewriter, the workspace is cleared, the old form is not deleted, and the new form is not entered into the forms storage. If only the <SCA> string capacity alert diagnostic is given, the new form is set up as usual.

The two following primitives, 'delete definition' and 'delete all', are used to delete forms from forms storage:

'delete definition'
'delete all'
'segment string'
'text macros'

Delete Definition DD,N1,N2, ...

Two or more arguments. Null value string. If the N1 argument is missing, no action is taken. Execution of this primitive deletes from forms storage the forms whose name strings are the same as the N1, N2, etc., argument strings. If a name given in this primitive does not correspond to any form in forms storage, no action is taken with respect to that name.

- - -

Delete All DA

One argument. Null value string. Execution of this primitive deletes all forms held in forms storage.

- - -

The delete all primitive DA is useful because it is able to clear the forms storage of forms whose names might not be known at the time.

The expression #(DD) does nothing. Deletion of a single form which has a null string for a name string is accomplished by the expression #(DD,) .

## 5.5 THE PRIMITIVE FOR THE CREATION OF TEXT MACROS (SS)

The 'segment string' primitive with mnemonic SS modifies the text of a form in forms storage so that at a later time, when a copy is made of the form, it is possible to insert new strings at specified locations in the text of the copy. Text strings which allow insertions of new strings in this manner are called 'text macros'. The SS primitive is one of the most important primitives in TRAC language.

'segment string'
'segment gap markers'
'ordinal number'



Segment String SS,N1,T1,T2, ...

Three or more arguments. Null value string. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form cannot be found, nothing happens and execution of the primitive is completed. If the form is found, the form pointer of the form is restored to the position at the head of the form, preceding the first character or segment gap marker.

Each of the T1, T2, etc., argument strings is compared in sequence, from left to right, with the current state of the text of the form. When a match is found between an argument string and some substring in the current state of the text of the form, the matching substring of the form is deleted and a segment gap marker with ordinal number the same as the ordinal number of the argument string is inserted in its place. The comparison is then resumed at the right of the inserted marker, using the same argument string. When the comparison is completed for one argument string, the next argument string is taken. Arguments beyond the T128 argument are ignored. A null argument string makes no match to the text in the form, not even to a null string. A substring in the text of the form containing one or more segment gap markers does not match any argument string.

At the end of the process, the form pointer is left at the head of the form, and the modified form is replaced in forms storage in the same sequential location that it had before execution of the primitive.

- - -

The 'segment gap markers' mentioned in this definition are "invisible" markers placed in the text of the form. Their use is described later. The 'ordinal number' of a segment gap marker is an integer number, such as

1, 2, 3, etc., used to distinguish which argument string of the SS primitive caused the different segment gap markers to be placed in the modified text of the form. For instance, the argument string T1 results in segment gap markers of ordinal number 1, the argument string T2 gives markers of ordinal number 2, and so on.

As the text of the form is modified, as matches occur, the text is adjusted to the left or right to eliminate any blank spaces.

This primitive may be used on forms which already contain segment gap markers. The old markers are not deleted, and the new markers have ordinal numbers which correspond to the ordinal numbers of the arguments of the new SS primitive.

A sequence of null strings may be used in the T1, T2, etc., arguments when it is desired to generate segment gap markers having ordinal numbers greater than 1, 2, etc. For example, the primitive

#(SS,XX2,,,,CAT)

will result in the occurrences of the substring CAT in the form XX2 being replaced by segment gap markers of ordinal number 4.

## 5.6 THE CALL PRIMITIVE (CL)

The 'call' primitive with mnemonic CL is the main primitive used for copying forms from storage into the workspace. In addition, if the form has been modified by the SS primitive to make it into a text macro, the CL primitive is able to insert new strings into the locations marked by the segment gaps as the text of the form is copied into the workspace.

Call CL,N1,T1,T2, ...

Two or more arguments. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form cannot be found, this primitive has a null value string, and the primitive does nothing.

If the form is found, the value string of the primitive is generated from the text of the form and the arguments T1, T2, etc., of the primitive. Beginning at the point in the text marked by the form pointer, the characters to the right of the pointer are copied one by one into the value string of the primitive. When a segment gap marker is encountered in the text of the form, the argument of the primitive is located which corresponds to the ordinal number of the segment gap marker, and in the value string of the primitive, the segment gap marker is replaced by a copy of this corresponding argument string. The form pointer is not moved.

- - -

In this definition, argument T1 corresponds to the segment gap marker with ordinal number 1, argument T2 corresponds to the ordinal number 2, and so on. The text of the form, the segment gap markers, and the location of the form pointer are not modified by the action of this primitive. Segment gap markers which have no corresponding argument strings are replaced in the value string by null strings. Any additional argument strings T1, T2, etc., with no corresponding segment gap markers in the form are ignored.

Note that the call primitive CL may copy out only a part of the form, since the value string of the primitive begins with the character to

'procedures'
'formal variables'
'parameters'
'default call'

the right of the form pointer, and the form pointer may be at any place in the form. Furthermore, the CL primitive does not move the form pointer as a part of its action.

The CL primitive merely "copies" characters from the form in forms storage, and consequently it does not remove the characters from the form in performing its action. The character string in the text of the form is left intact.

Some additional terminology is useful in connection with executable scripts having segment gap markers for data insertion. Such scripts, when stored as forms, are called 'procedures'. In creating a procedure, the argument strings T1, T2, etc., used in the SS primitive to produce the segment gap markers, are called the 'formal variables' of the original script. The corresponding argument strings T1, T2, etc., of the CL primitive, when it is used to call or initiate the action of the procedure, are called the 'parameters' of the call to the procedure.

## 5.7 THE DEFAULT CALL

When the first argument in the execution diagram is not a two-letter TRAC language mnemonic, the 'default call' action takes place. The default call is not a primitive.

In all respects, except for the designations of the arguments and for the placement of the resume marker, the action of the default call is the same as the action of the regular call primitive CL .

In the regular call, the arguments are represented by:

CL,N1,T1,T2, ...

The corresponding default call has the arguments:

N1,T1,T2, ...

and the CL mnemonic is missing. The name string in the argument N1 must not be any of the two-letter mnemonics for a primitive recognized by the processor (i.e., it cannot be any of the mnemonics for a primitive in the TRAC T-64 language).

With a default call, the value string is generated exactly as with the corresponding regular call, but at the end of the action, the resume marker is always placed at the left end of the value string generated by transformation Rule 15. Accordingly, the value string of a default call is always acted upon by the scanning action of the processor.

Default Call  N1,T1,T2, ...

In the case that the N1 argument string is not one of the two-letter mnemonics recognized by the processor, the expression is executed according to the default call action.

The value string produced by the default call action is exactly the same as if the arguments CL,N1,T1,T2, ... had been in the execution diagram, but in all cases the resume marker is placed at the left end of the value string produced.

- - -

## 5.8 THE PARTIAL CALL PRIMITIVES (CR, CC, CS, CN, IN)

The partial call primitives are used to copy out portions of the text strings of forms, and also to move the form pointer. The location of the text that is copied out is determined in part by the current location of the 'form pointer'.

The form pointer may be located anywhere among the text characters or

segment gap markers of a form. When the pointer is in the middle of a piece of text, the form pointer is considered to lie between two adjacent characters, or between a character and a segment gap marker, or between two segment gap markers. Thus, the form pointer cannot point to any specific character or marker -- instead it must lie immediately to the left or to the right of some character or marker.

When a form is created, the form pointer is set to the head of the form, that is, it is set to a location just preceeding the first character or segment gap marker of the form. Whenever the segment string primitive SS acts upon the form, the form pointer is always reset to the head of the form.

A primitive whose purpose is to bring the form pointer to the head of the form is the 'call restore' primitive with mnemonic CR :

Call Restore CR,N1

Two arguments. Null value string. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form cannot be found, this primitive does nothing. If it is found, the form pointer of the form is moved to the head of the form.

- - -

The four partial calls which are defined below usually produce value strings. However, if a form is found, and for some reason (as defined for each primitive) the usual value string of the primitive cannot be produced, each of these primitives has a default action.

In the default action, the value string of the primitive is provided by the 'default argument' string of the primitive designated by Z in the

definition. By transformation rule 16, when the Z default argument furnishes the value string, the resume pointer is always set to the left-hand end of the value string. Accordingly, irrespective of whether the execution diagram of the primitive has a #( or ##( syntactic group, the scanning action always acts upon the default value string.

In the definitions of the partial calls, the segment gap markers are not "characters". Accordingly, the segment gap markers produce no characters in the value string when a portion of a form is copied into the value string of the primitive. Where a match of character substrings is required, a substring containing a segment gap does not match any character string.

Single characters of a form can be copied out, one at a time, by means of the 'call character' primitive with mnemonic CC .

Call Character CC,N1,Z

Three arguments. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form cannot be found, this primitive has a null value string, and the primitive does nothing.

If such a form is found, the processor attempts to locate a character to the right of the form pointer, skipping over any segment gap markers in search for such a character.

If such a character cannot be found, the form pointer is set to the right-hand end of the form, and the Z argument supplies the value string of the primitive.

If such a character is found, the value string of the primitive consists of a copy of that character. The form pointer is reset to the location just to the right of the character which was found and copied.

- - -

'segments'
'call segment'

Setting the form pointer to the right-hand end of the form means that the form pointer is set to the location just to the right of the right-most character or of the right-most segment gap marker of the form.

The CC primitive produces either a character or the Z argument if the form is present; otherwise, it has no action.

The substrings of a form, which are marked out by the segment gap markers, are called 'segments'. The 'call segment' primitive with mnemonic CS allows these segments of text to be copied out one at a time.

Call Segment CS,N1,Z

Three arguments. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form cannot be found, this primitive has a null value string, and the primitive does nothing.

If such a form is found, and the form pointer is at the right-hand end of the form, the Z argument furnishes the value string. The form pointer is not moved.

If the form pointer is not at the right-hand end of the form, the value string of the primitive consists of a copy of the string of characters of the form extending from the location of the form pointer to the location of the next segment gap marker which is to the right of the form pointer, or to the right-hand end of the form if there is no such segment gap marker. The form pointer is reset to the location just to the right of the segment gap marker which terminates the copied string, or otherwise to the right-hand end of the form.

- - -

Note that two adjacent segment gap markers define a null string segment. A form consisting only of two adjacent segment gap markers, and

no text characters, produces the following results from successive applications of the call segment CS primitive. First application: null string. This is the null string preceeding the first segment gap marker. Second application: null string. This is the null string segment located between the two segment gap markers. The form pointer is now at the right-hand end of the form. Third application: Z argument.

The primitive 'call N', with mnemonic CN , gets its name from the fact that it is used to copy a specified number of characters from the text of a form. In the definition of this primitive, the number of characters to be copied is specified by a string of decimal digits found at the right-hand end of the D1 argument string. The D1 argument string may also contain alphabetic or other characters which prefix the decimal digits found at the right-hand end. Except for a single plus + or minus - sign immediately preceeding the string of decimal digits at the right-hand end, all other non-numerical prefix characters of the D1 argument are ignored. The single sign character, if there is any, together with the terminal string of decimal digits is called the 'signed numerical part' of the D1 argument string. If the D1 argument fails to have a signed numerical part, or if it is a null string, a default signed numerical part of +0 is taken. It should be noted that this signed numerical part is determined in exactly the same manner as the signed numerical part of the arguments of the arithmetical primitives which are defined later.

When the form pointer is at the left-hand end of the form, it is located just to the left of the left-most character or segment gap marker

of the form. When the form pointer is at the right-hand end of the form, it is located just to the right of the right-most character or segment gap marker of the form.

Call N CN,N1,D1,Z

Four arguments. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string.

If such a form cannot be found, the primitive has a null value string, the primitive does nothing, and the action of the primitive is complete.

If such a form is found, the signed numerical part of the D1 argument string is found and is interpreted as a decimal number. If D1 is a null string, or if no signed numerical part is found, the default value of +Ø is taken as the signed numerical part, and:

If the signed numerical part is a positive or unsigned number K (where K may be Ø), and:

If the form pointer is at the right-hand end of the form, the Z default argument supplies the value string of the primitive, the form pointer is not moved, and the action of the primitive is complete.

If the form pointer is not at the right-hand end of the form, the pointer is moved to the right until it stands just to the left of the first character encountered, or if there is no such character, it is moved to the right-hand end of the form, and:

If the signed numerical part is Ø or +Ø, the primitive has a null value string, the form pointer is not again reset, and the action of the primitive is complete.

If K is greater than Ø, and if there are at least K characters to the right of the form pointer, the value

string of the primitive is formed by copying a sequence of K characters from the form, the sequence beginning with the character just to the right of the form pointer and ending with the Kth character to the right of the form pointer, ignoring all segment gap markers. The form pointer is reset to the location just to the right of the Kth character copied, and the action of the primitive is complete.

If K is greater than Ø, and if there are not as many as K characters to the right of the form pointer, the value string is formed by copying the characters which are available, the form pointer is reset to the right-hand end of the form, and the action of the primitive is complete.

If the signed numerical part is a negative number -K (where K may be Ø), and:

If the form pointer is at the left-hand end of the form, the Z default argument supplies the value string of the primitive, the form pointer is not moved, and the action of the primitive is complete.

If the form pointer is not at the left-hand end of the form, the pointer is moved to the left until it stands just to the right of the first character encountered, or if there is no such character, it is moved to the left-hand end of the form, and:

If the signed numerical part is -Ø, the primitive has a null value string, the form pointer is not again reset, and the action of the primitive is complete.

If K is greater than Ø, and if there are at least K characters to the left of the form pointer, the value string of the primitive is formed by copying a sequence of K characters from the form, the sequence beginning with the Kth character to the left of the form pointer and ending with the character just to the left of the form pointer,

ignoring all segment gap markers. The form pointer is reset to the location just to the left of the first character copied, and the action of the primitive is complete.

If K is greater than Ø, and if there are not as many as K characters to the left of the form pointer, the value string is formed by copying the characters which are available, the form pointer is reset to the left-hand end of the form, and the action of the primitive is complete.

- - -

If the form consists of nothing but segment gap markers, and if the form pointer is at the left-hand end of the form, then the CN primitive with a D1 argument of Ø or +Ø will produce a null value string, and the form pointer will move to the right-hand end of the form. Another application of the same primitive will now produce a value string provided by the Z default argument. If the form pointer is at the right-hand end of the form, and the primitive with -Ø in the D1 argument is used, the first application produces a null value string, and the second application produces a value provided by the Z default argument.

The CN primitive can therefore be used to determine whether the form pointer is at the end of the form. To test whether the pointer is at the right-hand end of the form, the primitive with Ø in the D1 argument is used. If the pointer is at the right-hand end, the Z default argument provides the value string of the primitive; if not, the primitive has a null value string. The use of -Ø in the D1 argument will similarly test whether the pointer is at the left-hand end of the form.

Although the CN primitive behaves differently for +Ø or -Ø in the D1 argument, the arithmetic primitives ignore this distinction.

By the use of the 'initial' primitive with mnemonic IN , it is possible to copy a substring out of the text of a form, where the substring copied extends from the form pointer up to some specified sequence of characters.

Initial IN,N1,T1,Z

Four arguments. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form cannot be found, this primitive has a null value string, and the primitive does nothing.

If such a form is found, the text of the form is examined from left to right for the occurrence of the first substring in the text of the form which matches the T1 argument string. If the T1 string is a null string, it does not match anything. A substring of the form which contains a segment gap marker does not match anything.

If no match is found, or if the form pointer is at the right-hand end of the form, the Z default argument supplies the value string of the primitive, and the form pointer is not moved.

If a match is found, the value string of the primitive is formed by copying the characters of the form beginning with the first character to the right of the form pointer, and ending with the last character preceeding the matching substring. Segment gap markers are ignored in making the copy. The form pointer is reset to the location just to the right of the right-most character of the matching substring of the form.

- - -

Note that if the form has a segment gap marker at the right-hand end, and the form pointer is not at the right-hand end, the IN primitive is unable to move the form pointer to the right-hand end of the form.

The IN primitive can be used to determine whether a form of a specified name is in forms storage. This is done by putting the specified name string in the N1 argument, the null string in the T1 argument, and some non-null string in the Z default argument. Then, if the form is in forms storage, the value string of the primitive is provided by the Z argument string. If the form is not there, the IN primitive has a null value string.

## 5.9 TREATMENT OF NUMERICAL STRINGS

The primitives concerned with numerical quantities provide a special manner of treatment for 'numerical argument' strings D1, D2, etc., which represent numerical quantities. A special treatment is necessary because TRAC language permits any sequence of characters to be used at any place or in any argument.

In a primitive taking numerical arguments, the scanning rules are followed up to and including the formation of the execution diagram for the primitive. At this stage, the arguments which represent numerical quantities are given a special treatment to separate out the string of numerical digits which represents the numerical quantity of the argument.

The processor examines the numerical argument string, character by character, from right to left, beginning at the right-hand end of the string, searching for the first instance of a non-numerical character. A non-numerical character is a character which is not in the range of digits appropriate for the numerical representation being used. The arithmetic primitives are defined for decimal digits, so therefore the non-numerical

characters are those which are not in the range ø to 9. When the first instance of a non-numerical character is found, the part of the argument string to the right of this first non-numerical character is taken as the 'unsigned numerical part' of the argument. If the actual unsigned numerical part is a null string, or if the numerical argument string is missing or is a null string, then a default unsigned numerical part consisting of the digit ø is used. If the first non-numerical character is a + or - sign, then this sign is taken as a prefix character to the unsigned numerical part to form a string which is called the 'signed numerical part' of the argument string. If the first non-numerical character is not a + or - sign, then nothing is added to the unsigned numerical part to form the signed numerical part of the argument string.

The arithmetic operations in TRAC language are performed on the basis of the signed numerical parts of the numerical arguments.

In an argument string in which the signed numerical argument has been located, there may be additional leading characters to the left of the signed numerical part. The substring of the argument formed by these leading characters is called the 'prefix part' of the original argument string. The prefix part may consist of any characters whatsoever, such as additional plus and minus signs, alphabetic characters, other digits in the range ø to 9, punctuation marks, or any other characters.

It should be noted that, according to this technique for the determination of the signed numerical part of the argument, the expression -153.ø2 will have ø2 as its signed numerical part, and -153. as its

'numerical arguments'
'prefix part'
'signed numerical part'
'numerical result'
'concatenating'

prefix part. Therefore when decimal or monetary expressions are to be handled, adequate provision must be made for dealing with the situation.

In the numerical value strings which are produced by the arithmetic primitives, the numerical part of the answer string is given a sign only when the numerical result is a negative number, in which case it is given the - prefix sign. Leading or extra zero digits to the left end of the numerical string are deleted in the value string of the primitive. If the numerical result is zero, the numerical value string is given as $\emptyset$ .

Although the call N primitive makes use of a representation with signed values of zero (i.e., $+\emptyset$ and $-\emptyset$ are distinguished), the arithmetic primitives treat $+\emptyset$ and $-\emptyset$ as being numerically equal.

## 5.10 THE FOUR ARITHMETIC PRIMITIVES (AD, SU, ML, DV)

The arithmetic operations in TRAC language are performed according to whole-number or integer arithmetic. Each of the arithmetic primitives has two 'numerical arguments' which contain the decimal representation of the numerical quantities on which the arithmetic operation is to be performed. Either of the numerical arguments may have a non-numerical 'prefix part' in addition to a 'signed numerical part'. The numerical computation of the arithmetic primitives is performed according to the signed numerical parts of the two numerical arguments. The result of the computation is a 'numerical result' string.

The complete value string of each of the arithmetic primitives is formed by 'concatenating' or putting together the prefix part of the first numerical argument of the primitive with the numerical result string

obtained from the numerical computation of the primitive. The prefix part of the second numerical argument of the primitive is ignored. For example, if the two numerical argument strings APPLES 3 and ORANGES 6 are added by the AD primitive, the total value string is APPLES 9 .

In case any of the arithmetic operations overflows the capacity of the TRAC processor, due to the excessive magnitude of a numerical quantity, or due to an excessive number of characters in its representation, the arithmetic primitives are provided with a Z default argument. Then if an overflow occurs, the value string of the primitive is the Z default argument string. In case division by zero is attempted, the value string of the primitive is also provided by the Z default argument.

Add AD,D1,D2,Z

Four arguments. Execution of this primitive adds the number represented by the signed numerical part of the D1 argument to the number represented by the signed numerical part of the D2 argument to produce a numerical result string. The value string of the primitive is normally obtained by concatenating the prefix part of the D1 argument with the numerical result string. In case of overflow, the value string is supplied by the Z default argument.

- - -

Subtract SB,D1,D2,Z

Four arguments. Execution of this primitive subtracts the number represented by the signed numerical part of the D2 argument from the number represented by the signed numerical part of the D1 argument to produce a numerical result string. The value string of the primitive is normally obtained by concatenating the

'multiply'
'divide'

prefix part of the D1 argument with the numerical result string. In case of overflow, the value string is supplied by the Z default argument.

- - -

Multiply ML,D1,D2,Z

Four arguments. Execution of this primitive multiplies the number represented by the signed numerical part of the D1 argument by the number represented by the signed numerical part of the D2 argument to produce a numerical result string. The value string of the primitive is normally obtained by concatenating the prefix part of the D1 argument with the numerical result string. In case of overflow, the value string is supplied by the Z default argument.

- - -

Divide DV,D1,D2,Z

Four arguments. Execution of this primitive divides the number represented by the signed numerical part of the D1 argument by the number represented by the signed numerical part of the D2 argument to provide a quotient. If the quotient is not an integer, the quotient is rounded downward by decreasing its absolute value to the next smaller integer, and the resulting rounded signed integer value is the numerical result string.

The value string of the primitive is normally obtained by concatenating the prefix part of the D1 argument with the numerical result string. In case of overflow, or in case that the signed numerical part of the D2 argument is zero, the value string is supplied by the Z default argument.

- - -

In all the four arithmetic primitives, the usual algebraic sign conventions are adhered to. Accordingly, the primitives #(DV,3,2) and

'string arithmetic'
'Boolean vectors'
'Boolean arguments'
'octal digits'
'octal part'



#(DV,-3,-2) have the value string 1 , while #(DV,-3,2) and #(DV,3,-2) have the value string -1 . Furthermore, #(DV,-2,3) has the value string ø , While #(DV,3,ø,Z) has the value string Z .

In some TRAC processors, 'string arithmetic' is performed, In these processors, numbers are represented by strings of digits, where the strings may be arbitrarily long. An overflow situation is extremely unlikely with the three primitives AD , SU and ML in such processors, since overflow can occur only when the total string capacity of the processor is exceeded.

## 5.11 THE BOOLEAN PRIMITIVES (BU, BI, BC, BR, BS)

TRAC language provides for "logical" bit manipulations of 'Boolean vectors', where the Boolean vectors are sequences of binary zeroes and ones. There are five Boolean primitives, and each of them has either one or two arguments which are 'Boolean arguments'. In the Boolean arguments, the Boolean vectors are represented by strings of 'octal digits', i.e., by digits in the range of ø to 7 . Each octal digit represents a group of three binary zeroes or ones. For example, the octal digit 5 represents the the binary group 1ø1 , and the octal string ø47 represents the Boolean vector øøø1øø111 .

Since the Boolean arguments, like any of the arguments of TRAC primitives, may contain any characters whatsoever, the Boolean primitives operate only upon the 'octal part' of these Boolean arguments. The octal part of a Boolean argument corresponds to the unsigned numerical part of a numerical argument of an arithmetic primitive. The octal part is found in much the same way as an unsigned numerical part.

After the execution diagram has been formed, the processor examines a Boolean argument string character by character, from right to left, beginning at the right-hand end of the string. It searches for the first instance of a non-octal character, i,e., a character which is not a digit in the ø to 7 range. When the first instance of a non-octal character is found, the part of the argument string to the right of the first non-octal character is taken as the octal part of the argument.

The octal part of an argument may be a null string. A missing Boolean argument, or a null-string Boolean argument, provides a default octal part which is the null string.

The logical bit manipulations of the Boolean primitives are performed on the Boolean vectors represented by the octal parts of the Boolean arguments. The result of such manipulations on the Boolean vectors is again a Boolean vector. The final argument string of the primitive is the octal string representation of the resulting Boolean vector. Prefix parts of the Boolean arguments are ignored.

The 'Boolean union' primitive, with mnemonic BU , depends upon use of the 'inclusive logical OR' operation on corresponding binary digits of a pair of Boolean vectors. If two corresponding binary digits are both ø , then the resulting inclusive logical OR binary digit is ø . In all other cases, the resulting inclusive logical OR binary digit is 1 .

Boolean Union BU,B1,B2

Three arguments. Execution of this primitive determines the octal parts of the two Boolean arguments B1 and B2 . If the octal part of either argument has fewer octal digits in its

string representation than the other, octal zero digits are added to the left end of the shorter octal part to make the octal parts have octal strings of the same number of octal digits. The two octal parts are converted to equivalent Boolean vectors, and the inclusive logical OR operation is performed on corresponding binary digits of the two vectors to produce a Boolean vector result. The value string of the primitive is the octal string representation of the Boolean vector result.

- - -

The 'Boolean intersection' primitive, with mnemonic BI , depends upon the use of the 'logical AND' operation on corresponding binary digits of a pair of Boolean vectors. If two corresponding binary digits are both 1 , then the resulting logical AND binary digit is 1 . In all other cases, the resulting logical AND binary digit is $\emptyset$ .

Boolean Intersection BI,B1,B2

Three arguments. Execution of this primitive determines the octal parts of the two Boolean arguments B1 and B2 . If the octal part of either argument has more octal digits in its string representation than the other, excess octal digits are deleted from the left end of the longer octal part to make the octal parts have octal strings of the same number of octal digits. The two octal parts are converted to their equivalent Boolean vectors, and the logical AND operation is performed on the corresponding binary digits of the two vectors to produce a Boolean vector result. The value string of the primitive is the octal string representation of the Boolean vector result.

- - -

'Boolean complement'
'logical complement'
'Boolean rotate'
'Boolean shift'

The 'Boolean complement' primitive, with mnemonic BC , depends upon the use of the 'logical complement' operation on the binary digits of the Boolean vector of the one Boolean argument of this primitive. The logical complement of a Boolean vector is formed by replacing each binary ø digit by a binary 1 digit, and each 1 digit by a ø digit, throughout the Boolean vector.

Boolean Complement BC,B1

Two arguments. Execution of this primitive determines the octal part of the Boolean argument B1 . The octal part is converted to its equivalent Boolean vector, and the logical complement of this vector is formed to provide the Boolean vector result. The value string of the primitive is the octal string representation of the Boolean vector result.

- - -

In the case that the B1 argument is a null string, or is missing, the value string of the Boolean complement primitive is a null string.

The 'Boolean rotate' and the 'Boolean shift' primitives, with the mnemonics BR and BS , make it possible to move the binary digits of a Boolean vector to the left or to the right by a specified number of binary places. The direction of the move, and the number of places moved, is specified by a signed decimal number provided by the numerical argument D1 of the primitive. The total number of binary digit places in the Boolean vector does not change when the binary digits are moved. With the Boolean rotate primitive, the digits which are moved off one end of the vector are replaced in the vacated places at the other end of the vector. With the

Boolean shift primitive, the digits moved off the end of the vector are lost, and the vacated places at the other end of the vector are filled with binary ø digits.

Boolean Rotate BR,D1,B1

Three arguments. Execution of this primitive determines the octal part of the Boolean argument B1 and the signed numerical part of the numerical argument D1 . The octal part is converted to its equivalent Boolean vector. The binary digits of the vector are moved to the left or to the right as a group, with a step of one bit at a time, in the fixed-length vector, with the direction of the movement being to the left if the signed numerical part is positive, and to the right if it is negative. Digits moved off one end of the fixed-length vector are replaced in the vacated place at the other end of the vector. The number of steps of movement is determined by the absolute value of the signed numerical part. When the movement of the binary digits is completed, the resulting Boolean vector is converted to its octal representation to provide the value string of the primitive.

- - -

The 'absolute value' of a number is the number without its sign designation. Thus the absolute value of +9 and of -9 is simply 9 .

Boolean Shift BS,D1,B1

Three arguments. Execution of this primitive is carried out exactly the same as for the Boolean rotate primitive, except that the binary digits which are moved off one end of the fixed-length vector are ignored, and the vacated palces at the other end of the vector are filled by binary ø digits.

- - -

If the numerical argument D1 of either the BR or BS vectors is a null string, no movement takes place, and the value string of the primitive is the same as the octal part of the B1 argument. If the Boolean argument B1 is a null string, the value string of the primitive is a null string. If the absolute value of the signed numerical part of the D1 argument is greater than the number of binary digits in the Boolean vector represented by the octal part of the B1 argument, the value string of the BS primitive will either be composed exclusively of octal ø digits, or will be the null string if the B1 argument is a null string.

## 5.12 THE DECISION PRIMITIVES (EQ, GR)

TRAC language provides two decision primitives which permit branching in TRAC language scripts. The first of these compares two strings to determine whether the characters in two strings are identical or equal. This primitive is called 'string equality', with the mnemonic EQ . The second primitive, called 'greater than' with the mnemonic GR , makes an arithmetic comparison of two numerical values to determine whether one is algebraically greater than the other. Depending upon the way the comparison goes, with both the EQ and GR primitives, the value string of the primitive is one or the other of the two remaining arguments of the primitive.

String Equality EQ,T1,T2,T3,T4

Five arguments. Execution of this primitive compares the character string of the T1 argument with the character string of the T2 argument. If the two character strings are identical, the value string of the primitive is the T3 argument

string, and if they are not identical, the value string is the T4 argument string.

- - -

Greater Than GR,D1,D2,T1,T2

Five arguments. Execution of this primitive algebraically compares the signed numerical part of the D1 argument with the signed numerical part of the D2 argument. If the signed numerical part of the D1 argument represents a number algebraically greater than the signed numerical part of the D2 argument, then the value string of the primitive is the T1 argument string, otherwise the value string is the T2 argument string.

- - -

In the GR primitive, prefix strings in the D1 or D2 arguments are ignored. Note that #(GR,-1,-1Ø,T1,T2) takes its value string from the argument T1 , since the comparisons are made in the algebraic sense. The two expressions +Ø and -Ø are equal, so a comparison on these would provide the value string from the T2 argument.

## 5.13 THE AUXILIARY STORAGE PRIMITIVES (SB, FB, EB)

With TRAC language, there are two methods for the storage of strings. By the first method, the define string DS primitive is used to set up forms in the forms storage area of the processor. Such forms are directly accessible for manipulation by means of the various kinds of calls. However, the amount of space available for the storage of forms in the forms storage area of the processor is often limited. Therefore there is a need for the second method of string storage, by means of which various

'auxiliary storage'
'store block'
'block'
'fetch block'
'erase block'
'block name'
'hardware address'
'block form'

seldom-used forms can be moved to a more capacious 'auxiliary storage' area which may be outside the processor. Such auxiliary storage may be on disks, drums or tapes.

The transfer of forms between forms storage and auxiliary storage is controlled by three auxiliary storage primitives. The 'store block' primitive with mnemonic SB takes a group of forms from the forms storage area and puts them into a 'block' in auxiliary storage. The 'fetch block' primitive with mnemonic FB takes a copy of a group of forms in a block in auxiliary storage and restores the forms in the forms storage area of the processor. When a group of forms is no longer needed, the 'erase block' primitive with mnemonic EB erases the block and its forms from auxiliary storage.

All the segment gap markers and form pointers of the forms remain intact during the storing and fetching.

A block may contain one or more forms, or it may be empty -- containing no forms. When a block is created, it is given a 'block name' which is usually quite different from the names of the forms in the block.

The location of the block in the auxiliary storage device is specified by a 'hardware address'. The hardware address consists of a string of digits or other characters. The processor uses the hardware address in retrieving or storing a block. When a new block is created, the processor automatically creates a new hardware address.

When the processor creates a new block in auxiliary storage, it also automatically creates a 'block form' which is a form whose name string

consists of the name string of the block, and whose text consists of the digits or characters of the hardware address. Except for the use to which it is put, the block form is no different from any other TRAC form.

For retrieval of a named block by the fetch block primitive, or for erasing a named block, the hardware address stored in the block form is used by the processor to locate the actual block. If the block form is missing, the processor has no way of determining the hardware address of a block, and then the fetch and erase operations cannot be performed on a block, even if the block name is known.

If the store block operation is performed with a block name which is the same as the name of an existing block, the previous contents of the block are replaced by the new group of forms. The hardware address is not changed.

If, for some reason, the operation of a storage primitive cannot be carried out in the ordinary fashion, because of some kind of logical, mechanical or electrical error, the processor types out a diagnostic of <STE> meaning 'storage transfer error'. When this happens, the various blocks in auxiliary storage and forms in the processor are preserved intact, and the primitive produces no change or other action.

Store Block SB,N1,N2, ...

Three or more arguments. Null value string. Execution of this primitive first determines whether the N1 argument is present. If the argument is missing, the primitive does nothing, and the action of the primitive is complete.

If the N1 argument is present (even if a null string),

the processor searches for a form in forms storage having a name string the same as the N1 argument string. If such a form is found, it is treated as a block form and the text of the form is taken as the hardware address of the block to be stored. If such a form is not found, the processor generates a new hardware address. The block with this hardware address is located in auxiliary storage.

The processor then determines whether the N2 argument is present. If it is, the processor searches for and locates forms in forms storage with name strings the same as the N2, N3, etc., argument strings. If a form for a given argument cannot be located, the argument is ignored and the processor goes on to the next argument. If the N2 argument is not present, the argument is not replaced by a default null string, the N2 argument is ignored, and the search is completed. The group of forms resulting from the search (which may be a null group) is then copied as a group into auxiliary storage at the block location specified by the hardware address.

In the case that a new hardware address was generated, and if the transfer of the group to auxiliary storage was successful, a new block form is created whose name string is provided by the N1 argument string and whose text is the new hardware address.

If the transfer of the group to auxiliary storage was not successful because of equipment error or invalidity of the hardware address used, or if it was not possible to set up the new block form with the name provided by the N1 argument string, then the diagnostic action is taken. In the diagnostic action, the diagnostic <STE> is typed out at the reactive typewriter, the previous content in auxiliary storage at the hardware address is left intact, any new content copied into auxiliary storage is erased, and the action of the primitive is complete.

If the diagnostic action is not taken, the previous content in auxiliary storage at the hardware address is erased, the new content just copied in is retained with the hardware address, and in forms storage the forms of the group are erased, with the exception of any form with name provided by the N1 argument string, and the action of the primitive is complete.

- - -

The N1 argument specifies the 'block name' for the block of forms, and also provides the name string for the 'block form' which contains the hardware address of the block in auxiliary storage.

If the N1 argument is missing, as in the expression #(SB) , the primitive does nothing. If the N2 argument is missing, as in #(SB,N1) , a valid block is created, but the block has no forms in its content. A missing N2 argument is not given a default null string value. A form with a null string name can be stored with the expression #(SB,N1,) .

Sometimes it is useful to create an empty block and its corresponding block form. This can be done with an expression #(SB,N1) . Then, at a later time, forms can be inserted into the block.

By simple technological means in the processor, the single hardware address can be made momentarily to address both the old and the new content in auxiliary storage while the action of the primitive is being completed. The action of the SB primitive has been carefully defined so that it prevents the possibility of a block of forms being made inaccessible as a consequence of the block form being sent to auxiliary storage. Also, if the complete transfer and action of the primitive cannot be successfully accomplished, nothing happens to the forms in forms storage or the block in auxiliary storage.

In general, the diagnostic action <STE> is taken whenever the orderly execution of the store block primitive, or of the other two auxiliary storage primitives, FB and EB , cannot be completed. Some of the possible causes for the diagnostic action are as follows: There may be equipment error, such as an electrical malfunction of the auxiliary storage device, the fact that the device is not connected or has no power, some electrical error in the data transmission, a mechanical malfunction, or the like. There may be an error due to an invalid hardware address, which may be caused by a text string which does not have the correct form for a hardware address in the particular processor and system, or an address of correct form which does not correctly address a valid block for some reason. Another cause for the diagnostic action is a space overflow either in the processor itself -- in the workspace or forms storage -- or an overflow in the auxiliary storage device.

Each of the three auxiliary storage primitives is defined so that occurrence of the diagnostic action does not destroy forms stored either in the forms storage area or in auxiliary storage.

Fetch Block FB,N1

Two arguments. Null value string. If the N1 argument is missing, a default null string is supplied. Execution of this primitive causes a search for a form in forms storage having a name string the same as the N1 argument string. If such a form is not found, the primitive does nothing, and the action of the primitive is complete.

If such a form is found, it is treated as a block form and the text of the form is taken as the hardware address which is

then used to locate a block in auxiliary storage. The group of forms of the block are copied temporarily into the processor.

If the transfer of the group from auxiliary storage is not successfully completed because the block cannot be located, because of equipment error, because of invalidity of the hardware address, or because of lack of space to hold the forms in forms storage, the diagnostic action is taken. In the diagnostic action, the diagnostic <STE> is typed out at the reactive typewriter, the temporary copy in the processor is erased, and no changes are made in the forms in forms storage or in the blocks in auxiliary storage, and the action of the primitive is complete.

If the diagnostic action is not taken, the group of forms in the temporary copy are copied into forms storage, with any form of the group replacing any form already in forms storage with the same name. The temporary copy is erased. No changes are made in the blocks in auxiliary storage or in the block form. The action of the primitive is complete.

- - -

Erase Block EB,N1

Two arguments. Null value string. If the N1 argument is missing, a default null string is supplied. Execution of this primitive causes the processor to search for a form in forms storage with a name string which is the same as the N1 argument string. If such a form is not found, the primitive does nothing, and the action of the primitive is complete.

If the form is found, it is treated as a block form and the text of the form is taken as a hardware address which is then used to locate a block in auxiliary storage.

If the block cannot be located, or if the hardware address is invalid, or if there is an equipment error, the diagnostic action is taken. In the diagnostic action, the diagnostic <STE>

is typed out at the reactive typewriter, nothing is changed in forms storage or in auxiliary storage, and the action of the primitive is complete.

If the diagnostic action is not taken, the block at the hardware address is erased and the block form with name given by the N1 argument is deleted from forms storage. The action of the primitive is complete.

- - -

Absence of the block form from forms storage, or its deletion by the DD primitive, does not affect the corresponding block held in auxiliary storage. Therefore, if the hardware address string is copied out in some fashion and is separately preserved, the block form may be deleted from forms storage. At a later time, when the block form is needed for operation of one of the auxiliary storage primitives, the block form may be recreated, by means of the define string DS primitive.

A group of block forms may themselves be put into a 'super block', and the super block may be stored by means of the store block primitive. When this is done, in order to use a form in any block of the group, a fetch block action must be done with the super block name, and then another fetch block action must be done with the name of the block containing the desired form.

## 5.14 THE DIAGNOSTIC PRIMITIVES (LN, PF, TN, TF)

The four diagnostic primitives display what is happening inside the processor and they provide assistance in discovering errors during the development of scripts.

The 'list names' primitive with mnemonic LN provides a list of names of all the forms in forms storage.

List Names LN,T1

Two arguments. Execution of this primitive produces a value string which lists in sequence a copy of the name string of each of the forms in forms storage, with each of the name strings preceded by a copy of the string from the T1 argument. The first or left-most name in the value string is the name of the most-recently-created form, with the rest of the form names given in order.

- - -

Operation of the segment string primitive SS does not change the sequence of names provided by LN . However, the action of the store block SB and the fetch block FB primitives does change the sequence of names, since the original forms are deleted by the SB primitive, and are again recreated in forms storage by the FB primitive. Redefinition of a form by DS will ordinarily change the sequence.

As with all value strings produced by primitives, the value string of the LN primitive is copied into the workspace and is scanned according to the scanning rules. However, the other diagnostic primitives are null valued.

The 'print form' primitive PF displays the text, segment gap markers, and form pointer of any named form. The representation of the form is typed out at the reactive typewriter. The display string does not enter the workspace.

'print form'
'trace mode'
'trace on'
'trace off'

Print Form PF,N1

Two arguments. Null value string. Execution of this primitive causes the processor to search for a form in forms storage with a name string which is the same as the N1 argument string. If such a form is not found, no further action is taken.

If the form is found, the text of the form is printed out at the reactive typewriter with an indication for the location of each segment gap marker and for the location of the form pointer. Each segment gap marker is indicated by the appropriate decimal numeral for the ordinal number of the marker, with the numeral being enclosed by angle brackets, e.g., <1> . The location of the form pointer is indicated by the "up arrow" character, also enclosed by angle brackets, <↑>.

- - -

With reactive typewriters which do not have an "up arrow" character, some other character may be substituted.

A TRAC processor has a 'trace mode' in which the step-by-step action of the processor is displayed at the reactive typewriter. The trace mode action is turned on by the 'trace on' primitive with mnemonic TN , and the action is turned off by the 'trace off' primitive with mnemonic TF.

Trace On TN

One argument. Null value string. Execution of this primitive causes the processor to go into the trace mode.

In the trace mode, the processor continues to operate and to execute TRAC language scripts as usual, with the exception that whenever the execution diagram of a primitive is fully developed, the action stops and a representation of the execution diagram is printed out at the reactive typewriter.

A single-line representation is used. An upstairs syntactic group #( or ##( is shown in the single-line representation by #/ or ##/ . Each syntactic comma argument separator is shown in the single-line representation by * . The right-hand parenthesis of the primitive is shown by / . If the sequence of characters in the representation would overflow the line length of the reactive typewriter, carriage return and line feed characters are automatically inserted where necessary at the end of each display line.

After the representation of the execution diagram for a primitive has been printed out, the action pauses and the processor waits for a continutation signal to be given from the reactive typewriter keyboard. When the carriage return character continuation signal is given (or the enter character for some computers), the processor action resumes and the primitive whose execution diagram was just displayed is now executed.

The scanning action of the processor continues until the execution diagram of the next primitive is fully developed. The processor then pauses and prints out the representation of this new execution diagram and waits for a continuation signal. And so on.

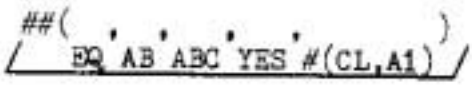
The trace action may be stopped from the keyboard at any pause by pressing any other key followed by the carriage return (or enter character) continuation signal key. The processor then goes out of the trace mode, the workspace is cleared and the the processor resumes its normal operation.

- - -

This kind of 'abortive stop' is generally used when it becomes clear that the script being observed is grossly defective, and it is desired to abruptly stop the running of the script.

An example of the single-line trace mode representation of an execution diagram is shown below:

Execution Diagram:




Trace Mode Representation:

##/EQ*AB*ABC*YES*#(CL,A1)/

Trace Off TF

One argument. Null value string. Execution of this primitive causes the processor to go out of the trace mode and normal operation resumes. The workspace is not cleared.

- - -

Operation of the processor upon selected portions of a long script can be displayed with the trace mode by inserting the TN primitive just preceeding the portion to be displayed, and the TF primitive just following it.

In order to use the two input primitives RS and RC in the trace node, extra care must be used. With the RS primitive, when the execution diagram is displayed, the continuation signal character is first typed at the keyboard. This starts execution of the primitive, and characters can be read in from the keyboard. The desired input string is now typed, followed by the meta character (and by the enter character if needed). The

RS primitive will accept the string, placing it in the workspace, and action of the processor in the trace mode will resume.

For input with the RC primitive, the continuation signal is given after the execution diagram has been displayed. Then a single input character is typed at the keyboard. (If an enter character is required, it is then typed. A meta character is not needed.) The processor will accept the input character, placing it in the workspace, and the action of the processor will resume.

## 5.15 THE TWO HOUSEKEEPING PRIMITIVES (HL, MO)

The two housekeeping primitives provide for supervisory control of the TRAC processor. The primitive 'halt' with mnemonic HL provides a means of stopping the action of the processor. When the processor stops with HL , the state of the workspace and of forms storage is maintained intact. Control of the situation then passes to the computer or to the computer's operating system. By an appropriate action taken later through the computer or keyboard, the action depending upon the particular equipment in use, the processor can be made to resume its scanning and executing actions exactly where it had left off.

With some computers and operating systems, it is possible to store the inactive processor, and then later to retrieve it, revive it, and cause it to resume its action.

In general, the HL primitive should be used to terminate every TRAC language session, especially when the user is working with a time-sharing system.

Halt HL

One argument. Null value string. Execution of this primitive causes the action of the TRAC processor to halt. The state of the workspace, the scanning action, and the forms storage area is held intact. Control passes to the computer or to its operating system. By suitable action taken through the computer or the keyboard, the processor can be made to resume its activity where it left off.

- - -

The 'mode' primitive with mnemonic MO has a variety of functions. Only one is standardized in this document. This one permits rigorous standardization of a TRAC processor, even though the processor may be provided with nonstandard features or additions. With the mode primitive, it is possible either to enable or to disable nonstandard features which may be attached to a standard TRAC processor.

Subsequent TRAC Standards will define other standard uses for the mode primitive.

Mode MO,T1

One or more arguments. Null value string. Execution of this primitive with the single argument MO causes a standard processor to assume exactly the standard TRAC T-64 capability and to type out the verification diagnostic <T64> at the reactive typewriter. If the standard T-64 capability is not available in the processor, the verification will consist of another suitable descriptive diagnostic.

If the processor contains nonstandard features or extensions, the extended version of the processor is made available by the execution of the MO primitive with the two arguments MO,E . A suitable descriptive diagnostic is typed out.

- - -

# 6. LIMITATIONS OF THIS STANDARD

## 6.1 DESIRE FOR COMPLETE SPECIFICATION

It is the desire and intent of Rockford Research, Inc., that this "Definition and Standard for TRAC T-64 Language" should be a complete definition, down to the last detail, of TRAC T-64 language, and of the corresponding functional behavior of the TRAC processor. Unfortunately, for technical reasons beyond our control, standardization to this degree of completeness and exactness is not possible at this time.

Limitations and variations among the different computers, their operating systems, and the connected reactive typewriters or keyboard-display devices, make such complete standardization impossible. It is useful in this Standard to spell out explicitly the things that it has not been possible to standardize.

The matters which escape complete standardization fall into three categories, each due to variations of a particular kind in the external equipment and systems. The three classes of variations are:

1) Variations in computer memory sizes.

2) Variations in equipment and systems involving input and output treatment of particular characters, the enter character, the editing of other characters, and the panic stop.

3) Variations due to features connected with auxiliary storage.

These matters are described in the three following sections.

For these three nonstandard areas, the user of a TRAC processor must seek auxiliary information in order to determine exactly the peculiarities of his particular computer and equipment.

## 6.2 VARIATIONS IN COMPUTER MEMORY SIZE

Going from the smallest of the minicomputers to the larger time-shared machines, a typical TRAC language processor will have a total string capacity ranging from 1,500 to 20,000 or more characters. The same TRAC language is usable with all such computers, irrespective of capacity. Yet some of the longer TRAC language scripts, which run with ease on the larger processors, will run only with difficulty, or not at all (without modification) on the smaller processors. This is unfortunate. However, it would be even less desirable in a TRAC language standard to restrict all users to the same small-sized processor.

Therefore, this Definition and Standard does not specify the capacity of a standard TRAC processor in terms of the maximum number of characters it can hold. Instead of giving such a specification as part of the language definition, several alternative techniques are provided.

By the first technique, each TRAC language user should know the capacity of his TRAC processor in terms of its maximum capacity of characters. Then, when such a worker exchanges TRAC language scripts with other workers who have processors of other sizes, he will be able to characterize the behavior of his scripts with a statement like, "These scripts work well in a TRAC processor providing 2,500 characters of workspace and forms storage." A recipient of such a script, with a processor of the same or larger size, should then expect no trouble in running these scripts. On the other hand, a recipient with a smaller processor whould expect to make some adjustments of the scripts in order to get them to run properly.

<SCA> 'string capacity alert'
<SCE> 'string capacity exceeded' - 78 -

The second technique is based on the fact that certain processors, such as those for the larger computers, have a variable size. They permit the user to specify the capacity of the TRAC processor at the time that the processor is put into operation at the beginning of a session. A user with such a processor is able to tailor the capacity of his processor to the requirements of the scripts he will be working with. For reasons of minimal charge for computer use, and also for ease of exchange of scripts, a user with this kind of expansive facility is advised to develop his scripts while using one of the optional smaller sizes of the processor.

A final technique for dealing with the variations in computer memory size, and the related variable size of the TRAC processors, is provided by the two overflow diagnostics which are standard to the language. During operation, whenever the processor finds that fewer than 1ØØ additional characters would overflow the processor, the processor emits the 'string capacity alert' diagnostic <SCA> at the reactive typewriter. Usually a corrective action can then be taken. In case an actual overflow situation does occur, the 'string capacity exceeded' diagnostic <SCE> is provided to indicate that the processor has overflowed and that the workspace has been cleared.

Some of the TRAC processors, especially those for computers with limited memory size, may not provide arithmetic computations based on numerical representations by strings of an indefinitely large number of digits. For such processors, the overflow diagnostic of the individual primitive will signal an overflow situation for numbers beyond a certain limited size. However, where a string numerical arithmetic facility is

contained in the processor, overflow will generally be very unusual.

## 6.3 PROBLEMS WITH CHARACTERS

Once a string of characters has been received in the workspace of the processor, the operation of TRAC language is exactly specified by this Standard. However, because of peculiarities of computers, operating systems, and typewriters or displays, a number of serious problems exist in moving character strings from the keyboard to the processor, and back out again.

One of the worst, and most annoying, problems is the requirement by many computers for a special 'enter character' which must be used to cause a typed string to be passed through the operating system to the processor. As already mentioned in this Standard (in connection with the read string primitive and the read character primitive) the redundant enter character is a serious nuisance. At present, there is no consistent way for dealing with the enter character which can be given in this Standard.

Some typewriters or keyboard-display devices fail to have certain characters (such as the reverse slant) which are described in this Standard. Thus substitute characters may be necessary for the print form representation, or the trace mode representation, or for the error correction characters.

Because of limitations or peculiarities of some keyboard units, or of some computer operating systems, the actual performance of the error correction actions may be different with different devices.

Some computer operating systems unfortunately "edit out" or ignore

certain characters as they come from the keyboard, or they may perform peculiar or unexpected actions on certain characters. This is particularly apt to be true of the nonprinting 'control characters' in the ASCII code. The user of TRAC language must carefully determine which characters are affected, and how, for the particular equipment he is using.

Finally, the method for making the 'panic stop' varies greatly from one kind of equipment to another. Since the panic stop is a very important control action for emergency use, the user of TRAC language must determine how to perform the panic stop action with his particular equipment.

## 6.4 PROBLEMS WITH AUXILIARY STORAGE

In the same way that standard TRAC language runs into difficulties in communicating with the reactive typewriter, it also runs into peculiarities in the manner of use of auxiliary storage.

It is not possible to standardize the nature of the hardware address string, which is used by the auxiliary storage primitives.

The situations which give rise to the error diagnostic <STE> , 'storage transfer error', cannot be completely standardized.

Some computer systems permit blocks in auxiliary storage to be retained in storage between working sessions. The manner for doing this cannot be specified in this Standard, and must be determined for the particular equipment in use.

* * *

## POSTSCRIPT

We at Rockford Research, Inc., will appreciate receiving notice of any inconsistencies, ambiguities, or points lacking in clarity in this Definition and Standard. It may be possible to remedy the point of difficulty in the next revision. Because of the length of time that TRAC T-64 language has been in existence and in experimental use, only minor adjustments of the definitions are now expected to be required.

* * *

## THE 34 TRAC LANGUAGE PRIMITIVES AND THEIR ARGUMENT TYPES

INPUT AND OUTPUT CONTROL (Sec. 5.2)

Print string ----- PS,T1

Read string ------ RS

Change meta ------ CM,T1

Read character --- RC

STRING STORAGE AND DELETION (Sec. 5.4)

Define string ---- DS,N1,T1

Delete definition DD,N1,N2, ...

Delete all ------- DA

CREATION OF TEXT MACROS (Sec. 5.5)

Segment string --- SS,N1,T1,T2, ...

CALLS AND PARTIAL CALLS (Sec. 5.6, 5.8)

Call ------------- CL,N1,T1,T2, ...
  Default call --- N1,T1,T2, ...

Call restore ----- CR,N1

Call character --- CC,N1,Z

Call segment ----- CS,N1,Z

Call N ----------- CN,N1,D1,Z

Initial ---------- IN,N1,T1,Z

ARITHMETIC OPERATIONS (Sec. 5.10)

Add -------------- AD,D1,D2,Z

Subtract --------- SU,D1,D2,Z

Multiply --------- ML,D1,D2,Z

Divide ----------- DV,D1,D2,Z

BOOLEAN OPERATIONS (Sec. 5.11)

Boolean union
  Logical OR ----- BU,B1,B2

Boolean intersection
  Logical AND ---- BI,B1,B2

Boolean complement
  Logical NOT ---- BC,B1

Boolean rotate --- BR,D1,B1

Boolean shift ---- BS,D1,B1

DECISION (Sec. 5.12)

String equality -- EQ,T1,T2,T3,T4

Greater than ----- GR,D1,D2,T1,T2

AUXILIARY STORAGE (Sec. 5.13)

Store block ------ SB,N1,N2, ...

Fetch block ------ FB,N1

Erase block ------ EB,N1

DIAGNOSTIC (Sec. 5.14)

List names ------- LN,T1

Print form ------- PF,N1

Trace on --------- TN

Trace off -------- TF

HOUSEKEEPING (Sec. 5.15)

Halt ------------- HL

Mode ------------- MO,T1

Arguments are separated by commas and are listed without the syntactic markers #( ::: ) or ##( ::: ) . The first two-letter argument is the mnemonic for the name of the primitive, except for the default call, where the mnemonic argument CL is omitted. The other arguments of different types are symbolized by letters. N1,N2 are name strings. T1,T2 are text strings. D1,D2 are decimal (numeric) strings. B1,B2 are Boolean (octal) strings. Z is the default argument string. ... means that additional arguments of the same kind may follow.

INDEX

*  *  *