Department of Applied Analysis
and Computer Science
Technical Report CSTR 1004
July, 1970

THE DIALATOR SYSTEM
TRAIZE PROGRAMMER'S GUIDE

by

Doron J. Cohen, Paul M. Fawcett
Eric G. Manning & Larry Smith

Faculty of Mathematics

University of Waterloo

Waterloo, Ontario

Canada

Department of Applied Analysis

&

Computer Science

Department of Applied Analysis
and Computer Science

Technical Report CSTR 1004
July, 1970

THE DIALATOR SYSTEM
TRAIZE PROGRAMMER'S GUIDE

by

Doron J. Cohen, Paul M. Fawcett
Eric G. Manning & Larry Smith

We wish to thank the Defence Research Board of Canada, the Northern Electric Research & Development Laboratories, Ottawa, and the Faculty of Mathematics of the University of Waterloo for financial, technical and moral support which made this system of programs possible.

Other Manuals in this series are:

1) TRAIZE User's Guide

2) FAUST User's Guide

3) General Programmer's Guide

4) FAUST Programmer's Guide

5) File System User's Guide

6) File System Programmer's Guide

PROGRAMMER'S GUIDE - DIALATOR SYSTEM

TRAIZE *

This is a programmer's guide to program TRAIZE of the DIALATOR system. This guide should not be studied before the TRAIZE User's Guide has been read.

The Guide is laid out as follows:

Section	1)	Introduction
Section	2)	Program PARSER
Section	3)	Program BUILDER
Section	4)	Subroutine CCINT
Section	5)	Subroutine CRESSA
Section	6)	Subroutine LEOR
Section	7)	Subroutine NECAR
Section	8)	Subroutine REBL
Section	9)	Subroutine SCAN

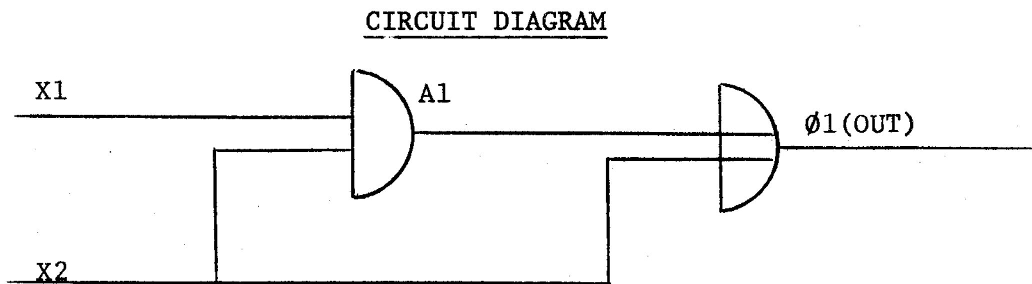
* The General Programmer's Guide should be read before this manual is studied.

Section 1) INTRODUCTION

TRAIZE is composed of two main routines PARSER and BUILDER. Descriptions of PARSER and BUILDER appear in that order, followed by descriptions of the subroutines called by these two programs. Those called subroutines which do not appear are contained in the Programmer's Guide for the file system.

The actions of TRAIZE will be traced using the circuit described below.

CIRCUIT DIAGRAM



CIRCUIT 2

Input Description for TRAIZE program

```
% RUN:      TRAIZE CIRCUIT = CIRCUIT 2 , DISP = NO;
            INPUTS    (2)    X1, X2;
            OUTPUTS   (1)    Ø1;
X1:         INP     A1;
X2:         INP     A1, Ø1;
A1:         AND     X1, X2/Ø1;
Ø1: ØUT:        ØR      A1, X2;
                END CIRCUIT 2 ;
```

Section 2) PROGRAM PARSER

This procedure scans the circuit description of the specified circuit, breaks down the statements into syntactic units, detects syntax errors and builds the tables SCSYMTA, SCLDSC, and SCSS, which are then stored as temporary files on disc.

The subroutines called are LUPTA, SCAN, FETCH, CCINT, STORE and PUTST.

Description of Variables:

NSU is the next syntactic unit. It is filled by SCAN. NSU will always be the set of non-blank characters between two adjacent delimeters. Remember that blank is a delimiter.

DELIS is a character string containing all valid delimiters. It is initialized in CCINT.

TYPSWT is a label switch for processing header statement keywords (eg. "INPUTS").

FNCSWT is a label switch for processing function statement keywords (eg. "AND").

FLSHSWT is a switch (label) which returns the program execution to the appropriate place after a card has been flushed.

COMBFLAG is a bit indicating whether or not the circuit is combinational ('1').

N # is the number of leads.

I # is the number of inputs.

Ø # is the number of outputs.

F # is the number of feed backs.

E # is the number of lead names.

K # is the pointer to the next available location in the user supplied successor's list (SCSS.SS). K # -1 indicates the number of entries in SCSS.SS.

P contains the index of the delimiter in DELIS. P is the value returned by SCAN.

SCSYMTA is a scratch file which will contain the contents of the future structure $SYMTA in stream file form. (i.e. we can access only one member of the file at a time. This saves space).

SCSS is a scratch file which will contain the contents of the future structure $SUCS in stream file form. Note that SCSS contains two extra entries FLAG and LNAME. FLAG is employed to indicate the end of a list of output references for a lead. LNAME holds the name of the lead for which ØNAME is the output reference lead. The output reference pointer of SCLDSC is not used and so is not necessary.

SCLDSC is a scratch file which will contain the contents of the future structure $LDSC in stream file form.

Control Card Scan:

CCINT is called to scan for a control card. CTRLERR is

the label transferred to if any more illegal cards are found. OPR is the variable, defined in CCINT, which holds the second syntactic unit of the control card. For a TRAIZE run, this syntactic unit must be 'TRAIZE'.

SCAN_TEXT:

I#, Ø#, F# are set to their default initial values. FLSHSWT is set to SCAN_#S, which indicates we are processing header statements. FETCH is called to get the operator's table, TRAIZE.$OPTAB. SCAN is called to get the next operand delimiter pair.

SCAN_#S:

If the delimiter is a color, we go to SCAN_ATS, since the statement can not be a header statement. The keyword is looked for by LUPTA and a switch made to the appropriate action according to the value returned by LUPTA. If LUPTA returns a zero, the keyword is invalid and a message is printed.

For all subscript values of TYPSWT except 34, 35 and 36, error messages are printed and the remainder of the statement is flushed.

If the subscript value of TYPSWT is 34, we have an 'INPUTS' statement. The delimiter from the last SCAN call should be an opening parenthesis. If it is not, a message is printed, the remainder of the statement is flushed and a transfer is made to SCAN_#S. If it is, SCAN is called to get the stated number of inputs. The delimiter should be a closing parenthesis.

If not, the action is similar to that for the missing open parenthesis. If it is, SCAN is called to pick up an input name. Input names are picked up as long as the delimiter returned is a comma and the number of input labels is less than the stated number. If the delimiter is not a comma, but is a semicolon, the last input name is picked up; otherwise, an error message is printed and the remainder of the statement flushed. The stated number of inputs is compared with the actual number. If the stated number is larger than the actual number, a message is printed and the stated number corrected. TYPSWT (35) and TYPSWT (36) handle output and feedback header statements respectively. The action for each is similar to that above.

Here is a trace of the Header Statement Scan for card 2 of CIRCUIT2 description.

Present Label	Present Action
	K#= 1
	I# = 0
SCAN _ #S	SCAN looks at columns 1 to 20 of CARD2
	P ← 8 and NSU ← 'INPUTS'
	Look up 'INPUTS' position in $OPTAB using LUPTA
	position is 34
	GO TO TYPSWT (34)
TYPSWT (34)	P = 8
	SCAN looks at columns 21 to 22 of CARD2
	P ← 9 and NSU ← '2'
	I# ← 2 and D (1) ← 2
	J ← 1
	SCAN looks at columns 23 to 27 of CARD2
	P ← 3 and NSU ← 'X1'
	INPUT-LIST (1) ← 'X1'
	J ← 2
	SCAN looks at columns 28 to 30 of CARD2
	P ← 6 and NSU ← 'X2'
	P ≠ 3 and J ≮ I#
	P = 6

Present Label	Present Action
	INPUT - LIST ← 'X2'
	J = I#
	SCAN looks at rest of CARD2 and columns
	1 to 21 of CARDS
	GO TO SCAN _ #S

SCAN_ATS:

E# and N# are initialized, and the value in FLSHSWT changed. A check is made to see if all the header statements have been read. If not, a message is printed. If there is no feedback statement, a message is printed indicating that the circuit is combinational. The scratch files are opened for output.

SCANATLOOP:

The leads counter, N# , is bumped by one. IR is initialized. If the delimiter is a colon and NSU is nonblank, SCAN is called to pick up the lead name and the label counter, E#, is bumped. There may be more than one name per lead. The lead name and the associated line number are written onto the SCSYMTA file. If the label counter has not been bumped at least once since the beginning of this statement, a message is printed. NSU should contain the function keyword. LUPTA is called to identify the keyword by returning its associated lead function number.

Action is dependent on the value returned by LUPTA. For all subscript values of FNCSWT except one through fourteen, error messages are printed and the remainder of the statement is flushed.

If the subscript of FNCSWT is one, we have an input lead. BLDSC.FUNC is filled and a transfer taken to SCAN_ORS.

If the subscript of FNCSWT is two, we have a feedback lead. BLDSC.FUNC is assigned and since a feedback lead should have one input reference, a check is made to see that it exists

and is followed by a semicolon. If there are output references, go to SCAN_ORS. If all of these conditions fail, a message is printed and the remainder of the statement is flushed.

For the remainder of the valid functions, BLDSC.FUNC is filled and a transfer made to SCAN_IRS.

Note for macro leads, which have not been implemented yet but will be in the future, all the subcomponents after the first will have to be given a negative value for BLDSC.FUNC. This identifies them with each other. Also, they must be described sequentially by the user. In other words, the user describes them as a unit and TRAIZE program keeps them together as a unit, thus, the negative function values are assigned.

SCAN_IRS:

If the delimiter, from the last SCAN call is not a blank, there are no input references. An error message is printed, the statement is flushed, and a transfer is made to SCANATLOOP. Otherwise, SCAN is called, and, while the delimiter is a comma, the input reference names are picked up. The number of input references is checked to see if it exceeds five. If it does, a message is printed and the remainder of the statement is flushed. Transfer to SCANATLOOP. If it does not, and the delimiter is a semicolon END_STAT is executed. If it does not, and the delimiter is a slash, a transfer is made to SCAN_ORS. If there is no slash or semicolon, a message is printed, the remainder of the statement is flushed, and a transfer is made to SCANATLOOP.

END_STAT:

The input reference names, and the lead function number, are written on to the SCLDSC file. If NSU from the last SCAN call is 'END', a transfer is made to the end of PARSER, END_PARSER; otherwise the transfer is to SCANATLOOP.

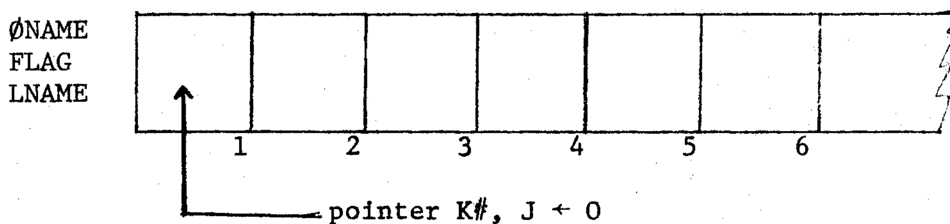
Here is a trace of the Lead Statement Scan and Input references using CARD 7 of CIRCUIT 2. Assume this is the first lead CARD.

Present Label	Present Action
SCAN_ATS	E#, N# ← 0, P=2, NSU = 'Ø1'
	Since I# ≠ 0 and Ø# ≠ 0 and F# = 0
	'combinational circuit'
	Open files SCYMTA, SCLDSC, SCSS
	ØLDE# ← E# :. ØLDE# = 0
SCANATLOOP	N# ← 1
	Initialize BLDSC.IR ← '000000'
	E# ← 1
	BSYMTA.NAME ← 'Ø1'
	BSYMTA.LINM ← '1'
	WRITE (3) and (4) onto SCSYMTA file
	SCAN looks at columns 5 to 8 of CARD5
	P ← 2 and NSU ← 'ØUT'
	E# ← 2
	BSYMTA.NAME ← 'ØUT'
	BSYMTA.LINM ← 1
	WRITE (5) and (6) onto SCSYMTA file
	SCAN looks at columns 9 to 17 of CARD5
	P ← 0 and NSU ← 'OR'
	ØLDE# = 0 and E# = 2
	∴ Look up 'ØR' using LUPTA in $OPTAB
	GO TO FNCSWT (14)

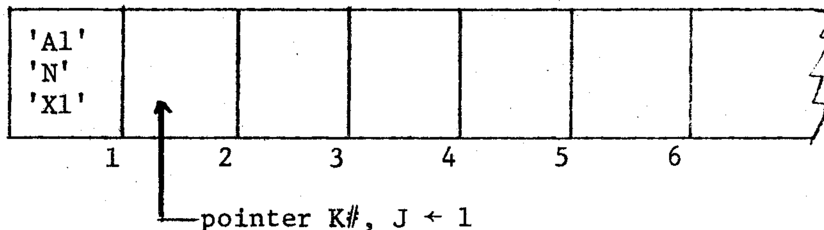
Present Label	Present Action
SCAN_IRS	P = 0
	J ← 1
	SCAN looks at columns 18 to 19 of CARD5
	P ← 3 and NSU ← 'A1'
	BLDSC.IR (1) ← 'A1'
	J ← 2
	SCAN looks at columns 20 to 22 of CARD5
	P ← 6 and NSU ← 'X2'
	J ≯ 5
	BLDSC.IR (2) ← 'X2'
	WRITE (1), (2) (part), 7, 8, 9, onto BCLDSC file
	CALL SCAN to get first NSU of next card.

The structure of SCSS.SS is such that we can refer, independently of the output reference pointer in SCLDSC (BLDSC.OUTPTR*), to the lead from which the output leads come. For this reason LNAME was added to the structure. LNAME is the lead name of the lead which has 'ØNAME' as an output reference.

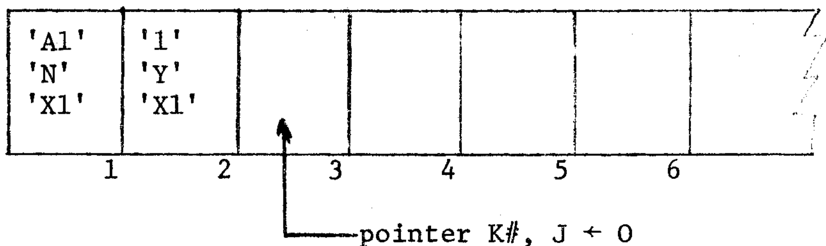
Imagine SCSS as a sequential file where K# points to the first available slot. For CIRCUIT2 the following pictures show how SCSS.SS is filled.



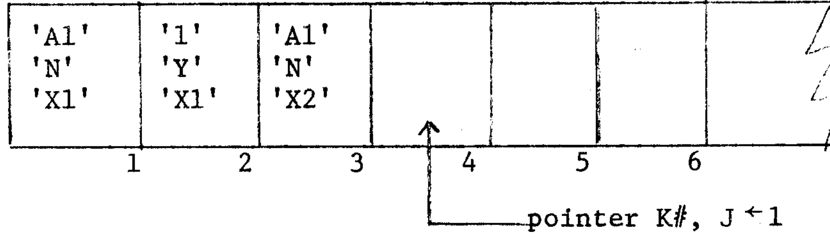
After the first output reference;

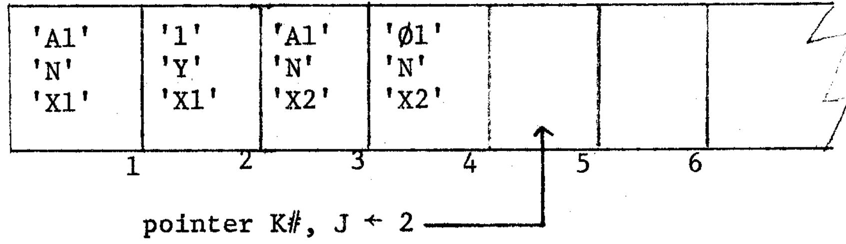


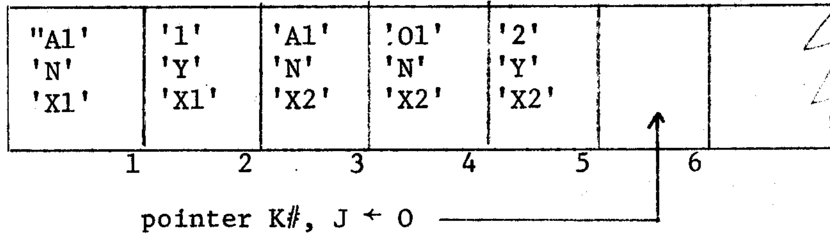
After the second output reference;

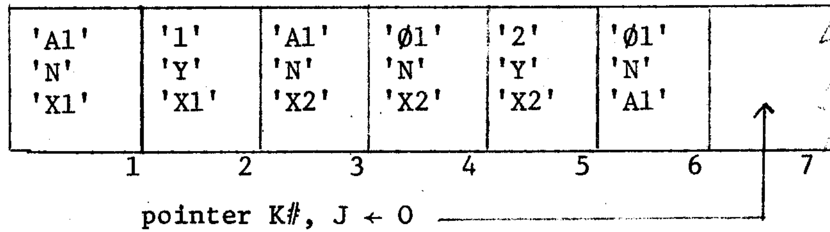


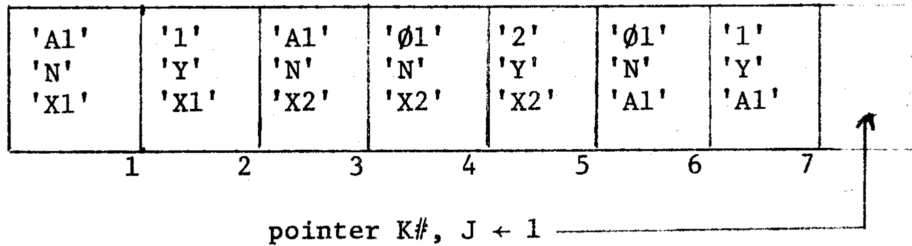
* This substructure should be deleted from your system if it is present..... and so on;











Now, SCSS is filled and K# points to the next available slot.

SCAN_ORS:

If the delimiter is a semi-colon, go to END_STAT: otherwise, SCAN is called. If NSU is blank, a message is printed, the remainder of the statement is flushed and a transfer is made to SCANATLOOP. Otherwise, SCAN is called while the delimiter is a comma, and, the output reference names are picked up. With the output name is an associated flag saying that it is not a list tailer. The two of these are written onto the SCSS file, and K# is bumped by one to point to the next available location of SCSS.

If the delimiter is not a semicolon nor a comma, a message is printed, the statement is flushed and action returns to SCANATLOOP. If the delimiter is a semicolon, the last output reference is written out with the associated flag indicating that it is not a list tailer. The list tailer is now written and contains the number of output references for the lead. Go to END_STAT.

Here is a trace of the Lead Statement Scan and Output Reference Scan using CARD5 of CIRCUIT2. Assume this card is the first Lead Statement card.

Present Label	Present Action
SCAN_ATS	E#, N# ← 0, P = 2, NSU = 'X2'
	Since I# ≠ 0 and o# ≠ 0 and F# = 0
	'Combinational Circuit'
	Open files SCYMTA, SCLDSC, and SCSS
	OLDE# ← E# ∴ OLDE # = 0
SCANATLOOP	N# ← 1
	Initialize { BLDSC. IR ← '000000'
	P = 2 and NSU = 'X2'
	E # ← 1
	BSYMTA.NAME ← 'X2'
	BSYMTA.LINM ← '1'
	WRITE (3) and (4) onto SCSYMTA file
	SCAN looks at columns 5 to 21 of CARD5
	P ← 0 and NSU ← 'INP'
	OLDE# = 0 and E# = 1
	∴ Look up 'INP' using LUPTA in $OPTAB
	GO TO FNSCWT (1)
	BLDSC.FUNC ← 1
	GO TO SCAN_ORS
SCAN_ORS	P = 0
	SCAN looks at columns 22 to 23 of CARD5

Present Label	Present Action
	P ← 3 and NSU ← 'A1'
	J ← 1
	P = 3
	BSS.ØNAME ← 'A1'
	BSS.FLAG ←'N'
	BSS.LNAME ← 'X2'
	Write (6) and (7) onto SCSS file
	K# ← 2 and J ← 2
	SCAN looks at colums 24 to 26 of CARD5
	P ← 6 and NSU ← 'Ø1'
	BSS.ØNAME ← 'Ø1'
	BSS.FLAG ← 'N'
	BSS.LNAME ← 'X2'
	WRITE (8) and (9) onto SCSS file
	K# ← 3
	K# ← 4
	JCHARS ← 2
	BSS.ONAME ← '2'
	BSS.FLAG ← 'Y'
	WRITE (11) and (12) onto SCSS file

Present Picture of SCSS file

	1	2	3	4	5
FLAG	'N'	'N'	'Y'		
ØNAME	'A1'	'Ø1'	'2'		
LNAME	'X2'	'X2'	'X2'		

K# → 4

↑ SCLDSC.OUTPTR (between 2 and 3)

Present Label	Present Action
	GO TO END_STAT
END_STAT	WRITE (1), (2), (5), (10) onto SCLDSC file
	SCAN looks at first NSU of next card.

END_PARSER:

The scratch files are closed. This is done so these files can be opened in BUILDER for input purposes. The necessary parameters of the circuit (eg. I#, E#,) are stored in a scratch file called STRMIN.

SECTION 3) Program BUILDER

This procedure accepts the scratch files filled by PARSER and creates, for the circuit, $CIDS, $LDSC, $SYMTA, $REFS and $SUCS. BUILDER detects logic errors in the circuit description. To minimize core requirements, the structures are freed as they are completed.

The subroutines called are STORE, FETCH, PUTST, SORT, LUPTA, LOUP, CRESSA, FCID, [illegible]EDR, LDIR, SDIR and PCID.

Description of Variables:

The variables not described here are described at the beginning of SECTION 2).

N holds the output reference pointer for a lead (obsolete)

P is the return value of subroutine STORE.

KYWRD holds the keyword to be looked for by LOUP.

The STRMIN file is read of its contents, which are then printed along with headings. FETCH is called to get TRAIZE.$OPTAB.

The scratch files are opened for input. K# is decremented by one so that it indicates how many entries are in the user supplied successor's list. The dimensions vector is filled for $SYMTA and $SYMTA is allocated. Its two parameters are filled.

READSYM:

The symbol names and their corresponding line numbers are obtained from SCSYMTA. Since a lead may have more than one name (i.e. there may be more than one label per function statement), a slight ambiguity arises when referring to the label of a given lead function or line number. In fact, if we wish to obtain a label name for a line

given by $SYMTA.BLINK , we may get anyone of the label names for that line (if there is more than one).

For example, suppose the third lead statement is:

X1: @ 24: INP;

Also assume X1 and @ 24 are the 4th and 5th entries into $SYMTA. We would have:

$SYMTA.BLINK (3) = 4

$SYMTA.BLINK (3) = 5

Here is a trace for lead statements 4, 5, 6 and 7 of CIRCUIT2.

Picture of SCSYMTA:

'X1' 1	'X2' 2	'A1' 3	'Ø1' 4	'OUT' 4	
1	2	3	4	5	

Present Label	Present Action
READSYM:	I ← 1; Read (SCSYMTA) into (BSYMTA)
	$SYMTA:NAME (1) ← BSYMTA.NAME (1) ← X1
	$SYMTA.LNM (1) ← BSYMTA.LINM (1) ← 1
READSYM:	I ← 2; Read (SCSYMTA) into (BSYMTA)
	$SYMTA.NAME (2) ← BSYMTA.NAME (2) ← 'X2'
	$SYMTA.LNM (2) ← BSYMTA.LINM (2) ← 2
READSYM:	I ← 3; Read (SCSYMTA) into (BSYMTA)
	$SYMTA.NAME (3) ← BSYMTA.NAME (3) ← 'A1'
	$SYMTA.LNM (3) ← BSYMTA.LINM (3) ← 3

Present Label	Present Action
READSYM:	I ← 4; Read (SCSYMTA) into (BSYMTA)
	$SYMTA.NAME (4) ← BSYMTA.NAME (4) ← 'Ø1'
	$SYMTA.LNM (4) ← BSYMTA.LINM (4) ← 4
READSYM:	I ← 5: Read (SCSYMTA) into (BSYMTA)
	$SYMTA.NAME (5) ← BSYMTA.NAME (5) ← 'ØUT'
	$SYMTA.LNM (5) ← BSYMTA.LINM (5) ← 4
	SORT THE LEADS
	Form the back links.
	I ← 1, $SYMTA.BLINK (1) = 1
	I ← 2, $SYMTA.BLINK (2) = 2
	I ← 3, $SYMTA.BLINK (3) = 3
	I ← 4, $SYMTA.BLINK (4) = 4
	I ← 5, $SYMTA.BLINK (4) = 5

The next structure filled is $LDSC. The dimensions vector is filled and $LDSC allocated. The parameters are filled in.

READLDSC:

SCLDSC is read into BLDSC for each lead. The addresses or line numbers, of the non zero input references are found by a look up, using LOUP, into $SYMTA. These numbers are placed in $LDSC.IR. A message is printed if a label is not found. The function number of the lead is stored in $LDSC.FNC.

Here is a picture of SCLDSC for statements 4, 5, 6, 7 of CIRCUIT2.

FNC	1	1	13	14	
IR(1)	'000000'	'000000'	'X1'	'A1'	
IR(2)	'000000'	'000000'	'X2'	'X2'	
IR(3)	'000000'	'000000'	'000000'	'000000'	
IR(4)	'000000'	'000000'	'000000'	'000000'	
IR(5)	'000000'	'000000'	'000000'	'000000'	

Here is a trace for statement number 6 of the CIRCUIT2 description.

Present Label	Present Action
READLDSC:	I ← 3; Read (SCLDSC) into (BLDSC)
	KYWD ← BLDSC.IR (1) ← 'A1'
	LOUP returns 3
	$LDSC.IR (3, 1) ← 3
	KYWD ← BLDSC.IR (2) ← 'X2'
	LOUP RETURNS 2
	$LDSC.IR (3, 2) ← 2
	$LDSC.FNC (3) ← 13

BUILD_$REFS:

The dimensions vector is filled in order to allocate $REFS. $REFS, INPUT_LIST, OUTPUT_LIST and FEEDBACK_LIST are allocated. The last three structures are filled from the STRMIN file. Of course, if F# is not greater than zero FEEDBACK_LIST remains empty.

INPUT_LIST is read to get each of the inputs. A check is made to see if the input is in the $SYMTA table and that it has the proper function (i.e. 'INP').

The input lead line number is entered into $REFS.IL:

The same procedure is carried out for OUTPUT_LIST (and FEEDBACK_LIST if necessary).

The parameters of $REFS are filled and the structures INPUT_LIST, OUTPUT_LIST and FEEDBACK_LIST are freed.

LEOR is called to level and organize the circuit. The back links must be recalculated since the old line numbers may not coincide with the new ones.

CRESSA is called to create the successors list ($SUCS) from the input references. CRESSA returns the number of elements in $SUCS.SS including the pointers.

The user supplied successors are checked against the created ones in $SUCS. Remember that the user supplied successors list has list tailers, not list headers. Also the order of the lead numbers might have changed after the call to LEOR, therefore the output references may not appear in the same order.

CHECK_SS:

Each member of SCSS is examined. If it contains the first output reference name, the lead name for which it is the output reference is looked for in $SYMTA. We now access the created successors of $SUCS for this lead through $LDSC.ØR.

The output reference name is looked for in $SYMTA. Finally a check is done to see if its lead number appears in $SUCS.SS. If it is not found, a message is prin t ed.

For CIRCUIT2 SCSS.SS would be:

	1	2	3	4	5	6	7
ONAME	'A1'	'1'	'A1'	'Ø1'	'2'	'Ø1'	'1'
FLAG	'N'	'Y'	'N'	'N'	'Y'	'N'	'Y'
LNAME	'X1'	'X1'	'X2'	'X2'	'X2'	'A1'	'A1'

K# ← 7

$SUCS.SS would be:

1	2	3	4	5	6	7
1	3	2	3	4	1	4

$LDSC.ØR would be: $LDSC.ØR (1) = 1

$LDSC.ØR (2) = 3

$LDSC.ØR (3) = 6

$LDSC.ØR (4) = 0

Present Label	Present Action.
CHECK_SS:	K ← 0; K# ← 7; I ← 1
	BSS.FLAG = 'N'
	KYWRD = 'X1', K = 1
	N ← 1
	KYWRD = 'A1', J = 3
	L ← 2
	$SUCS.SS (2) = 3 = J
CHECK_SS:	I ← 2
	BSS.FLAG = 'Y'; K=0
CHECK_SS:	I ← 3
	BSS.FLAG = 'N'
	KYWD ← 'X2'; K←2
	N ← 3
	KYWD ← 'A1'
	J ← 3
	L ← 4
	$SUCS.SS (4) = 3 = J
CHECK_SS:	I ← 4
	BSS.FLAG = 'N'
	KYWD = 'Ø1'; J = 4
	L ← 4

Present Label	Present Action
	$SUCS.SS (4) = 3 ≠ J
	L ← 5
	$SUCS.SS (5) = 4 = J
CHECK_SS:	I ← 5
	BSS.FLAG = 'Y'; K = 0
CHECK_SS:	I ← 6
	BSS.FLAG = 'N'
	KYWD ← 'A1'; K ← 3
	N ← 6
	KYWD ← 'Ø1'; J = 4
	L ← 7
	$SUCS.SS (7) = 4 = J
CHECK_SS:	I ← 7
	BSS.FLAG = 'Y'; K=0

BUILD_$CIDS:

$CIDS is allocated and filled in. The five structures are stored in the file if the DISP value is appropriate.

END_BUILDER:

The scratch files are closed. If some structures have been stored and the file directory is in good condition, LDIR is called to list the directory and SDIR to save the updated directory. The directory is fetched using FCID and printed out using PCID to give the user picture of the circuit description TRAIZE has created.

SECTION 4)

SUBROUTINE CCINT

This subroutine interprets control cards for FAUST and TRAIZE.

The subroutines called are ALLF, SCAN, FETCH, FCID, FLEV2 and PCID.

Description of Variables:

NSU is the next syntactic unit

DISP is the disposition of the following description or simulation desired by the user with respect to the file.

LIST indicates whether or not the circuit description is to be listed.

OEOF, OEOD hold the previous values of EOF, EOD.

CARDIN and JOBPARM are allocated if necessary, and DEL, QUT, QUP, SOUP, NC, EOF, EOD, SOCAR, LINM, LEN are initialized.

The present EOF and EOD labels are saved; they are re-initialized.

If SOCAR (1) is '%' we have a control card already, so SOUP is set to one and SOCAR (1) set to blank. Go to CONTROL.

NOTYET:

If we do not have a '%' sign, SCAN is called and the remainder of the card is flushed. GO TO NOTYET.

CONTROL: Variables are initialized to default values. The first NSU picked up by SCAN, is the RUNAME. SCAN is called to pick it up. The delimiter must be a colon.

The operator is picked up by SCAN indicating a circuit description (TRAIZE) or a simulation run (SIMULATE) follows.

The delimiter must be a blank.

The keywords CIRCUIT, FAULTS, LIST, and DISP are looked for while there is no semicolon as delimiter. For each of these keywords, the delimiter following must be an equal sign. If it is, SCAN is called to pick up the CNAME, FLOP, LIST and DISP, respectively. For each, a check is made to see that the value is not blank.

Error messages are printed if the keyword found is not one of the above; If the delimiter following is not an equal sign or if the operand following the keyword is a blank.

EOFILE:

If end of file is reached before a control card is found, a message is printed.

FAIL:

The return value is set to negative one, a message is printed with some data. The rest of the statement is flushed.

BYEBYE:

EOD and EOF regain their original labels and the value in CCC is returned.

SECTION 5) SUBROUTINE CRESSA

This subroutine creates the successors structure $SUCS from the input references.

Description of Variables:

CNAME is the circuit name.

N# is the number of leads in the circuit.

S# is the number of successors including pointers.

IRI is the input reference lead number.

NOS is the counter of the number of input references for each lead.

IS is the counter for the number of successors.

NSLOOP:

This loop counts the number of successors for each lead and the total number of successors, S#. S# is incremented by one each time a new lead is found, that has successors. The reason for this is that each lead with successors must have an additional entry, to indicate the number of successors.

Thus for a lead with K successors, K + 1 locations in the successors list are saved.

The structure is now allocated and the parameters filled in.

ISLØØP:

The output reference pointers from $LDSC are filled from NØS. This is accomplished by looking at each lead of the circuit. If it has any output references, the output reference pointer for that lead is set to point to the next location (IS + 1).

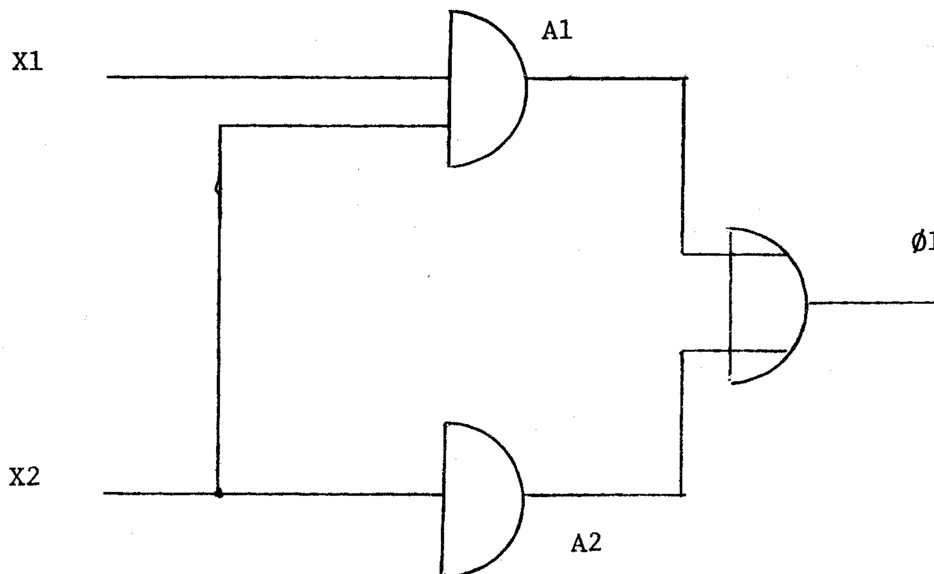
The value of the location pointed to is set to the number of successors of that lead. IS is incremented by the number of output references for the same lead. This puts IS in position for the next lead and its output references. NØS is set to point the same as $LDSC.OR.

SSLOOP:

This loop fills in the successors. Each lead is examined for input references. One by one the input reference numbers are placed in IRI. NOS for that lead is incremented by one, to point at the next available successors slot for that lead and the lead is placed in the successors list.

Here is an illustration of the program action by tracing.

CIRCUIT DIAGRAM.



Let the line numbers associated with the leads be:

X1 ↔ 1

X2 ↔ 2

A1 ↔ 3

A2 ↔ 4

Ø1 ↔ 5

Let the input references be given as follows:

For X1 $LDSC.IR (1, J) = 0 for J = 1 to 5

For X2 $LDSC.IR (2, J) = 0 for J = 1 to 5

For A1 $LDSC.IR (3, 1) = 1,

$LDSC.IR (3, 2) = 2,

$LDSC.IR (3, J) = 0 for J = 3 to 5

For A2 $LDSC.IR (4, 1) = 2

$LDSC.IR (4, J) = 0 for J = 2 to 5

For Ø1 $LDSC.IR (5, 1) = 3

$LDSC.IR (5, 2) = 4

$LDSC.IR (5, J) = 0 for J = 3 to 5

From loop NSLOOP, we calculate.

NOS (1) = 1 Also S# = 9

NOS (2) = 2

NOS (3) = 1

NOS (4) = 1

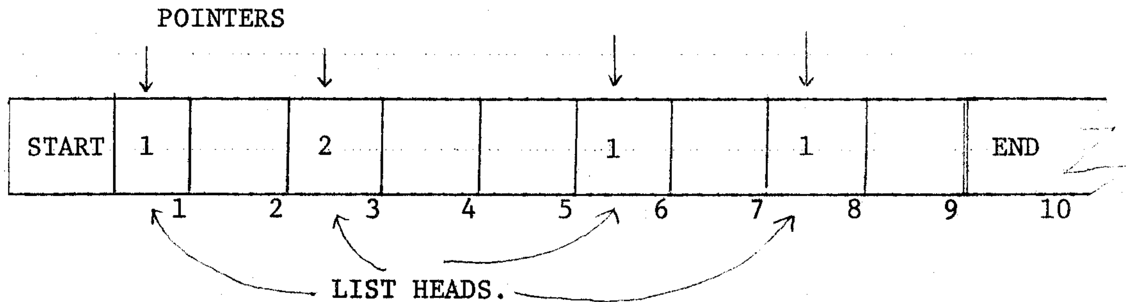
NOS (5) = 0

From loop ISLOOP, we calculate

POINTERS	LIST HEADS
$LDSC.OR (1) = 1	$SUCS.SS (1) = 1
$LDSC.OR (2) = 3	$SUCS.SS (3) = 2

POINTERS	LIST HEADS
$LDSC.OR (3) = 6	$SUCS.SS (6) = 1
$LDSC.OR (4) = 8	$SUCS.SS (8) = 1

So far we have created the imaginary file.



From loop SSLOOP we calculate

$SUCS.SS (2) = 3,

$SUCS.SS (4) = 3,

$SUCS.SS (5) = 4,

$SUCS.SS (7) = 5,

$SUCS.SS (9) = 5.

(Remember new NOS vector is changed in ISLOOP)

POINTERS

NOS (1) = 1

NOS (2) = 3

NOS (3) = 6

NOS (4) = 8

This subroutine organizes the circuit leads into levels, if possible.

Description of Variables:

1 TLED
  2 TFNC
  2 TLVL
  2 TIR (5)
  2 TOR is a temporary structure to hold an element of the $LDSC table while elements are being interchanged.

N# is the number of leads.

I# is the number inputs

Ø# is the number outputs

F# is the number of feedbacks

LV is a pointer to the current level.

K is an indicator of the number of leads that have been levelled.

MAP, PAM indicate the new line numbers assigned to the leads. MAP is a mapping from the old line numbers to the new and PAM is the inverse mapping.

OK is the old value of K.

Allocation of certain structures is checked and parameters filled in.

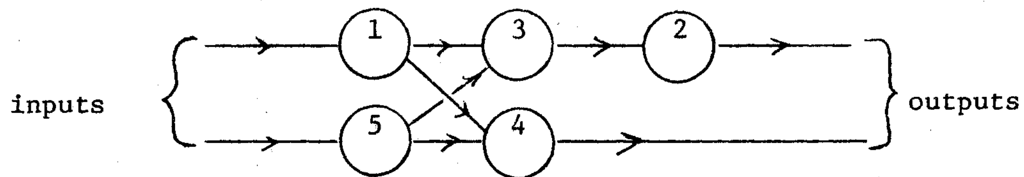
ORG:

The initial level given to all leads is 999. MAP is and PAM are set to the identity mapping, since no leads have been interchanged.

The levels for the feed back nodes are set to zero. This makes it possible to level the remainder of the circuit.

LLOOP:

This loop attempts to level and organize the circuit. Suppose we have the following circuit with the lead numbers as shown.



The mapping that should result is 1 →1, 2→5, 3 → 3, 4 → 4, 5 → 2.

Here is a trace of LLOOP for the above circuit.

Present Label	Present Action
ØRG:	LVL (1) = LVL (2) = LVL (3) = LVL (4)= LVL (5) = 999
	ØK, K = 0, LV = 0
LLOOP:	I = 1
ØLØØP:	L = 1
ILOOP:	IR (1, J) for J = 1 to 5 are zero
	K = 1
	LVL (1) = 0
NYET:	I = 2
ØLØØP:	L = 2
ILOOP:	IR (2, 1) = 1, LVL (1) = 0
NYET:	I = 3

Present Label	Present Action
ØLØØP:	L = 3
ILOOP:	IR (3, 1) = 1, LVL (1) = 0
NYET:	I = 4
ØLØØP:	L = 4
ILOOP:	IR (4, 1) = 1, LVL (1) = 0
NYET:	I = 5
ØLØØP:	L = 5
ILOOP:	IR (1, J) for J = 1 to 5 are zero
	K = 2
	Call Interchange
	ITC (5, 2)
	LVL (2) = 0
NYET:	I = 6
	OK = 2, LV = 1
LLOOP:	I = 3
ØLØØP:	L = 3
ILOOP:	IR (3, 1) = 1, LVL (1) = 0
	IR (3, 2) = 5, LVL (2) = 0
	IR (3, 3) = IR (3, 4) = IR (3, 5) = 0
	K = 3
	LVL (3) = 1
NYET:	I = 4
ØLØØP:	L = 4
ILOOP:	IR (4, 1) = 1, LVL (1) = 0
	IR (4, 2) = 5, LVL (2) = 0
	IR (4, 3) = IR (4, 4) = IR (4, 5) = 0

Present Label	Present Action
	K = 4
	LVL (4) = 1
NYET:	I = 5
ØLØØP:	L = 5
ILOOP:	IR (5, 1) = 1, LVL (1) = 0
	IR (5, 2) = 3, LVL (3) = 1
NYET:	I = 6
	OK = 4, LV = 2
LLOOP:	I = 5
ØLØØP:	L = 5
ILOOP:	IR (5, 1) = 1, LVL (1) = 0
	IR (5, 2) = 3, LVL (3) = 1
	IR (5, 3) = IR (5, 4) = IR (5, 5) = 0
	K = 5
	LVL (5) = 2
NYET:	I = 6
	OK = 5, LV = 3

MAP-IR-LOOP:

Here, the input reference numbers are mapped into the new numbers.

In LLOOP , different action is taken if the function lead number is negative. This is due to the fact that macro components must appear together in the same order. The negative function number indicates a macro component, thus it must be changed so that it immediately follows the head component in the new mapping.

FBKF:

This section finds maximal connected subgraphs.

Description of Variables:

VIS keeps track of the unlevelled leads we have seen, when tracing out a maximal connected subgraph path.

MMB marks the numbers of the maximal connected subgraph with a '1'.

S, P are pointers to leads which are unlevelled.

IN is a pointer to an input reference lead number.

MLK is a set of links in the linked list of the maximal connected subgraphs. Each member points to another lead in the list.

MC holds a pointer to the linked lists for each maximal connected subgraph.

MMB, MC, MLK, M are initialized. A list of the leads which could not be levelled is printed.

For each lead of the unlevelled portion of the circuit, VIS is initialized to zero and MCSG called if the current lead is not a member of a maximal connected subgraph already.

If MCSG returns a '1', the number of maximal connected subgraphs is incremented by one.

MSCG:

This routine creates a linked list for each maximal connected subgraph.

VIS is set to one for lead p. We travel all input reference links of p. For each of the input references, if VIS (IN) '0', we have not seen this lead on this path yet, and if MMB (IN) is '0', it does not belong to a maximal connected subgraph.

INLOOP:

If IN equals S (the original lead MCSG was called on), we have made a loop, so MMB for the present lead is set to '1'.

If VIS (IN) and MMB (IN) are '1', an input reference to a lead has been seen on this path and in a maximal connected subgraph, therefore MMB of the present lead is set to '1'.

If VIS (IN) and MMB (IN) are '0', an input reference has not been seen yet on this path, and is not in a maximal connected subgraph, therefore, we must check it out and MCSG is called on the input lead.

If MMB is '0' after INLOOP has been finished, '0' is returned. Otherwise, if we have not printed any member of this maximal connected subgraph MC (M) will be zero, so a heading is printed. The current value in MC is put into the linked list MLK at position p, where p is the new lead to be added to the list, and the head pointer MC changed to point to p. The lead is printed.

MAP-IR-LOOP:

This loop maps the input references in to their new lead numbers using MAP. The line numbers associated with the structure $SYMTA are also mapped.

The input, output and feedback lists are mapped into the new lead numbers.

ITC:

This routine interchanges two members of the $LDSC structure using the structure TLED as a temporary storage when switching MAP and PAM are also up dated.

SECTION 7) SUBROUTINE NECAR

This subroutine reads data cards one at a time, keeps a pointer to the current column of the card and returns the character in the column pointed to by the pointer.

The pointer, SOUP, is incremented by one each time NECAR is called. If SOUP is greater than 72, a new card is read. Thus, the user can use columns 1 to 72 for input information; however, the last 8 columns are ignored. If the character in column one is '%', that column is set to blank and a transfer made to process the control card.

Otherwise, the character in the column, pointed to by SOUP, is returned to SCAN or REBL.

SECTION 8) SUBROUTINE REBL

This subroutine reads through blank columns and comments, if contained in the proper quotes.

The subroutine called is NECAR.

BLANK:

NECAR is called repeatedly while the returned value is blank.

If the value returned is a beginning quote of a comment, NECAR is called until an ending quote is found, otherwise the index of the character in the delimiter stack is returned. After a comment, NECAR is called and a transfer made to BLANK.

SECTION 9) SUBROUTINE SCAN

SCAN scans the source input stream and assigns to its only parameter the next syntactic unit found. The parameter is CHAR (*) VAR. The value returned by SCAN is the index of the delimiter in the delimiters string (DELIS) of the source description. If there is no delimiter after the syntactic unit (i.e. the delimiter is a blank), SCAN returns a zero. The included structure is SD and the external subroutines are REBL and NECAR.

Each time SCAN is summoned, set NSU to blank and length of NSU to zero. Call REBL to read through blanks and comments. If REBL returns a non zero number, it has found a delimiter; therefore, return from SCAN without filling NSU. If REBL returns a zero, it has found a character, so concatenate this character and all following characters until a blank is found or a character which is a delimiter. If a delimiter has been found and is a quote, read through comment until a non blank character is reached and return the index of the character in DELIS. If a delimiter has been found and is not a quote, return index of the delimiter in DELIS. If a blank has been found, read through following blanks and comments until non blank character is found and return its index in DELIS.