UC Irvine
ICS Technical Reports

Title
EXTRAC: Initial Report and User's Manual

Permalink
https://escholarship.org/uc/item/4k13v047

Authors
Hogan, Ken
Kiersey, Dave
Parker, Ben

Publication Date
1972-06-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

18

EXTRAC

Initial Report and User's Manual

by

Ken Hogan
Dave Kiersey
Ben Parker

TECHNICAL REPORT #18 - June 1972

0. Introduction

0.1 Identification

Title: Initial Report on EXTRAC

Version: #2

Date: May 5, 1972

Prepared by: TRAC Implementation Group
Ken Hogan
Dave Keirsey
Ben Parker

Institution: University of California at Irvine

0.2 Scope of Document

This document is a report on EXTRAC, a new implementation of "TRAC 64" on UCI's PDP 10. The goal is to present all information necessary for a person with knowledge of "TRAC 64" to use this new version. Thus, this document is not intended to be a complete description of EXTRAC. Some key ideas, such as the method of functional evaluation, will not be discussed. This is because they are purely "TRAC 64" ideas and descriptions can be found in the references.

0.3 References

1. Mooers,C.N. "TRAC, A Procedure describing language for the Reactive Typewriter", Communications of the ACM, Vol. 9, No. 3, March 1966, pp 215-219

2. Mooers,C.N. And L.P. Deutsch, "TRAC, A Text handling Language", Proceedings of the ACM 20th National Conference, 1965, pp 229-246

0.4 Table of Contents

0.5 Terminology

"TRAC" and "TRAC-64" are the trademarks of Rockford Research Institute, Inc., Cambridge, Mass. in connection with their standard languages.

Whenever the term null argument is referred to, it signifies either "STRING,, STRING)" or "STRING,)" as opposed to no argument which occurs when a function expects some argument and none are supplied. An example of this is the following example where the primitive greater than expects four arguments; #(GR,1,2).

EOF stands for end of file, TTY stands for teletype.

1. Overview

1.1 General Description

EXTRAC is an interactive language designed for nonnumerical problems involving string manipulation. The strings can be any strings of ASCII characters and the manipulations can be of virtually any kind. These manipulations can be defined in the form of macro-like functions which can be called from anywhere, any number of times. The latter gives EXTRAC a natural ability for recursion.

EXTRAC has been designed to be flexible enough for a wide range of problems and situations. The user himself can specify to a great extent what EXTRAC should consider to be an error in his program and how it should be handled. Input and output can take place over a variety of devices. Also, internal storage is dynamically allocated to provide efficiency throughout a wide range of program requirements.

1.2 Historical Background

EXTRAC is basically an implementation of "TRAC 64". However, much has been added to make it more powerful and flexible. The main inspiration for these additions was UMIST, the University of Michigan's version of "TRAC 64". In particular, the adopted forms of the CM, DF, PF, and PAR primitives arose from that language in addition to the idea of protection of dictionary entries. Caltech's TTM was also studied and the idea of character class operations incorporated.

1.3 Learning EXTRAC

The basic ideas of EXTRAC, its logic, syntax, and methods of functional evaluation, are identical to those of "TRAC 64". Thus, rather than detail them in this document, the reader is referred to references 1 and 2.

Anyone with a knowledge of "TRAC 64" should have very little difficulty in using EXTRAC. With few exceptions (see section 4) EXTRAC is completely consistent with the "TRAC 64" primitives. Additional capabilities of EXTRAC are generally described in section 2. Detailed descriptions of both the new primitives and the old "TRAC 64" primitives are given in section 3.

2. Using EXTRAC

This section gives a general description of the features of EXTRAC not found in "TRAC 64". For more detailed information refer to the individual descriptions of the appropriate primitives (section 3).

2.1 Meta Strings

The idea of meta characters used in "TRAC 64" has been expanded as follows. Seven meta strings are used in functional evaluation. They are:

```
MAF - begin active function  - #(
MNF - begin neutral function - ##(
MEF - end function           - )
MRL - right literal          - )
MLL - left literal           - (
MARG - argument delimeter    - ,
MEP - end all functions      - ..)
```

The first six meta strings are consistent with "TRAC 64". The last simply supplies all the MEF's needed to balance the parentheses.

Meta strings may be up to 5 characters long and can be changed using the CM primitive.

As in "TRAC 64", the end of input meta character used by RS is a '. It can be changed using the PAR primitive with the first argument META. The end of input meta character is not a meta string, however, and thus cannot be longer than one character.

2.2 System Flags

System flags have been incorporated to allow the users program to handle several things the user would usually handle himself. They are turned on when the appropriate events occur and may be checked using the SFC primitive. The flags are turned off after being checked by SFC, or when the idler is reinitialized.

The currently implemented flags are:

DEV - signals that an automatic change of input device back to the users TTY occured; this due to an EOF being encountered on the previous input source (see section 2.3)

ERR - signals that an error occured which exceeds the specified error cutoff value (see section 2.8)

2.3 Input Output Devices

The current input and output devices are completely flexible and may be changed at will using the PAR primitive.

When an EOF is encountered on input, the input device is automatically changed back to the users TTY. To allow the user to program checks for such a switch, a system flag, DEV, is turned on when the change is made. This can be detected using the SFC primitive.

2.4 Implicit Calls

In a standard EXTRAC function call of the form:

#(s1,s2,s3,s4,s5)

if s1 names a primitive, then it is called and supplied with s2,s3,s4,s5 as its arguments. If s1 is not a primitive name, then the statement is considered to be an implicit call on CL with arguments s1,s2,s3,s4,s5. Thus,

#(s1,s2,s3,s4,s5)=#(cl,s1,s2,s3,s4,s5) where s1 does not name a primitive.

The same holds true for neutral function calls.

Another alternative is to have CP assumed instead of CL. This can be obtained by using the DF primitive on the individual forms which require this property, or by using PAR,CALL.

2.5 Dictionary Entries

In "TRAC 64", two types of entries exist in the dictionary. The first is the standard name-form association created by DS. The other is the type of entry created by SB and contains information on the location of a block of forms on external storage. Using the form manipulating primitives (eg: SS, CS, ...) on this latter type of entry wil apparently be undefined and produce null values.

In EXTRAC, two more types of these "formless" entries are possible. First, the primitives themselves have been moved into the dictionary. The entry under their names simply contains information identifying them as system primitives and has the jump address for begining execution. This information cannot be reached by the user, but it can be deleted from the dictionary, letting the user use the names of system primitives for his own functions. The other new type of "formless" dictionary is the character class. These entries contain highly packed information regarding the elements of the class and can only be handled using the character class primitives.

If any general primitive which needs a form encounters one of these "formless" entries, it will consider it an error (see section 2.8). On the other hand, these entries are perfectly normal in any operation which does not actually have to use the form, such as redefinition, deletion, etc.

2.6 Protection of Dictionary Entries

Protection is handled by associating each entry to a class and associating protection levels to each class. There are ten classes available for this purpose: PRIM, USER1,...,USER9. The classes can be protected (using PAR) against one or more of the following operations: segmentation, redefinition, and deletion.

When a primitive is used to make a dictionary entry, its class will be decided by the current default class. This default class is simply the last class used in a PAR primitive, or USER1 at the start. The only exception to this is DF, which allows the class of the new entry to be stated explicitly.

At the start, all the classes are unprotected except PRIM, the class of all the primitives, which is completely protected. For this reason, primitives cannot be accidently erased. They can only be deleted after their protection has been removed using PAR.

2.7 External Storage of Dictionary Entries

For numerous reasons, it is often useful to remove entries from the dictionary and then re-enter them if and when they are needed later. There are two general ways to do this.

The first is by the SB and FB primitives. SB stores the specified entries on the specified device so they can be recovered later using the FB primitive. Any type of entry can be saved in this manner.

The other method can be used only to save forms, i.e. as created using DS, DF, etc. It employs the PF and PAR, OUTPUT primitives. The PF primitive essentially outputs a EXTRAC DF command which, when read back in, will completely redefine the desired forms.

With both methods, the entries can be saved on any device from which they may be read back in, such as disk, paper tape, etc.

2.8 Error Handling

In pure "TRAC 64", there are essentially no errors. This is due to the lack of restrictive syntax and the policy of returning null values in almost every illegal situation. In EXTRAC, more powerful and complex primitives (eg, PAR) have produced greater syntax requirements and thus, more potential for error. To give the user some choice as to what should be treated as an error in his program, the following system is employed.

Each potential error has a numerical severity assigned to it; the higher the number, the more severe the error. Also, the user can establish an error cutoff value. Then, when a potential error is detected, its severity is compared to the cutoff value. If it exceeds the cutoff value, an error has occured. If not, the "error" is ignored and a null value is returned.

The exact severities for the individual error conditions can be found in section 5.1.

The user is also given four options on what should be done when an error does occur:

1) cease program execution; return the idler
2) continue execution with error message
3) continue execution with trace on.
4) Continue execution without error message

PAR is used to establish which of the three will be done. In all cases, when an error does occur, an error message is output (see section 5.1).

Finally, the user may program checks for errors. When an error occurs, the system flag ERR is turned on. This can be detected using the SFC primitive.

2.9 Storage Management

Most of the details on storage handling are automatic. Available core is allocated dynamically for most efficient use. When program requirements exceed available space, more space is requested from the monitor. If program needs become lesser later, extra space is reurned to the monitor.

If the user wants, he may specify minimum and maximum values for his core size and also the increments in which additional K should be requested from the monitor. He may also request notification of changes in his core size. Finally, he can either have his program automatically killed when it tries to use more than its assigned maximum or he can have several options presented to him, if and when this occurs. All of these features are handled by the PAR,CORE primitive. Details can be found in section 3.

3. Extrac Primitives

3.1 Comparison with "TRAC 64"

3.1.1 "TRAC 64" Primitives Unchanged

DS,SS,SB,FB,RS,RC,PS,DD,DA,CR,CS,CC,CN,IN,CL,EQ,
GR,AD,SU,ML,DV,BU,BI,BC,BR,BS,BYE

3.1.2 "TRAC 64" Primitives Changed

PF extended
TN,TF combined into TP
CM revised, calling PAR,META in EXTRAC is equivalent to the CM call in "TRAC 64"
LN extended
EB not implemented
SD replaced by calls on PAR,INPUT and PAR,OUTPUT

3.1.3 New primitives

The following primitives are implemented in EXTRAC but not in "TRAC 64"

LB Link Block - creates header block in dictionary for external block previously created by SB PAR sets a variety of system parameters such as input device, output device, error severities, etc.

DF Define Form - as in UMIST, similar to DS in "TRAC 64"

AP Append - adds a character string to an existing form

SC Segment and Count - same as SS but in addition it returns the number segmentations made

SKS Skip Segment - changes the form pointer as in CS

SKN Skip N characters - changes the form pointer as in CN

SKI SKip Initial - changes the form pointer as in IN

CB Call Before - similar to CL except that it takes everything before the form pointer instead of after

CP Call Procedure - similar to CP in UMIST

DCL Define character CLass - as described in TTM

DNC Define Negative Class - described in TTM

CCL Call character CLass - described in TTM

FN Find Name - determines whether or not a name is defined

SFC System Flag Check - see section 2.2

REM REMainder - gives the remainder from an integer division

MS Meta String - returns the characters in the requested meta string

ATO ASCII To Octal

OTA Octal To ASCII

3.2 Index to Detailed Descriptions

The detailed descriptions of the primitives are organized in the following manner.

1. Input-Output:
   - RS - Read string
   - RC - Read character
   - PS - Print string
   - SB - Store block
   - FB - Fetch block
   - LB - Link block
   - OF - Output form
   - PAR,INPUT - Input specification
   - PAR,OUTPUT - Output specification

2. Definition oriented:
   - DS - Define string
   - DF - Define form
   - AP - Append
   - SS - Segment string
   - SC - Segment and count
   - DD - Delete definition
   - DA - Delete all
   - TFP - Truncate at form pointer
   - CPY - Copy
   - APC - Add to protection class
   - PAR,CLASS - Class specification

3. Call oriented:
   - CL - Call
   - CB - Call before
   - CP - Call procedure

4. Form pointer oriented:
   - CR - Call restore
   - CS - Call segment
   - CC - Call character
   - CN - Call n characters
   - IN - Initial
   - SKS - Skip segment
   - SKN - Skip n characters
   - SKI - Skip initial

5. Character class:
   DCL - Define character class
   DNC - Define negative character class
   CCL - Call character class

6. Decision functions:
   EQ - Equals
   GR - Greater than
   FN - Find name
   SFC - System flag check

7. String oriented functions:
   CNT - n characters
   FL - Flip
   ISS - Isolate substring

8. Arithmetic functions:
   AD - add
   SU - subtract
   ML - multiply
   DV - divide
   REM - remainder

9. Boolean functions:
   BU - Boolean union
   BI - Boolean intersection
   BC - Boolean complement
   BR - Boolean rotate
   BS - Boolean shift

10. System status:
   MS - Meta string
   CM - Change meta string
   PAR (meta,bufkl,hash,core,quote,call)

11. Diagnostics:
   LN - List names
   TR - Trace
   PAR,ERROR - Error level specification

12. Miscellaneous:
   ATO - ASCII to octal
   OTA - Octal to ASCII
   OTD - Octal to decimal
   DTO - Decimal to octal
   BYE - Bye

3.3 Detailed Descriptions

The format of the detailed descriptions will be as follows:

NAME: The actual primitive name and the source of the abbrevation.
FORM: The actual form of the function and its arguments
VALUE: The value of the function upon evaluation
SIDE EFFECTS: Effects produced during evaluation
ERROR TYPES: See Section 5.1.2
EXAMPLES and COMMENTS: Additional information

3.3.1 Input - Output

RS NAME: Read String
FORM: #(RS)
VALUE: A character string which is read in from the current input device.
SIDE EFFECTS: none
ERROR TYPES: I9,I10
COMMENTS: End of input meta character used only to determine end of input string and thus, is not passed with the rest of the character string, e.g. evaluting a #(rs); user types ABC' the value of #(RS) is ABC.

RC NAME: Read character
FORM: #(RC)
VALUE: A character which is read in from the current input device
SIDE EFFECTS: none
ERROR TYPES: I9,I10
COMMENTS: Since RC only take one character it does not use the end of input meta character. It can thus be used to input the end of input meta character. characters with octal value 175, 176 and 177 are not read in.

PS NAME: Print string
FORM: #(PS,string)
VALUE: null
SIDE EFFECTS: The string is outputted over the current output device
ERROR TYPES: none

SB NAME: Store block
FORM: #(SB,n,name1,name2, ...)
VALUE: null
SIDE EFFECTS: The dictionary entries named by name1,name2, are stored in the disk file specified by the header block named by n. If n does not name a header block, one is created and a temporary disk file is created to store the dictionary entries named. Once stored, the entries are deleted from the dictionary.
ERROR TYPES: I5,I6
COMMENTS: 1) if the header block already exists, the original contents of its associated disk file are lost. 2) the temporary disk file created are of (055) protection.
EXAMPLES: see Section 3.3

FB NAME: Fetch block
FORM: #(FB,name1,name2, ...)
VALUE: null
SIDE EFFECTS: The header blocks named by name1, name2, are used to return the one-time dictionary entries from external storage to the dictionary.
ERROR TYPES: D3,P2,I3,I7
COMMENTS: 1) Neither the contents of the disk file or its header block in the dictionary are changed in any way.
EXAMPLES: See Section 3.3

LB NAME: Link block
FORM: #(LB,n,dev:file.ext)
VALUE: null
SIDE EFFECTS: A dictionary entry (a header block) is made with name n. It contains all the information needed be FB to recover dictionary entries stored by SB on the device named.
ERROR TYPES: P2,S2,I1,I2,I3,I4,I5,I6,I7,I12
COMMENTS: This primitive can be used to explicitly specify where SB should put its data. It can also be used whenever a SB type disk file exists but the dictionary entry (header block) needed be FB to access it does not. The latter could occur either because of deletion or because of signing off the system and coming back on later.

OF NAME: Output form
FORM: #(OF,n1,n2, ...)
VALUE: null
SIDE EFFECTS: For each of the arguments which name forms in the dictionary, an EXTRAC statement using DF is outputted over the current output device. These statements are such that, if read back in by EXTRAC interpreter, the forms would be completely defined including segment gaps, class, and form pointer.
ERROR TYPES: D1,D2,I6
COMMENTS: OF changes the dictionary in no way.
EXAMPLES: Say n1 names a form 12345 with a segment gap 2 after the 4 and the form pointer a the third character and belonging to class USER1. Then, #(DF,(n1),(12345),USER1,c1,3,2,4) is outputted.

PAR,INPUT
NAME: Parameter Input
FORM: #(PAR,INPUT,dev:file.ext).
VALUE: null
SIDE EFFECTS: The current input device is changed to the source specified by dev:file.ext.
ERROR TYPES: S1,S2,I1,I3,I4,I6,i12
COMMENTS: If input "device" is a disk file, it is automatically changed back to the users TTY when the EOF is encountered.
EXAMPLES: #(PAR,INPUT,dsk:file.ext) #(PAR,INPUT,ptr:)

PAR,OUTPUT

NAME: Parameter Output

FORM: #(PAR,OUTPUT,dev:file.ext)

VALUE: null

SIDE EFFECTS: Same as PAR,INPUT except changes the current output device

ERROR TYPES: S1,S2,I1,I2,I4,I5,I6,I7,I12

COMMENTS: Note, that input and output devices are independent. The TTY could be the input source while a disk file could be the output source.

EXAMPLES: #(PAR,OUTPUT,file.ext) #(PAR,OUTPUT,tty:)

3.3.2 Definition Oriented

DS NAME: Define string
FORM: #(DS,name1,string)
VALUE: null
SIDE EFFECTS: An entry is made in the dictionary with name name1 and form of string. The current default class determines the class of the entry.
ERROR TYPES: P2
COMMENTS: If the name already exists in the dictionary, the old entry is replaced with the new form.
EXAMPLES: #(DS,name,form) #(DS,abc,12345)

DF NAME: Define form
FORM: #(DF,n,f,class,call,fp,s1,p1,s2,p2,...)
VALUE: null
SIDE EFFECTS: An entry is made in the dictionary with name n and form f. Class can be one of [PRIM,USER1,...,USER9] and is used to determine the class of the entry. If class is null the current default class is used. If call is cp or cl, it is used to determine whether implicit call or implicit call procedure is used with the form. If fp is between 0 and the number of characters in the form, it is used to position the form pointer. If not, the form pointer is set to the beginq. The remaining pairs of arguments specify segment gaps. The first argument of the pair specifies the segment gap number. The other specifies the position in the form be giving the number of the characters it follows. Note the positions must be in ascending order.
ERROR TYPES: D5,P2,S2
COMMENTS: Due to the complexity of this primitive, it is seldom efficient to use it in programming. It was designed mainly to be produced by the OF primitive and used in saving forms on external storage.

AP NAME: Append
FORM: #(AP,n,string)
VALUE: null
SIDE EFFECTS: String is appended onto the end of the form named by n.
ERROR TYPES: D1,D2,P2
COMMENTS: Segment gaps are unchanged
EXAMPLES: If name is abcdefg then #(AP,name,!!!) changes name to abcdefg!!!

SS NAME: Segment String
FORM: #(SS,n,s1,s2,s3,...)
VALUE: null
SIDE EFFECTS: In the form named by n, every occurance of s1 is replaced by a segment gap #1. Then every occurance of s2 is replaced by a segment gap #2. This continues for all the

arguments supplied. All of these replacements take place from the position of the form pointer on. The part of the form before the form pointer is not changed.
ERROR TYPES: D1,D2,P1
EXAMPLES: Say abc names form abcdef and the form pointer points to the a then #(SS,abc,c,f) results in abc naming ab@1de@2 then #(SS,abc,,,,,a) results in abc naming @5b@1de@2 where @i denotes a segment gap #i.

SC NAME: Segment and count
FORM: #(SC,name,s1,s2, ...)
VALUE: Number of segmentations made
SIDE EFFECTS: Same as SS
ERROR TYPES: D2,P1
COMMENTS: Same as SS except it returns an ASCII string of digits equal to the number of segments made.
EXAMPLES: Say a equals 12345, then #(SC,a,3) has the value 1, and #(SC,a,c) has the value 0.

DD NAME: Delete definition
FORM: #(DD,name1,name2, ...)
VALUE: null
SIDE EFFECTS: The dictionary entries named are deleted.
ERROR TYPES: D1,P3
COMMENTS: To delete the null name, DD must be explicitly given a null argument. Thus, #(DD) will not delete the null name but #(DD,) will. Delete definition works within bounds of protection, and will not delete a protected name.

DA NAME: Delete all
FORM: #(DA)
VALUE: null
SIDE EFFECTS: All dictionary entries not protected against deletion are deleted
ERROR TYPES: none

TFP NAME: Truncate at form pointer
FORM: #(TFP,name)
VALUE: null
SIDE EFFECTS: The form is changed to be everything before the form pointer and anything after it is deleted.
ERROR TYPES: D1,D2,P2
COMMENTS: Segment gaps are not effected, and does not change the form pointer.
EXAMPLES: If abc has the form 123456 and the form pointer is between 3 and 4 the #(TFP,abc) results in abc with a form of 123.

CPY NAME: Copy
FORM: #(CPY,oldname,newname)
VALUE: null
SIDE EFFECTS: The form of the oldname is copied and is the form of the newname.
ERROR TYPES: D1,P2
COMMENTS: Everything is copied, including the class, segment gaps, form pointer, etc.

APC NAME: Add to protection class
FORM: #(APC,class,name1,name2, ...)
VALUE: null
SIDE EFFECTS: The definitions are changed form the previous classes they were in to the specified class.
ERROR TYPES: D1
EXAMPLES: Suppose one wanted to save just one function but wanted to delete all others. Then #(APC,PRIM,func1) #(DA) #(APC,USER1,func1) would do it.

PAR,CLASS
NAME: Parameter class
FORM: #(PAR,CLASS,user,REDEF,SEG,DEL)
VALUE: null
SIDE EFFECTS: If user specifies a class (ie. PRIM,USER1,...USER9), then that class becomes the current default class (as used in DS). Then all protection of the class is removed and reestablished in the following way. If any of the following arguments are SEG, REDEF, or DEL, then the class is protected against segmentation, redefinition and deletion, respectively.
ERROR TYPES: S2
EXAMPLES: #(PAR,CLASS,USER7,SEG,REDEF) changes the default class to USER7 and protects all forms in USER7 from segmentation and redefinition. #(PAR,CLASS,PRIM) changes the default class to PRIM and removes all protection from the class.

3.3.3 Call Oriented

CL NAME: Call
FORM: #(CL,name,s1,s2, ...) or #(name,s1,s2, ...)
VALUE: The form named by name from the form pointer on with segment gaps number 1 replaced with s1, segment gaps number 2 replaced with s2, etc.
SIDE EFFECTS: none
ERROR TYPES: D1,D2
COMMENTS: Usually the primitive CL is left out and the implicit call form is used.
EXAMPLES: If name has form ab@1cd@1ef and the form pointer is between the first segment gap and c, then #(cl,name,!!) gives cd!!ef.

CB NAME: Call before
FORM: #(cb,name,s1,s2, ...)
VALUE: The form named by name from the beginning to the form pointer, with segment gaps being replaced with the corresponding arguments.
SIDE EFFECTS: none
ERROR TYPES: D2
COMMENTS: Same as call, except instead of everything after the form pointer, it gives everything before the form pointer.
EXAMPLES: With name as in the CL example: #(CB,name,!!) produces ab!!.

CP NAME: Call procedure
FORM: #(CP,name,s1,s2, ..)
VALUE: Result of doing an active call, then evaluating result one more time if an active CP.
SIDE EFFECTS: None
ERROR TYPES: D1,D2
COMMENTS: The neutral CP same as a neutral CL, and an active CP takes the value produced by active CL and reevaluates it.
EXAMPLES: Say n1 names the form ##(CL,n2) and n2 names the form ##(CL,n3) and n3 names the form abc. Then, ##(CL,n1) has the value ##(CL,n2);
#(CL,n1) has the value ##(CL,n3);
##(CP,n1) has the value ##(CL,n3);
#(CP,n1) has the value abc.

3.3.4 Form Pointer Oriented

All the primitives of this type manipulate the form pointer. Those primitives (e.g. CS,CC, etc.) which can return their last argument as a default value, always return it actively, whether or not the call was active or neutral.

CR NAME: Call restore
FORM: #(CR,n1,n2, ...)
VALUE: null
SIDE EFFECTS: The form pointers of all the the forms named by the arguments are restored to the first character of the form.
ERROR TYPES: D1,d2
COMMENTS: If abc names 12345 and the form pointer points to the 4 then after a #(CR,abc) the form pointer points to the 1.

CS NAME: Call segment
FORM: #(CS,n1,d)
VALUE: The part of the form named by n1 from the form pointer to either the next segment gap, or the end of the form, whichever comes first. If the form pointer is already at the end of the form, then d is returned (actively) instead.
SIDE EFFECTS: The form pointer is advanced either to after the segment gap or to the end of the form.
ERROR TYPES: D1,D2
COMMENTS: Say ab names form a@1bc@3@1e where @i is segment gap #i if the form pointer is at a to begin with, then the following sequence of results will be obtained.
first #(CS,ab,alt) produces a
second #(CS,ab,alt) produces bc
third #(CS,ab,alt) produces
fourth #(CS,ab,alt) produces alt
Then, nothing but "alt" will be obtained; a #(CR,ab) would be necessary to return the form pointer to the beginning.

CC NAME: Call character
FORM: The character the form pointer points to in the form of name. If the form pointer is at the end of the form, then d is returned (actively) instead.
VALUE: The form pointer is advanced to the next character
SIDE EFFECTS: D1,D2
ERROR TYPES: Segment gaps are skipped
COMMENTS: With ab as in the CS example:
first #(CC,ab,alt) produces a
second #(CC,ab,alt) produces b
third #(CC,ab,alt) produces c
fourth #(CC,ab,alt) produces e
fifth #(CC,ab,alt) produces alt

3.3.5 Character Class

DCL NAME: Define character class
FORM: #(DCL,cclass,string)
VALUE: null
SIDE EFFECTS: A character class with name cclass is defined. The class will consist of every ASCII character which appears in string.
ERROR TYPES: P2,S1
COMMENTS: Note that character classes are formless entries, the string cannot be recovered, and it's elements can only be accessed using the other character class primitives.
EXAMPLES: #(DCL,numbers,0123456789) defines a character class with name numbers and which consists of the digits 0 to 9.

DNC NAME: Define negative class
FORM: #(DNC,nclass,cclass)
VALUE: null
ERROR TYPES: A character class with name nclass is defined. The class will consist of every ASCII character which does not appear in the character class named by cclass.
ERROR TYPES: D4,P2,S1
COMMENTS: see DCL comments
EXAMPLES: #(DNC,non-numbers,numbers) defines a character class with the name non-numbers and whose elements are all ASCII characters except those 0 thru 9. (assuming numbers is defined as in the DCL example)

CCL NAME: Call character class
FORM: #(CCL,name,cclass,d)
VALUE: Essentially the same as IN except instead of a string it is a set of characters, of which anyone produces a match.
SIDE EFFECTS: The form pointer changes as in IN.
ERROR TYPES: D1,D2,D4,S1
COMMENTS: If stuff names a form: abc1def2ghi and numbers is a character class with elements 0 thru 9 then, #(CCL,stuff,numbers,alt) produces
abc the first time, def the second time, alt the third time.

3.3.6. Decision Functions

EQ NAME: Equals
FORM: #(EQ,arg1,arg2,true,false)
VALUE: true if arg1 is identical to arg2, otherwise false.
SIDE EFFECTS: none
ERROR TYPES: S1
COMMENTS: Program branches can thus be obtained by having true and false be functions calls.
EXAMPLES: #(EQ,abc,abc,yes,no) gives yes.
#(EQ,abc,ab,yes,no) gives no.

GR NAME: Greater than
FORM: #(GR,arg1,arg2,true,false)
VALUE: true if the numerical value of arg1 is greater than the numerical value of arg2.
SIDE EFFECTS: none
ERROR TYPES: S1
EXAMPLES: #(GR,5,3,yes,no) evaluates to yes
#(GR,-5,3,yes,no) evaluates to no
#(GR,ab-7cd7ef-5,3,yes,no) evaluates to no
#(GR,a5,b,yes,no) evaluates to yes
#(GR,c,-7,yes,no) evaluates to yes

FN NAME: Find name
FORM: #(FN,name,true,false)
VALUE: true if name names a dictionary entry, false if not in dictionary.
SIDE EFFECTS: none
ERROR TYPES: S1

CSF NAME: Check system flag
FORM: #(CSF,flag,true,false) SYNTAX: "flag" = DEV, ERR
VALUE: The system flag specified by flag is checked. If it is on, then the value is true, if it is off false is the value.
SIDE EFFECTS: If the flag checked was on, it is turned off.
ERROR TYPES: S1,S2
COMMENTS: If true and false strings are null, the appropriate flag is turned off and null returned.
EXAMPLES: If ERR, the error flag is on, then #(CSF,ERR,(#(continue)),(#(error))) will execute the function "error". If ERR was off then "continue" would be executed.

3.3.7 String oriented functions

CNT NAME: Count
FORM: #(CNT,string)
Value: The number of characters in the string
SIDE EFFECTS: none
ERROR TYPES: S1
EXAMPLES: #(CNT,abc) value is 3

FL NAME: Flip
FORM: #(FL,string)
Value: The reversed string
SIDE EFFECTS: none
ERROR TYPES: S1
EXAMPLES: #(FL,abcd) the value is dcba

ISS NAME: Isolate substring
FORM: #(ISS,string,num1,num2)
Value: The first num1 characters of string after the first num2 characters, or null if there are not enough characters.
SIDE EFFECTS: none
ERROR TYPES: S1,S2
EXAMPLES: #(ISS,abcdef,2,3) gives de

3.3.8. Arithmetic Functions

All the arithmetic functions work with the decimal numerical value of a character string. This value is determined by taking the numeric string. The negative sign is also considered. If there are no digits, then the string has value zero. Thus, 53 = 53, -53 = -53, abc-53 = -53, a5b753 = 753, a5b7bcd = 0, f = 0.

AD NAME: Add
FORM: #(AD,num1,num2,d)
Value: The numerical value of num1 plus the numerical value of num2. If an alpha-numeric string prefixed the numeric value of num1, it will be used to prefix the result however, if overflow occurs, d is returned actively.
SIDE EFFECTS: none
ERROR TYPES: S1
EXAMPLES: #(AD,3,4) produces 7, #(AD,cntr3,1) produces cntr4, #(AD,1,cntr3) produces 4.

SU NAME: Subtract
FORM: #(SU,num1,num2,d)
Value: The numeric value of num1 minus the numeric value of num2. The prefix string of num1 is used to prefix the result. However, if overflow, d is returned actively instead.
SIDE EFFECTS: none
ERROR TYPES: S1

ML NAME: Multiply
FORM: #(ML,num1,num2,d)
Value: Similiar to AD and SU; either the product returned or d if overflow.
SIDE EFFECTS: None
ERROR TYPES: S1

DV. NAME: Divide
FORM: #(DV,num1,num2,d)
Value: Similar to other arithmetic primitives, either the result of dividing num1 by num2 or if zero divisor then d is actively returned.
SIDE EFFECTS: none
ERROR TYPES: S1

REM NAME: Remainder
FORM: #(REM,num1,num2,d)
Value: Like divide but the remainder
SIDE EFFECTS: none
ERROR TYPES: S1

3.3.9 Boolean Functions

The boolean functions work with the octal numerical value of a character string. The octal numbers represent bit strings and are obtained from the end of an alpha-numeric string in the same way as the decimal value are obtained. Thus, abc777 = 777, 987654 = 7654, -123 = 123, ab89 = 0.

BU NAME: Boolean union
FORM: #(BU,oct1,oct2)
Value: The union of the octal strings oct1 and oct2. (prefixed by the prefix string of oct1)
SIDE EFFECTS: none
ERROR TYPES: S1

BI NAME: Boolean intersection
FORM: #(BI,oct1,oct2)
Value: The intersection of the octal strings oct1 and oct2. (prefixed by prefix string of oct1)
SIDE EFFECTS: none
ERROR TYPES: S1

BC NAME: Boolean compliment
FORM: #(BC,oct1)
Value: The compliment of the octal string oct1. (prefixed with oct1's prefix)
SIDE EFFECTS: none

BR NAME: Boolean rotate
FORM: #(BR,oct1,num1)
Value: The octal number resulting from the octal string oct1 being rotated num1 times. It is rotated left if num1 is positive, and rotated right if negative.
SIDE EFFECTS: none

BS NAME: Boolean shift FORM: #(BS,oct1,num1)
Value: The octal number resulting from the octal string oct1 being shifted num1 times. It is shifted left if num1 is positive, and shifted right if negative.
SIDE EFFECTS: none
ERROR TYPES: S1

3.3.10 System Status

MS NAME: Meta string
FORM: #(MS,argl,string)
SYNTAX: argl must be one of the following; MEF,MRL,MLL,MARG,MEP; (see section 2.1 for description)
VALUE: The meta string specified by argl.
SIDE EFFECTS: none
ERROR TYPES: S2
EXAMPLES: Given the standard meta strings,
##(MS,MEF) produces)
##(MS,MNF) produces ##(

CM NAME: Change meta
FORM: #(CM,argl,stringl,arg2,string2,...)
VALUE: null
SIDE EFFECTS: The meta string specified by argl is changed to stringl. This is repeated for each pair of arguments given. See section 2.1
COMMENTS: Meta strings may be at most five characters long. If an attempt is made to make them longer five characters will be used.
EXAMPLES: #(CM,MAF,![,MARG,/,MLL,") changes the MAF to ![, the MARG to / and the MLL to ". To change them back: ![CM/MAF/#(/MLL/(/MARG/,)

PAR,META
NAME: Parameter meta FORM: #(PAR,META,char)
VALUE: null
SIDE EFFECTS: The end of input meta character (used in RS) is changed to the first character in char.
ERROR TYPES: S1
EXAMPLES: #(PAR,META,!) changes the end of input meta character to an ! so if evaluating an #(RS) and the user types abc! then the value of it is abc.

PAR,BUFKL
NAME:Parameter buffer kill
FORM: #(PAR,BUFKL,char)
VALUE: null
SIDE EFFECTS: Changes the character that reinitializes the buffer to the first character of char.
ERROR TYPES: S1
COMMENTS: The default is !U

PAR,QUOTE NAME: Parameter quote
FORM: #(PAR,QUOTE,char)
VALUE: null
SIDE EFFECTS: Changes the quote character to the first character

of char.
ERROR TYPES: S1
COMMENTS: Default is "; The quote character causes the character which follows it to be passed without being scanned.

PAR,CORE
NAME: Parameter core
FORM: #(PAR,CORE,mess,min,max,autok)
VALUE: null
SIDE EFFECTS: If mess = yes then a message is output when core allocation is changed. If mess = no then no message is output. If incr is non-null then the increment of core is changed to the number (in K). If min is non-null then the minimum core allocated in the low segment to the user is the number (in K) and no core is released below that. (minimum is at least 2) If max is non-null then the maximum core allocated in the low segment is changed to the number (in K). If autok = yes then whenever the maximum is reached the the EXTRAC automatically restarts (puts in the idler) If autok = no then whenever the maximum is reached then a message is typed and asks for a specific response. K restarts, C continues but does not get anymore core and does the best it can. (after a several C's EXTRAC will die if no more core becomes available.), G gets another increment of core.
ERROR TYPES: S1
COMMENTS: The default is #(PAR,CORE,yes,1,2,100,no)

PAR,CALL
NAME: Parameter call
FORM: #(PAR,CALL,argl)
VALUE: null
SIDE EFFECTS: If argl = cl then whenever a form is defined, it will defined so #(funcl,...) = #(CL,funcl,...) Whereas if argl = cp then whenever a form is defined it will be defined so #(funcl,...) = #(CP,funcl,...).
ERROR TYPES: S1

3.3.11 Diagnostics

LN NAME: List names
FORM: #(LN,delim,class1,class2,...)
VALUE: A list of all the names of the entries in the dictionary specified by class with delim preceeding each one. If delim is null, a carriage return, line feed is used to delimit them. What names to be listed are determined as follows: if there is no argument for class1 then all classes but PRIM are listed, if class1 is a null argument then all dictionary entries are listed including PRIM, if a specific class is specified by class1, then that class, and any others named, are listed.
SIDE EFFECTS: none
ERROR TYPES: S2
EXAMPLES: ##(LN) ##(LN,) all list everything but primitives. ##(LN,,) lists everything. #(LN,##(MS,MARG),USER7) lists all in user7 with marg as a delimiter.

TR NAME: Trace
FORM: #(TR,arg1,arg2)
VALUE: null
SIDE EFFECTS: If arg1 = "ON" or "OFF" then trace is turned on or off respectively. Otherwise, trace is turned to the opposite of its current state, ie. turned on if off, turned off if on. If arg2 = pause, trace stops after each evaluation. If arg2 = nopause, then trace prints out the evaluations but does not wait for a confirm to go on.
ERROR TYPES: none
COMMENTS: The trace feature is comparable to that of TRAC 64. Hitting the rubout key causes evaluation. Hitting !N continues but with trace turned off. Anything else causes reinitialization.

PAR,ERROR
NAME: Parameter error
FORM: #(PAR,ERROR,num1,cond)
VALUE: null
SIDE EFFECTS: If num1 is non-null the number becomes the value for the error cutoff value (The latter is what is compared to the numerical severity of potential errors to see if they should be treated as actual errors or not)

What the system will do on detecting an error is re-established: if cond is k, then the idler is restored. If cond is m, program execution continues with a error message printed out, and set the system flag. If cond is t, program execution continues with trace on. If cond is n, the program continues, the system flag set and no message will be typed out.
ERROR TYPES: S1,S2
COMMENTS: Initial values are equivalent to #(PAR,ERROR,0,k)

3.3.11 Miscellaneous

ATO NAME: ASCII to octal
FORM: #(ATO,char)
VALUE: The octal digits representing the ASCII value of the first character of char.
SIDE EFFECTS: none
ERROR TYPES: S2
EXAMPLES: #(ATO,1) has the value 61, #(ATO,a) has the value 101

OTA NAME: Octal to ASCII
FORM: #(OTA,octl,oct2, ...)
VALUE: The character determined by the octal value of octl, concatentated with all succeeding converted arguments all of which the octal value is taken mod 200 octal.
SIDE EFFECTS: none
COMMENTS: The octal values of 175,176 and 177 are ignored.
EXAMPLES: #(OTA,101,102) has the value AB #(OTA,3) has the value !C

OTD NAME: Octal to decimal
FORM: #(OTD,octl)
VALUE: The decimal digits that are the equivalent to the octal number octl, (prefixed by character string if any)
SIDE EFFECTS: none
EXAMPLES: #(OTD,cnt20) gives cnt24

DTO NAME: Decimal to octal
FORM: #(DTO,numl)
VALUE: The octal digits that are the equivalent to the decimal number numl, (prefixed by character string if any)
SIDE EFFECTS: none
ERROR TYPES: S1
EXAMPLES: #(DTO,cnt19) gives cnt23

BYE NAME: Bye
FORM: #(BYE,message)
VALUE: null
SIDE EFFECTS: Message is output over the current output device and a return is made to the monitor
ERROR TYPES: none

4. Compatability with "TRAC 64"

The following EXTRAC primitives are the only ones whose effects differ from those given in "TRAC 64".

PF Still null valued but it outputs a DF statement which if read back in would completely define the form being printed. It has been changed to provide another way of saving forms. It differs from SB in that it outputs files in ASCII that can then be edited indepedently of EXTRAC. Its disadvantage is that it cannot be used to store all dictionary entries but only those composed completely of text. When used in conjunction with the PAP primitive the form being printed can be output to any device or file that can take ASCII text.

FB This primitive is not implemented in EXTRAC.

CM It has been changed to conform with the idea of meta strings and no longer changes the end of input meta character.

TN,TF Their functions are handled by the EXTRAC primitive TP.

DA,DD They now operate only within established protection levels.

LN It has been modified to give options on what classes of entries should be listed.

For more detailed descriptions of the new functions of the above primitives the user should refer to section 3.

5. Appendix

5.1 Error Messages

5.1.1 Format

Error messages have the form:

```
* * ERROR MESSAGE
?#(FUNCTION,A1,A2_,A3)?
```

Where A2 is the probable argument that cause the error

5.1.2 Error Types

D Dictionary oriented
P Protection oriented
S Syntax oriented
I Input-output oriented

Code	Message	Severity
D1	NAME NOT DEFINED	7
D2	NO FORM ASSOCIATED WITH NAME	23
D3	DICTIONARY ENTRY NOT CREATED BY SB OR LB	21
D4	NOT A CHARACTER CLASS	23
D5	SEGMENT GAP IN WRONG ORDER	23
P1	PROTECTED AGAINST SEGMENTATION	15
P2	PROTECTED AGAINST REDEFINITION	15
P3	PROTECTED AGAINST DELETION	15
S1	MISSING ARGUMENTS	3
S2	UNACCEPTABLE ARGUMENTS	27
I1	FILE NOT FOUND	35
I2	NO SUCH PROJECT PROGRAMMER NUMBER	35
I3	FILE READ PROTECTED	35
I4	ILLEGAL FILE NAME	35
I5	FILE WRITE PROTECTED	35
I6	FILE WAS BEING MODIFIED	35
I7	DEVICE NOT AVAILABLE	35
I8	OUTPUT ERROR - IMPROPER MODE	35
I9	INPUT ERROR - DEVICE DETECTED ERROR	35
I10	INPUT ERROR - DATA ERROR	35
I11	INPUT ERROR - BLOCK TOO LARGE	35
I12	ERROR SYNTAX OF FILE NAME SPECIFICATION	35