LRN

Programming my TI

TI-58 / TI-58C / TI-59

Pierre Houbert

Introduction

The programmable calculators Texas Instrument **TI58** and **TI59** appeared in 1977, followed in 1979 by the **TI58C**.

Based on an AOS system (direct algebraic notation), they were programmable with a specific language named, in French, LMS (for specialized machine language).

Some users more saw in these machines their side “scientific calculator” (or mathematical) because of their numerous mathematical and statistical functions, the others adopted these calculators as "pocket computers" and even invented, in these years of rising micro-computing, the term of "pico-computing".

We shall approach here the side programmable calculator and would try to discover this language, at first sight rudimentary and simplistic, which was nevertheless able to fascinate a lot of followers.

Indeed, this language turned out to be really attractive because enough complete for elaborating complex programs. The field of the possible applications even allowed a professional use in a lot of domains.

The modules of marketed programs concerned the mathematics, the navigation, the electric engineering, the agriculture, the financial investment, the stock management and many other activities without forgetting the games.

The only limits were due to the physical constraints of these machines: no alphanumeric display (but paper printing for texts), little memory, no possibility to save programs or data (magnetic cards only for the TI59).

Then why to be interested, today, in the era of "smart phones" and other "tablets", on these ancestral machines and this language of formerly ?
For the same reason which makes that in the age of space shuttles, high speed trains, and other fast machines, our children, and our grandchildren continue to want to learn to make of the “velocipede” : for the pleasure !

Today some emulators of TI exist on diverse operating systems (MS DOS, Windows, Android, Pocket PC) and allow to find this particular pleasure to program with a such language.

First program

First steps

To start with, let us observe the keyboard of our calculator.

The first key which we shall approach is the key 2nd.

It is going to allow us to reach the "second" function of a key, so to obtain Π (Pi) we have to use the second function of the key 3.

Such,

The sequence of keys 2nd Π will give **3.14159265359**

To calculate the circumference of a circle of 4 inches of radius, it is necessary to make 4 x 2 x Π = and thus type :

4 × 2 × 2nd Π =

We can make a first program allowing to calculate the circumference of a circle for any value of the radius...

This program will be like this kind :

- input number
- Multiply by 2
- Multiply by Pi
- display result

Number input will be made with keyboard, then it will be necessary to launch the execution of the program which will stop, displaying the result.

To launch the program (and stop) it, we shall use the key (and the instruction) R/S which means *Run / Stop*.

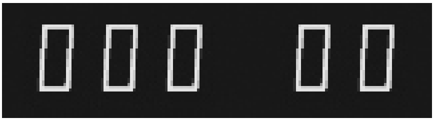
Our program will thus be like that :

× 2 × 2nd Π = R/S

Enter the program

To enter a program, we have to choose the "program mode" using the key LRN (*Learn*).

When we press this key LRN, the display changes and shows two sets of numbers separated by a space.



The first group, consisting of three digits represents the instruction address (we will say "step" of program), and the second group, composed of two digits, is the instruction code.
Each instruction represented on the keyboard has a two-digit code (00 through 99) and our program

× 2 × 2nd π = R/S

could be written

65 02 65 89 95 91

since the respective codes are

- **65** for ×
- **02** for 2
- **89** for 2nd π
- **95** for =
- **91** for R/S

To switch back to "calculator mode", and for exit from "program mode", we press LRN.

Before introducing our program, we will ensure that no other program is in memory by erasing program memory with **CP** obtained pressing 2nd CP (*Clear Program*).

If we go back to programming mode by pressing LRN, we are on step 000 with 00 as instruction code.

When we press ×, the step 001 is shown with 00 as instruction code.

We press 2, then ×, then 2nd π, then =, then R/S.

According to our program entry, we can see the steps of program incremented for a positioning on the step of the following instruction to be entered.

To verify our entry, we have two solutions: we have either to "walk" in our program to display the successive steps, or go out of the programming mode (with LRN) and print our program.

Let us walk…

To verify our entry, we can “go back up” in our program by means of BST.

Every pressing on BST, makes us "go back up" of one step and we see displaying the address of the step and the code of the instruction :

BST display **005 91** then BST **004 95** then BST **003 89** then BST **002 65** then BST **001 02** then BST **000 65**.

We can also "come down" in our program by means of SST.

SST display **001 02**, then SST **002 65**, then SST **003 89** then SST **004 95** then SST **005 91**.

We press on LRN to come back in “calculator mode”.

We left the programming mode while the pointer of step was on the step **005**.

If we press again on LRN to go back, in program mode, it’s step **005** which is displaying..

If we tried to launch the program, nothing would happen because the execution pointer is positioned on the stop command.

To return, in "calculator mode", you must press LRN then RST (*Reset*) to bring the pointer to step **000**.

For checking the program, we are going to print it by using **LST** (2nd List)

On the printer, we obtain :

```
000   65    ×
001   02    2
002   65    ×
003   89    π
004   95    =
005   91  R/S
```

The paper printing gives us the address (step) of the instruction, its code and also its translation.

First test

We can now test our program.

It is necessary to :

- return the pointer to **000** : RST
- enter a radius : for example 2 5
- launch the program : R/S

and we get 157,0796327

The use could be improved by avoiding having to use keys such as RST and R/S.

Indeed, the calculator possesses function keys (**A**, **B**, **C**, **D**, **E**) who could be useful.

We are thus going to use the notion of "label".

To modify our program, we return the pointer to the address **000** with RST, then toggle in programming mode with LRN.

We are on the step **000** before which we are going to insert 2 lines by using **INS** twice : 2nd Ins 2nd Ins

We can now create our label with 2nd Lbl (**LBL**), then A.

We return in "calculator mode" to print : LRN, then RST 2nd List

On the printer, we obtain :

000	76	LBL
001	11	A
002	65	×
003	02	2
004	65	×
005	89	π
006	95	=
007	91	R/S

We can now re-test our program.

It is necessary to :

- enter a radius : for example [2] [5]
- launch the program : [A]

and we get 157,0796327

If after the execution we press on [LRN] to switch in programming mode, we notice that the execution pointer is placed on the step **008**.

We are going to add the second part allowing the calculation of the area of the circle :

[2nd] [Lbl] [B] [x^2] [×] [2nd] [π] [=] [R/S]

or : **LBL B X^2 X Π = R/S**

then [LRN] to go back to “calculator mode”.

[RST] [2nd] [List] for printing.

000	76	LBL
001	11	A
002	65	×
003	02	2
004	65	×
005	89	π
006	95	=
007	91	R/S
008	76	LBL
009	12	B
010	33	X²
011	65	×
012	89	π
013	95	=
014	91	R/S

Now, a number n followed by [A] display the circumference and a number n followed by [B] display the area of the circle.

We can now modify this program to enter the radius only once and make our two calculations in continuation by typing :

radius [A] [B]

To do that, we have to store in memory the radius in the procedure [A] and recall the stored radius in the procedure [B].

Storage in memory

The TI contains several "areas" of storage to keep the used data. These "areas" are called **Registers**.

The first register is the one which corresponds to the digital display : it is the register “**x**”.

The second register is the register of test named “**t**”.

The command [x⇌t] allows, as its name indicates it “x exchange t”, to exchange the values of **x** and **t**.

Example :

1 2 3 [x⇌t] 456	puts the value 123 in **t** and 456 in **x**
[x⇌t]	exchanges : 123 in **x** and 456 in **t**

Other registers are used for the storage, they are numbered from 00 to 99 [1].

To manipulate these registers various instructions are usable :

[STO] nn	copies register **x** in register nn
[RCL] nn	copies register nn in register **x**
[SUM] nn	adds register **x** to register nn
[INV] [SUM] nn	subtracts register **x** from register nn
[2nd] [Prd] nn	multiplies register nn by register **x**
[INV] [2nd] [Prd] nn	divides register nn by register **x**
[2nd] [Exc] nn	exchanges the register nn with register **x**

[1] Differ according to the model of TI and the reserved options - See OP 16 / OP 17

For our program “circle”, we are thus going to store the radius in the register 01 to take it back later.

Behind 2nd Lbl A we will insert STO 0 1.

During the input of the address of the register (01), the display does not move forward for the next step.

Indeed, after STO, 2 characters are expected and take only a single step of program.

Behind 2nd Lbl B we insert RCL 0 1.

We get :

```
000 76 LBL
001 11  A
002 42 STO
003 01 01
004 65  x
005 02  2
006 65  x
007 89 PI
008 95  =
009 91 R/S
010 76 LBL
011 12  B
012 43 RCL
013 01 01
014 33 X²
015 65  x
016 89 PI
017 95  =
018 91 R/S
```

To test, we need to input the radius, to press on A to obtain the circumference then press on B to obtain the area.

If, in “calculator mode” we press on RCL 0 1 , the radius is displayed.

Printing

We are going to use the printer to improve the presentation of the results.

For the use of the printer, we have already seen **LST** (2nd List) who allows to list a program.

We can also use :

INV 2nd List to print the contents of registers (**INV LST**)

2nd Prt to print the register **x** (**PRT**)

2nd Adv to move forward of one line (**ADV**)

Furthermore, some "special features" are usable thanks to the instruction **OP** (2nd Op) :

- **OP 01**, **OP 02**, **OP 03**, **OP 04** et **OP 05** Allow to print an alphanumeric text until 20 characters, a line of printer making twenty characters of wide.
- **OP 06** prints the register **x** followed by 5 alphanumeric characters
- **OP 07** prints a curve with the character "*"
- **OP 08** prints the list of labels used by the program in memory.

The printing of an alphanumeric text is made on a line of 20 characters divided into 4 groups of 5 characters.

- **OP 01** allocates values to the group 1 (outside left)
- **OP 02** allocates values to the group 2 (inside left)
- **OP 03** allocates values to the group 3 (inside right)
- **OP 04** allocates values to the group 4 (outside right)

1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20
OP 01	OP 02	OP 03	OP 04

- **OP 05** prints the alphanumeric line
- **OP 00** erases the contents of the 4 groups (zero)

To allocate values to the groups, the TI uses a cross-reference table of characters :

	0	1	2	3	4	5	6	7
0		0	1	2	3	4	5	6
1	7	8	9	A	B	C	D	E
2	-	F	G	H	I	J	K	L
3	M	N	O	P	Q	R	S	T
4	.	U	V	W	X	Y	Z	+
5	×	*	√	π	e	(	)	,
6	↑	%	↕	/	=	'	×	x̄
7	²	?	÷	!	⊓	△	Π	Σ

So, the character "**A**" is obtained with the code **13**, the character "**=**" with the code **64**…

Thus to print :

RAYON =

We have to write :

OP 00	erases groups
3 5	**R** (character #1)
1 3	**A** (character #2)
4 5	**Y** (character #3)
3 2	**O** (character #4)
3 1	**N** (character #5)
OP 01	allocates to group 1
0 0	space (character #6)
6 4	**=** (character #7)
0 0	space (character #8)
0 0	space (character #9)
0 0	space (character #10)
OP 02	allocates to group 2
OP 05	prints the line

We can also print a text of 5 characters, just behind the number which is in the display (register **x**) using **OP 04** (group 4) and **OP 06**.

To print :

12 cm²

We have to write :

OP 00
1 5
3 0
7 0
0 0
OP 04
1
2
OP 06

Full Program

000 76 LBL	037 42 STO	074 00 00	111 00 00
001 11 A	038 03 03	075 69 OP	112 00 00
002 42 STO	039 71 SBR	076 02 02	113 00 00
003 01 01	040 30 TAN	077 69 OP	114 00 00
004 32 X/T	041 43 RCL	078 05 05	115 00 00
005 01 01	042 03 03	079 92 RTN	116 00 00
006 32 X/T	043 71 SBR	080 76 LBL	117 00 00
007 22 INV	044 28 LOG	081 38 SIN	118 69 OP
008 77 GE	045 71 SBR	082 69 OP	119 03 03
009 96 WRT	046 23 LNX	083 00 00	120 69 OP
010 71 SBR	047 25 CLR	084 03 03	121 05 05
011 23 LNX	048 91 R/S	085 03 03	122 92 RTN
012 71 SBR	049 76 LBL	086 01 01	123 76 LBL
013 39 COS	050 39 COS	087 07 07	124 30 TAN
014 43 RCL	051 69 OP	088 03 03	125 69 OP
015 01 01	052 00 00	089 05 05	126 00 00
016 71 SBR	053 03 03	090 02 02	127 03 03
017 28 LOG	054 05 05	091 04 04	128 06 06
018 65 *	055 01 01	092 03 03	129 04 04
019 02 02	056 03 03	093 00 00	130 01 01
020 65 *	057 04 04	094 69 OP	131 03 03
021 89 PI	058 05 05	095 01 01	132 05 05
022 95 =	059 03 03	096 01 01	133 02 02
023 42 STO	060 02 02	097 07 07	134 01 01
024 02 02	061 03 03	098 03 03	135 01 01
025 71 SBR	062 01 01	099 07 07	136 03 03
026 38 SIN	063 69 OP	100 03 03	137 69 OP
027 43 RCL	064 01 01	101 05 05	138 01 01
028 02 02	065 00 00	102 01 01	139 01 01
029 71 SBR	066 00 00	103 07 07	140 05 05
030 28 LOG	067 06 06	104 00 00	141 01 01
031 43 RCL	068 04 04	105 00 00	142 07 07
032 01 01	069 00 00	106 69 OP	143 00 00
033 33 X2	070 00 00	107 02 02	144 00 00
034 65 *	071 00 00	108 06 06	145 06 06
035 89 PI	072 00 00	109 04 04	146 04 04
036 95 =	073 00 00	110 00 00	147 00 00

148 00 00	179 02 02	210 02 02	241 69 OP
149 69 OP	180 06 06	211 99 PRT	242 02 02
150 02 02	181 04 04	212 22 INV	243 00 00
151 69 OP	182 06 06	213 58 FIX	244 00 00
152 05 05	183 04 04	214 92 RTN	245 03 03
153 92 RTN	184 06 06	215 76 LBL	246 01 01
154 76 LBL	185 04 04	216 96 WRT	247 03 03
155 23 LNX	186 06 06	217 69 OP	248 02 02
156 06 06	187 04 04	218 00 00	249 03 03
157 04 04	188 06 06	219 00 00	250 00 00
158 06 06	189 04 04	220 00 00	251 01 01
159 04 04	190 69 OP	221 03 03	252 04 04
160 06 06	191 03 03	222 06 06	253 69 OP
161 04 04	192 06 06	223 01 01	254 03 03
162 06 06	193 04 04	224 03 03	255 03 03
163 04 04	194 06 06	225 02 02	256 05 05
164 06 06	195 04 04	226 04 04	257 01 01
165 04 04	196 06 06	227 03 03	258 07 07
166 69 OP	197 04 04	228 06 06	259 00 00
167 01 01	198 06 06	229 69 OP	260 00 00
168 06 06	199 04 04	230 01 01	261 07 07
169 04 04	200 06 06	231 02 02	262 03 03
170 06 06	201 04 04	232 04 04	263 00 00
171 04 04	202 69 OP	233 03 03	264 00 00
172 06 06	203 04 04	234 05 05	265 69 OP
173 04 04	204 69 OP	235 00 00	266 04 04
174 06 06	205 05 05	236 00 00	267 69 OP
175 04 04	206 92 RTN	237 04 04	268 05 05
176 06 06	207 76 LBL	238 01 01	269 25 CLR
177 04 04	208 28 LOG	239 03 03	270 35 1/X
178 69 OP	209 58 FIX	240 01 01	271 91 R/S

To use the program :

radius A

The result is printed.

<u>Example :</u>

Input : 1 5 A

Result :

```
====================
RAYON =
               15.00
PERIMETRE =
               94.25
SURFACE =
              706.86
====================
```

In this program, we notice at first that instructions can be used as labels :

LBL COS, **LBL LNX**, **LBL WRT**…

and that labels can be called by **SBR** (SBR) with return after the call thanks to **RTN** (INV SBR), **SBR** and **RTN** meaning respectively ***S****u****BR****outine* et ***R****e****T****ur****N***.

Other observation, we see that the printing of alphanumeric text is expensive in number of program steps :

- "RAYON =" routine **COS**, step 49 to 79 = 31 steps
- "PERIMETRE =" routine **SIN**, step 80 to 122 = 43 steps
- "SURFACE =" routine **TAN**, step 123 to 153 = 31 steps
- "=======…" routine **LNX**, step 154 to 206 =53 steps
- "SAISIR UN NOMBRE ! " routine **WRT** 215 to 271 = 57 steps

Let be a total of 215 steps for a program of 271 steps !

The program contains a test of value allowing to go towards a treatment of error (**LBL WRT**) if the value of the entered radius is lower than 1.

```
| 004 32 X/T |
| 005 01 01  |
| 006 32 X/T |
| 007 22 INV |
| 008 77 GE  |
| 009 96 WRT |
```

that could have been write :

```
| 004 32 X/T |
| 005 00 00  |
| 006 77 GE  |
| 007 96 WRT |
| 008 32 X/T |
```

GE and **INV GE** allow a conditional jump according to a comparison between registers **x** and **t**, **GE** meaning ***G**reater or* ***E**qual*.

The solution 1 (6 steps) puts the radius in **t** by exchange of **x** and **t**, puts the value "1" in **x**, re-exchanges **x** and **t** to have "1" in **t** and the radius in **x** then tests if **x** is strictly lower (**INV GE**) than **t** :

"Is the radius strictly lower than 1 ?"

The solution 2 (5 steps) puts the radius in **t** by exchange of **x** and **t**, puts the value "0" in **x** then tests if **x** is greater or equal to **t** :

"Is zero greater or equal to the radius ?"

(exchange **x** with **t**, after the test, put again the radius in **x** for the continuation of the calculations, **RCL 01** being more expensive than one step)

Two other characteristics are to be explained :

1. The routine **LOG** allows the printing of the contents of the register **x** by formatting it with two decimals.

```
| 207 76 LBL |
| 208 28 LOG |
| 209 58 FIX |
| 210 02 02  |
| 211 99 PRT |
| 212 22 INV |
| 213 58 FIX |
| 214 92 RTN |
```

   FIX 2 fixes the display to two decimals, **PRT** prints the register **x** and **INV FIX** cancels the formatting.

2. The routine **WRT**, for printing the error message, ends by :

```
| 269 25 CLR |
| 270 35 1/X |
| 271 91 R/S |
```

CLR 1/X puts the register **x** to zero and divide 1 by **x** what provokes an error (division by zero !) and activates the blinking of the display to indicate the error, **R/S** stopping the program.
(This "trick" is often used to alert the user of a typing error.)

Further to the previous remarks, we can modify this program to improve it, indeed the concerned calculators (**TI58**, **TI58C** and **TI59**) having a memory for program limited in number of steps, one of the main concerns of programming is the economy of step, an approach of excessive economy being able to damage the legibility, thus the maintainability, of a program …

Here is thus a version "optimized" for this program :

000 69 OP	037 07 07	074 01 01	111 00 00
001 00 00	038 00 00	075 69 OP	112 69 OP
002 03 03	039 00 00	076 01 01	113 01 01
003 06 06	040 07 07	077 06 06	114 01 01
004 01 01	041 03 03	078 04 04	115 07 07
005 03 03	042 00 00	079 00 00	116 03 03
006 02 02	043 00 00	080 00 00	117 07 07
007 04 04	044 69 OP	081 00 00	118 03 03
008 03 03	045 04 04	082 00 00	119 05 05
009 06 06	046 69 OP	083 00 00	120 01 01
010 69 OP	047 05 05	084 00 00	121 07 07
011 01 01	048 25 CLR	085 69 OP	122 00 00
012 02 02	049 35 1/X	086 02 02	123 00 00
013 04 04	050 91 R/S	087 69 OP	124 69 OP
014 03 03	051 76 LBL	088 05 05	125 02 02
015 05 05	052 11 A	089 43 RCL	126 06 06
016 00 00	053 42 STO	090 01 01	127 04 04
017 00 00	054 01 01	091 71 SBR	128 65 *
018 04 04	055 32 X/T	092 28 LOG	129 06 06
019 01 01	056 00 00	093 65 *	130 22 INV
020 03 03	057 77 GE	094 02 02	131 28 LOG
021 01 01	058 00 00	095 65 *	132 95 =
022 69 OP	059 00 00	096 89 PI	133 69 OP
023 02 02	060 32 X/T	097 95 =	134 03 03
024 03 03	061 71 SBR	098 42 STO	135 69 OP
025 01 01	062 23 LNX	099 02 02	136 05 05
026 03 03	063 69 OP	100 69 OP	137 43 RCL
027 02 02	064 00 00	101 00 00	138 02 02
028 03 03	065 03 03	102 03 03	139 71 SBR
029 00 00	066 05 05	103 03 03	140 28 LOG
030 01 01	067 01 01	104 01 01	141 43 RCL
031 04 04	068 03 03	105 07 07	142 01 01
032 69 OP	069 04 04	106 03 03	143 33 X2
033 03 03	070 05 05	107 05 05	144 65 *
034 03 03	071 03 03	108 02 02	145 89 PI
035 05 05	072 02 02	109 04 04	146 95 =
036 01 01	073 03 03	110 03 03	147 42 STO

148	03	03	165	01	01	182	23	LNX	199	69	OP
149	69	OP	166	07	07	183	25	CLR	200	02	02
150	00	00	167	00	00	184	91	R/S	201	69	OP
151	03	03	168	00	00	185	76	LBL	202	03	03
152	06	06	169	06	06	186	23	LNX	203	69	OP
153	04	04	170	04	04	187	06	06	204	04	04
154	01	01	171	00	00	188	04	04	205	69	OP
155	03	03	172	00	00	189	06	06	206	05	05
156	05	05	173	69	OP	190	04	04	207	92	RTN
157	02	02	174	02	02	191	06	06	208	76	LBL
158	01	01	175	69	OP	192	04	04	209	28	LOG
159	01	01	176	05	05	193	06	06	210	58	FIX
160	03	03	177	43	RCL	194	04	04	211	02	02
161	69	OP	178	03	03	195	06	06	212	99	PRT
162	01	01	179	71	SBR	196	04	04	213	22	INV
163	01	01	180	28	LOG	197	69	OP	214	58	FIX
164	05	05	181	71	SBR	198	01	01	215	92	RTN

216 steps program instead of 272, is an economy of 56 steps!

The language

We can now approach the "verbs" by themes to make the most exhaustive possible presentation :

- Programming
- Additional keys
- Data entry
- The arithmetic operations
- Erasing
- Roots and powers
- Mathematical functions
- Trigonometry
- Printing
- Options of display
- Data management
- Jump statements
- Statistics
- Function keys
- Read / Write
- Library modules
- Special operations
- Other functions
- Hidden verb

Programming

- **CP** (2nd CP) In "calculator mode", erase all the program memory (putting in zero of all the steps), puts back to zero all addresses of return of the subroutines, returns the pointer of step to the step 000 and erases the register **t**.

- **LRN** (LRN) allows to enter in "programming mode" or to go out of it (return in the "calculator mode").

- **SST** (SST) in "programming mode", goes forward of one step.

- **BST** (BST) in "programming mode", goes backward of one step.

- **INS** (2nd Ins) in ‘programming mode’, insert one step before current step.

- **DEL** (2nd Del) in “programming mode”, delete the current step.

In “programming mode”, press on a key replaces the instruction of the current step.

Additional keys

• **2nd** (2nd) allows to use the second function of a key corresponding to the statement written above the key.

Example : 2nd SBR gives **LBL**

• **INV** (INV) for some functions (**EE**, **ENG**, **FIX**, **LOG**, **LNX**, **Yx**, **INT**, **SIN**, **COS**, **TAN**, **PRD**, **SUM**, **DMS**, **P/R**, **STA**, **AVR**, **LST**, **SBR**, **EQ**, **GE**, **IFF**, **STF**, **DSZ**, **WRT**), activate the inverse function.

In some cases, both touches 2nd and INV can be used.

Example : the decimal logarithm is obtained by 2nd lnx and the antilog of the decimal logarithm is obtained by INV 2nd lnx, that we shall write respectively **LOG** et **INV LOG**.

The "calculator mode" allows to type as well 2nd INV lnx as INV 2nd lnx, the programming mode admitting only this last notation (INV before 2nd) we disadvise to get used to the inverse entry (2nd before INV).

• **IND** (2nd Ind) allows the indirect addressing of the registers management instructions, jump statements and some others specific instructions.

Are concerned by this use :

- registers management instructions **STO**, **RCL**, **EXC**, **SUM**, **INV SUM**, **PRD**, **INV PRD**,
- jump statements **GTO**, **SBR**, **EQ**, **INV EQ**, **GE**, **INV GE**, **DSZ**, **INV DSZ**, **IFF**, **INV IFF**
- other specific instructions **PGM**, **OP**, **FIX**, **STF**.

The indirect addressing allows to use a register like a container of the address to be used.

<u>Example :</u>

5 STO 0 1 puts the value 5 in the register 01,

5 STO 2nd Ind 0 1 puts the value 5 in the register the address of which is in the register 01. (If the register 01 contains 4, the value 5 will be stored in the register 04)

The instructions **DSZ** and **IFF** can use a double indirect addressing because they manipulate at the same time a register number and an jump address.

INV 2nd 1 2nd Ind 0 1 2nd Ind 0 2

INV IFF IND 01 IND 02 means that if the flag, the number of which is contained in the register 01, is lowered (flag=0) the program will go to the address which is specified in the register 02.

The writing of the instructions with indirect addressing can be different from "*instruction name*" follow by IND according to this board :

Sequence of keys	Instructions	Codes
STO 2nd Ind	**ST***	72
RCL 2nd Ind	**RC***	73
2nd RCL 2nd Ind	**EX***	63
SUM 2nd Ind	**SM***	74
INV SUM 2nd Ind	**INV SM***	22 74
2nd SUM 2nd Ind	**PR***	64
INV 2nd SUM 2nd Ind	**INV PR***	22 64
GTO 2nd Ind	**GT***	83
SBR 2nd Ind	**SBR IND**	71 40
2nd 7 2nd Ind	**EQ**	67 40
INV 2nd 7 2nd Ind	**INV EQ**	22 67 40
2nd 4 2nd Ind	**GE**	77 40
INV 2nd 4 2nd Ind	**INV GE**	22 70 40
2nd 0 2nd Ind	**DSZ**	97 40
INV 2nd 0 2nd Ind	**INV DSZ**	22 97 40
2nd 1 2nd Ind	**IFF**	87 40
INV 2nd 1 2nd Ind	**INV IFF**	22 87 40
2nd LRN 2nd Ind	**PG***	62
2nd 9 2nd Ind	**OP***	84
2nd (2nd Ind	**FIX**	58 40
2nd RST 2nd Ind	**STF**	86 40
INV 2nd RST 2nd Ind	**INV STF**	22 86 40

Data entry

• **Numbers** (0 1 2 … 9) introduction of numbers in the display register **x.**

• **Decimal point** (•) introduces decimal point.

• **Sign** (+/–) changes the sign of the display register **x.**

• **PI** (2nd Π) introduces the value 3.14159265359 in the display register **x.**

• **|X|** (2nd |x|) returns the absolute value of the display register **x.**

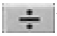
• **OP 10** (2nd Op 1 0) indicates if the value of the display register **x** is positive or negative.

Return **1** if **x > 0**, **0** if **x = 0**, **-1** if **x < 0**

• **INT** (2nd Int) returns the integer part of the register **x.**

• **INV INT** (INV 2nd Int) returns the decimal part of the register **x.**

The arithmetic operations

- **/** () division.

- ***** () multiplication.

- **-** () subtraction.

- **+** () addition.

- **=** () displays and "freezes" the result.

- **(** () opening parenthesis.

- **)** () closing parenthesis.

The calculators **TI58**/**TI58C**/**TI59** use the direct algebraic notation (AOS system).

The operations thus follow the rule of priority of the operators.

2 + 3 * 4 = will give 14 like 2 + (3 * 4) =, the parenthesis being useless, in that case.

On the other hand, (2 + 3) * 4 = will give as result 20.

Several levels of parenthesis can be used :

2 + 3 * 4 / 5 =	will give 2.8
((2 + 3) * 4) / 5 =	will give 4

Erasing

• **CE** (CE) erases the current introduction without interfering on the waiting operations and stops the blinking of the display.

• **CLR** (CLR) erases the register **x** and the current calculations. Also stops the blinking of the display.

• **CMS** (2nd CMs) erases all the data registers according to the defined partition (See **OP 16** et **OP 17**)

• **CP** (2nd CP) in programming mode, erases only the register **t**.

Roots et powers

• **X2** ([x^2]) raises to the square the value of the display register **x**.

• **SQR** ([$\sqrt{x}$]) return the square root of the display register **x.**
(if the register **x** contains a negative value [$\sqrt{x}$] activates the blinking of the display)

• **Yx** ([y^x]) raises the number contained in the display register to the entered power : [5] [y^x] [9] [=] will give 1953125

• **INV Yx** ([INV] [y^x]) calculates the x^th root of the number contained in the display register :
[1] [9] [5] [3] [1] [2] [5] [INV] [y^x] [9] [=] will give 5

Mathematical functions

• **1/X** (1/x) calculates the reciprocal of the content of the display register **x.**

• **LNX** (lnx) calculates the natural logarithm (base e) of the display register **x.** (if **x**<0 activates the blinking of the display)

• **INV LNX** (INV lnx) calculates the exponent (e^x) from the display register **x.**

• **LOG** (2nd log) calculates the decimal logarithm (base 10) of the display register **x.** (if **x**<0 activates the blinking of the display)

• **INV LOG** (INV 2nd log) calculates the antilog of the display register **x.** (10 raised to the power of **x**)
Often used in the programs to multiply by a multiple of 10 bigger than 100 :

RCL 01 * 1 0 0 0 0 0 = costs 10 steps
RCL 01 * 5 INV LOG = costs 7 steps !

• **P/R** (2nd P→R) converts the polar coordinates in Cartesian coordinates from registers **x** (*angle*) et **t** (*radius*) and returns the ordinate (*y*) in the register **x** and the abscissa (*x*) in the register **t**.

Example :

10 x/t	puts the radius in register **t**
35 P/R	puts the angle in the register **x**
	and returns the ordinate 5.73576436351
x/t	returns the abscissa 8.19152044289

• **INV P/R** (INV 2nd P→R) converts the Cartesian coordinates in polar coordinates from the ordinate (*y*) in the register **x** and the abscissa (*x*) in the register **t**, returns the angle in the register **x** and the radius in the register **t**.

It will be necessary to watch the choice of the angular mode (**DEG**, **RAD** or **GRD**) before proceeding to the calculation.

The angular mode defines the limits of the angle :

Angular mode	Lower border	Upper border
DEG	-90°	270°
RAD	-π/2	3π/2
GRD	-100	300

Trigonometry

• **DEG** (2nd Deg) selects the angular mode “degrees”.

• **RAD** (2nd Rad) selects the angular mode “radians”.

• **GRD** (2nd Grad) selects the angular mode “grads”.

• **SIN** (2nd sin) sine of the content of the display register **x.**

• **INV SIN** (INV 2nd sin) arcsine of the content of the display register **x.**

• **COS** (2nd cos) cosine of the content of the display register **x.**

• **INV COS** (INV 2nd cos) arccosine of the content of the display register **x.**

• **TAN** (2nd tan) tangent of the content of the display register **x.**

• **INV TAN** (INV 2nd tan) arctangent of the content of the display register **x.**

arccosecant = **1/X INV SIN**
arcsecant = **1/X INV COS**
arccotangent = **1/X INV TAN**

• **DMS** (2nd D.Ms) converts an angle measured in degrees, minutes, seconds in decimal degrees.

The input format is **DD.MMSSsss**, the decimal point has to separate the degrees of minutes.

• **INV DMS** (INV 2nd D.Ms) converts an angle measured in decimal degrees in degrees, minutes, seconds.

Printing

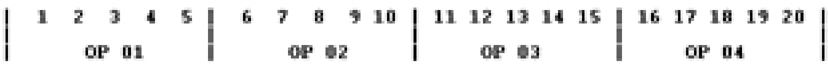
• **ADV** (2nd Adv) advances the paper of one line.

• **PRT** (2nd Prt) prints the register **x**.

• **LST** (2nd List) lists the program

• **INV LST** (INV 2nd List) prints the contents of registers since the register nn up to the last one, nn being the value in the register **x**.

• **OP 00** (2nd Op 0 0) erases the alphanumeric printing buffer.



• **OP 01** (2nd Op 0 1) allocates values to the group 1 (outside left) in the alphanumeric printing buffer.

• **OP 02** (2nd Op 0 2) allocates values to the group 2 (inside left) in the alphanumeric printing buffer.

• **OP 03** (2nd Op 0 3) allocates values to the group 3 (inside right) in the alphanumeric printing buffer.

• **OP 04** (2nd Op 0 4) allocates values to the group 4 (outside right) in the alphanumeric printing buffer.

• **OP 05** (2nd Op 0 5) prints the alphanumeric buffer.

• **OP 06** (2nd Op 0 6) prints, on the same line, the content of the display register **x** and the last 4 characters of the group 4 (outside right) of the alphanumeric buffer.

The coding of the alphanumeric printing buffer is made according the following table :

	0	1	2	3	4	5	6	7
0		0	1	2	3	4	5	6
1	7	8	9	A	B	C	D	E
2	-	F	G	H	I	J	K	L
3	M	N	O	P	Q	R	S	T
4	.	U	V	W	X	Y	Z	+
5	×	*	√	π	E	(	)	,
6	↑	%	↕	/	=	'	X	X̄
7	²	?	÷	!	‖	▲	Π	Σ

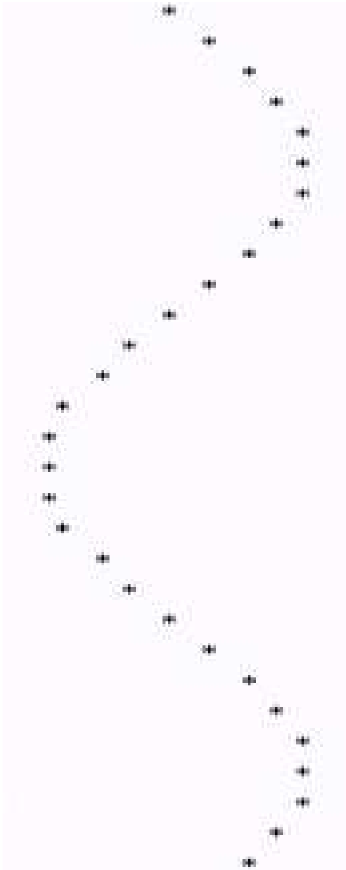
• **OP 07** (2nd Op 0 7) allows to draw a curve by printing one asterisks in a column 0 to 19.

Single one asterisk is printed on every line in the column corresponding to the integer part of the display register **x** in the range of value -1 < **x** < 20.

Example :

Sinusoid 18 degrees by 18 degrees.

```
000   76   LBL
001   11   A
002   43   RCL
003   01   01
004   38   SIN
005   85   +
006   01   01
007   95   =
008   65   *
009   09   09
010   93   .
011   09   09
012   95   =
013   69   OP
014   07   07
015   01   01
016   08   08
017   44   SUM
018   01   01
019   11   A
```



+1 = * 9.9 = allows to "calibrate" the value in an interval from 0 to 19.8 to determine the column of the asterisk.

• **OP 08** (2nd Op 0 8) lists the labels of the program.

```
001 25 CLR
015 15 E
108 11 A
191 12 B
267 35 1/X
294 23 LNX
322 24 CE
354 45 YX
423 33 X2
```

Options of display

The standard display of the TI is made on 10 digits, while the internal management is on 13 digits for more precision in the calculations.

The display is thus limited to the numbers included between .0000000001 et 9999999999 (In absolute value, the sign not taking a place on the ten characters).

The numbers exceeding these limits must be keyed in scientific notation.

So the number

-0.0000000000000000000000000000001234567

could be written

$$-1.234567 * 10^{-31}$$

and will be introduced in the following way

1.234567 [+/-] [EE] **31** [+/-]

and will be displayed

-1.234567 being the ***mantissa*** and **-31** being the ***exponent***

The mantissa is thus limited to 7 characters and the exponent to 2.

• **EE** (EE) allows to pass in scientific notation

• **INV EE** (INV EE) allows to cancel the scientific notation.

• **ENG** (2nd Eng) allows to pass in engineering notation.
Variant of the scientific notation, the engineering notation is characterized by an adjustment of the mantissa and the exponent to have an exponent multiple of three.
So -1.234567-31 will give -123.4567-33 in engineering notation.

• **INV ENG** (INV 2nd Eng) allows to cancel the engineering notation.

The engineering notation allows to represent the numbers in usual units of measure :

10^n	Prefix	Decimal number
10^{24}	Yotta	1 000 000 000 000 000 000 000 000
10^{21}	Zetta	1 000 000 000 000 000 000 000
10^{18}	Exa	1 000 000 000 000 000 000
10^{15}	Peta	1 000 000 000 000 000
10^{12}	Tera	1 000 000 000 000
10^9	Giga	1 000 000 000
10^6	Mega	1 000 000
10^3	Kilo	1 000
10^2	Hecto	100
10^1	Deca	10
10^0	*Unit*	1
10^{-1}	Deci	0,1
10^{-2}	Centi	0,01
10^{-3}	Milli	0,001
10^{-6}	Micro	0,000 001
10^{-9}	Nano	0,000 000 001
10^{-12}	pico	0,000 000 000 001
10^{-15}	femto	0,000 000 000 000 001
10^{-18}	atto	0,000 000 000 000 000 001
10^{-21}	zepto	0,000 000 000 000 000 000 001
10^{-24}	yocto	0,000 000 000 000 000 000 000 001

• **FIX** (2nd Fix) allows to choose the decimalization.

The digit following the key **FIX** indicates the number of fixed decimals (0 à 8).

• **FIX IND** (2nd Fix 2nd Ind) allows to choose, or cancel, the decimalization in an indirect way.

The number following the key **FIX** indicates the register number which contains the number of fixed decimals (0 to 8), or the value 9 to go back in floating decimal point.

• **INV FIX** (INV 2nd Fix) cancels the decimalization and goes back in floating decimal point. (**FIX 9** has the same effect)

Data management

• **X/T** (x⇄t) exchanges the contents of the registers **x** et **t**.

• **STO** (STO) stores the content of the register **x** in the register nn.

• **ST*** (STO 2nd Ind) stores the content of the register **x** in a register the address of which is contained in the register nn.

1 5 STO 2nd Ind 0 1

1 5 ST* 01 puts the value 15 in the register the address of which is stored in the register 01.

If the register 01 contains 20, puts 15 in the register 20,

If the register 01 contains 7, puts 15 in the register 7…

• **RCL** (RCL) puts the content of the register nn in the register **x.**

• **RC***(RCL 2nd Ind) puts the content of the register the address of which is contained in the register nn in the register **x.**

• **SUM** (SUM) adds the content of the register **x** to the content of the register nn.

• **SM*** (SUM 2nd Ind) adds the content of the register **x** to the content of the register the address of which is contained in the register nn.

• **INV SUM** (INV SUM) subtracts the content of the register **x** from the content of the register nn.

• **INV SM*** (INV SUM 2nd Ind) subtracts the content of the register **x** from the content of the register the address of which is contained in the register nn.

• **PRD** (2nd Prd) multiplies the content of the register nn by the content of the register **x.**

• **PD*** (2nd Prd 2nd Ind) multiplies the content of the register the address of which is contained in the register nn by the content of the register **x.**

• **INV PRD** (INV 2nd Prd) divides the content of the register nn by the content of the register **x.**

• **INV PD*** (INV 2nd Prd 2nd Ind) divides the content of the register the address of which is contained in the register nn by the content of the register **x.**

• **EXC** (2nd Exc) exchanges the content of the register nn with the content of the register **x.**

• **EX*** (2nd Exc 2nd Ind) exchanges the content of the register the address of which is contained in the register nn with the content of the register **x.**

• **OP 2**n (2nd Op 2 **n**) increments the value of the register n of 1. Applies to registers 0 to 9.

OP 21 is same as **1 SUM 01**

• **OP 3**n (2nd Op 3 **n**) decrements the value of the register n of 1. Applies to registers 0 to 9.

OP 31 is same as **1 INV SUM 01**

Jump statements

- **LBL** ([2nd] [Lbl]) allows to define program labels.

2 kinds of labels are usable :

- "user" labels (or function keys) : [A], [B], [C]…
- ordinary labels : all keys can be then used as labels with the exception of the digital touches ([0], [1], [2]…) and keys [2nd], [LRN], [BST], [SST], [2nd] [Ins], [2nd] [Del], [2nd] [Ind] et the specific key [R/S] (authorized but strongly disadvised).

  Naturally, in the case of use of a key as label, this last one will not be treated as instruction in the program execution but only as label.

- **GTO** ([GTO]) allows to jump to a precise address. moves the pointer of step at the indicated address and, in programming mode, continues the execution of the program from this address.

Two addressing are possible

- Logical addressing : **GTO** is then followed by a name defined besides as label.

  Example :

  [GTO] [x^2]… and somewhere else in the program [2nd] [Lbl] [x^2]…

- Absolute addressing : **GTO** is then followed by an address of step.

  Example :

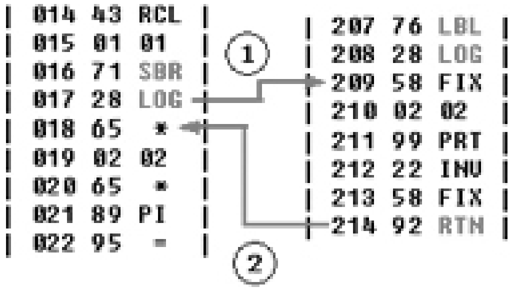
  [GTO] [1] [2] [3] which sends to the step 123

The advantage of the logical addressing is in the clarity and the legibility of the program, and in case of addition or deletion of a the step in the program, nothing changes the logical link. (This method costs at least 4 steps.)
The absolute addressing allows an economy of step (3 steps) but imposes a vigilance for the maintenance because adding or deleting a step in program moves the address of the step aimed by the **GTO** if these updates are made before the address of origin.

• **GO*** (GTO 2nd Ind) allows the relative addressing in a program by using a data register which contains the address of the step aimed by the jump.
<u>Example :</u>
GTO 2nd Ind 0 1 means that the address of jump is contained in the register **01**.

• **SBR** (SBR) allows the jump to the address specified, like for **GTO**, but the first return statement **RTN** (INV SBR) will send back the pointer behind the calling **SBR**.
SBR uses, like **GTO**, either the logical addressing, or the absolute addressing.

Example :



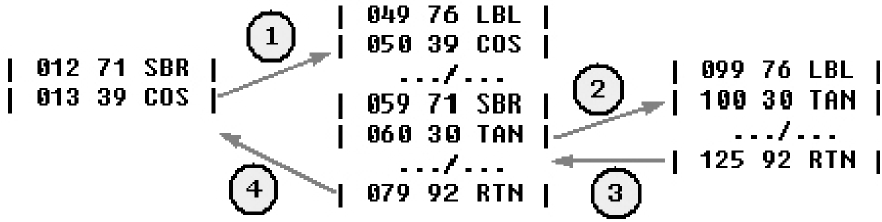
1 call of the procedure beginning at the label **LOG**,

2 return behind the call.

• **SBR IND** (SBR 2nd Ind) allows the relative addressing in a program by using a data register which contains the address of the step aimed by the procedure call.
The first return statement **RTN** (INV SBR) will send back the pointer behind the calling **SBR**.

• **RTN** (INV SBR) return from procedure called by **SBR** (*Return*).

In the case or the execution meets an instruction **RTN** while no **SBR** statement is in expectation of a return, then **RTN** behaves as **R/S** and stops the program.



In the case of imbricated calls, the return is made behind the last call made and so on until the exhaustion of the pile containing the return addresses.

• **RST** (RST) returns the steps pointer to the step 000, puts back to zero the return addresses of subroutines and puts back flags to zero (“low position”).

• **R/S** (R/S) in "calculator mode" launches the program from the current pointer or stops the running program, as verb in a program stops the program.

• **EQ** (2nd x=t) conditional test, goes to the specified address if the register **x** is equal to the register **t**, else the program continues in sequence.

EQ uses, like **GTO**, either the logical addressing, or the absolute addressing.

Example :

2nd x=t lnx	goes to label **LNX** if **x** = **t**
2nd x=t 1 2 3	goes to address **123** if **x** = **t**

• **EQ IND** (2nd x=t 2nd Ind) conditional test, using a data register which contains the address of the step aimed if the register **x** is equal to the register **t**, else the program continues in sequence.

• **INV EQ** (INV 2nd x=t) conditional test, goes to the specified address if the register **x** is different from the register **t**, else the program continues in sequence.

• **INV EQ IND** (INV 2nd x=t 2nd Ind) conditional test, using a data register which contains the address of the step aimed if the register **x** is different from the register **t**, else the program continues in sequence.

• **GE** (2nd x≥t) conditional test, goes to the specified address if the register **x** is greater than or equal to the register **t**, else the program continues in sequence.
GE uses, like **GTO**, either the logical addressing, or the absolute addressing.

• **GE IND** (2nd x≥t 2nd Ind) conditional test, using a data register which contains the address of the step aimed if the register **x** is greater than or equal to the register **t**, else the program continues in sequence.

• **INV GE** (INV 2nd x≥t) conditional test, goes to the specified address if the register **x** is less than the register **t**, else the program continues in sequence.

• **INV GE IND** (INV 2nd x≥t 2nd Ind) conditional test, using a data register which contains the address of the step aimed if the register **x** is less than the register **t**, else the program continues in sequence.

Conditional jumps		
Equal	**EQ**	2nd x=t
Different	**INV EQ**	INV 2nd x=t
Greater or equal	**GE**	2nd x≥t
Less	**INV GE**	INV 2nd x≥t

Except the conditional tests by comparison of registers x and t, the TI allows to manage up to 10 flags, the state of which (raised or lowered) can be tested for jumping.
Flags are numbered from 0 to 9.

• **STF** (2nd St flg) raises specified flag (*Set Flag*).

Example :

2nd St flg 1 raises flag 1

• **INV STF** (INV 2nd St flg) lowers specified flag.

Example :

INV 2nd St flg 1 lowers flag 1

• **IFF** (2nd If flg) conditional test, goes to the specified address if the flag is raised, else the program continues in sequence.
IFF uses, like **GTO**, either the logical addressing, or the absolute addressing.

• **IFF IND** (2nd If flg 2nd Ind) conditional test, using a data register which contains the address of the step aimed if the specified flag is raised, else the program continues in sequence.

• **INV IFF** (INV 2nd If flg) conditional test, goes to the specified address if the flag is lowered, else the program continues in sequence.

• **INV IFF IND** (INV 2nd If flg 2nd Ind) conditional test, using a data register which contains the address of the step aimed if the specified flag is lowered, else the program continues in sequence.

• **DSZ** (2nd Dsz) conditional test allowing to manage iterative sequences. manipulates a data register (0 to 9 only) and uses, like **GTO**, either the logical addressing, or the absolute addressing.

DSZ proceeds in two stages :

- Decrements the tested register if the value is positive (or increments it, if the value is negative)
- Tests if the register contains zero : if **NO** goes to the specified address, if **YES** continues in sequence.

• **DSZ IND** (2nd Dsz 2nd Ind) conditional test allowing to manage iterative sequences. manipulates a data register (0 to 9 only) and uses a data register which contains the address of the aimed step if the test is satisfied.

• **INV DSZ** (INV 2nd Dsz) conditional test allowing to manage iterative sequences. manipulates a data register (0 to 9 only) and uses, like **GTO**, either the logical addressing, or the absolute addressing.

INV DSZ proceeds in two stages :

- Decrements the tested register if the value is positive (or increments it, if the value is negative)
- Tests if the register contains zero : if **YES** goes to the specified address, if **NO** continues in sequence.

• **INV DSZ IND** (INV 2nd Dsz 2nd Ind) conditional test allowing to manage iterative sequences. manipulates a data register (0 to 9 only) and uses a data register which contains the address of the aimed step if the test is satisfied.

Statistics

The TI manages the statistics for a sample on two values representing a point on a plan of axes ***x*** and ***y***.
On the population of points, we can determine the average, the variance, the standard deviation …

• Initialization of the statistical data : The statistics use 6 data registers, and the register **t**, which must be put back to zero before any new input.

Register	01	02	03	04	05	06
Content	$\sum y$	$\sum y^2$	N	$\sum x$	$\sum x^2$	$\sum xy$

This initialization can be made :

- Or manually : 2nd CMs what erases all the registers,
- Or manually : CLR STO 0 1 STO 0 2 STO 0 3 STO 0 4 STO 0 5 STO 0 6,
- Or by using the initialization routine of the module 01 of the basic library (**ML-01**) : 2nd Pgm 0 1 SBR CLR

• **STA** (2nd Σ+) data input.

- or ***x*** x⇌t ***y*** 2nd Σ+ for entering ***x*** and ***y***
- or ***y*** 2nd Σ+ for entering ***y*** alone

the rank **i** is displayed for each couple (x_i, y_i) entered.

• **INV STA** (INV 2nd Σ+) cancelling data.

- or ***x*** x⇌t ***y*** 2nd Σ+ for cancelling ***x*** and ***y***
- or ***y*** 2nd Σ+ for cancelling ***y*** alone

• **AVR** (2nd $\bar{x}$) calculates and displays the average of the various values of ***y*** (x⇌t displays the average of the various values of ***x***).

• **INV AVR** (INV 2nd $\bar{x}$) calculates and displays the standard deviation of the various values of ***y*** (x⇌t displays the standard deviation of the various values of ***x***).

• **OP 11** (2nd Op 1 1) calculates and displays the variance of the various values of ***y*** (x⇌t displays the variance of the various values of ***x***).

• **OP 12** ([2nd] [Op] [1] [2]) **Linear regression** – calculates and displays the y-intercept (intersection point of the graph of function with the ***Y*** axis for ***x*** = 0) and [x≷t] displays the slope.

• **OP 13** ([2nd] [Op] [1] [3]) **Linear regression** – calculates and displays the correlation coefficient.

• **OP 14** ([2nd] [Op] [1] [4]) **Linear regression** – calculates and displays the value of ***y*** for an entered value of ***x***.

• **OP 15** ([2nd] [Op] [1] [5]) **Linear regression** - calculates and displays the value of ***x*** for an entered value of ***y***.

Function keys

The function keys (or user keys) are among 10. They are usable in the programs as label and can be called by the jump statements (**GTO**, **GE**, **EQ**…).

The use of one key alone is equivalent to **SBR**. (***Example :*** SBR A = A)

In "calculator mode", they allow to launch the program from a precise point.

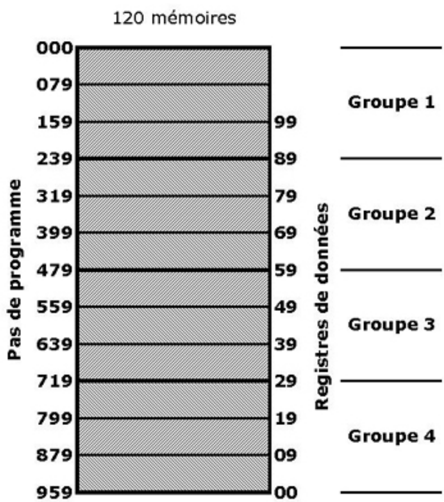
- **A** (A)
- **B** (B)
- **C** (C)
- **D** (D)
- **E** (E)

- **A'** (2nd A')
- **B'** (2nd B')
- **C'** (2nd C')
- **D'** (2nd D')
- **E'** (2nd E')

Read / Write

The instructions of reading / writing are usable only on **TI59** because she is the only one to be endowed with a magnetic cards reader.

The **TI59** possesses up to 120 storage memories distributed between 4 groups.



A magnetic card for TI59 contains 2 tracks which can record, each, one group.

Two cards are thus necessary to record all the memory of a TI59.

- **WRT** (2nd Write) writing on the magnetic card (must be preceded by the number of the group to be recorded 1, 2, 3 or 4)

• **INV WRT** (INV 2nd Write) reading of the magnetic card (if preceded by the number of the group, with negative sign ***-n***, forces the reading in the group ***n***)

Library modules

With the calculator, a pluggable module is always supplied.
Named "Master Library", it contains twenty five utility programs.
It can be replaced by other one of the modules marketed by Texas Instruments.

#	Code	Title
01	ML	Master Library
02	ST	Applied Statistics
03	RE	Real Estate / Investment
04	SY	Surveying
05	NG	Marine Navigation
06	AV	Aviation
07	LE	Leisure Library
08	SA	Securities Analysis
09	BD	Business Decisions
10	MU	Math / Utilities
11	EE	Electrical Engineering
12	FM	Agriculture
13	RP	RPN Simulator

• **PGM** (2nd Pgm) allows to activate, or to deactivate, a program of the library module.

- 2nd Pgm nn activates the program nn,
- 2nd Pgm 0 0 deactivates the current program.

2nd Pgm 0 1 SBR 2nd Write allows to display the number of the plugged module and prints its name if the printer is connected.

Example :

The program 24 of the "Master Library" converts from/to decimal length units (cm, m, km) from/to British length units (inch, foot, yard, miles)

So to know how much 1 yard makes of meters it is necessary to introduce the sequence :

2nd Pgm 2 4 1 C

ML-24 UNIT CONVERSIONS (I)				
cm -> in	m -> ft	m -> yd	km -> mi	n.mi -> mi
in -> cm	ft -> m	yd -> m	mi -> km	mi -> n.mi

• **OP 09** (2nd Op 0 9) loads the activated program in the program memory of the TI. (Erases the program in memory to replace it!)

Special operations

• **OP 01** to **OP 08** see ***Printing***

• **OP 09** see ***Library modules***

• **OP 10** see ***Data entry***

• **OP 11** to **OP 15** see ***Statistics***

• **OP 16** (2nd Op 1 6) displays the memory partition : distribution between the program steps and the data registers.

• **OP 17** (2nd Op 1 7) positions the memory partition : distribution between the program steps and the data registers, by group of 10 registers.

TI59	OP 17	TI58/TI58C
959-00	0	479-00
879-09	1	399-09
799-19	2	319-19
719-29	3	239-29
639-39	4	159-39
559-49	5	079-49
479-59	6	000-59
399-69	7	
319-79	8	
239-89	9	
159-99	10	

Example :

On **TI58**, 3 2nd Op 1 7 will give **239.29** that means 240 steps (from 000 to 239) and 30 registers (from 00 to 29)

• **OP 18** (2nd Op 1 8) raises the flag 7, if no error of execution is encountered.

• **OP 19** (2nd Op 1 9) raises the flag 7, if an error of execution is encountered.

• **OP 40** (2nd Op 4 0) on **TI58C** only, raises the flag 7 if the printer is connected.

• **OP IND** (2nd Op 2nd Ind) uses the content of a register nn to determine which **OP**eration is applicable.

<u>Example :</u>

2nd Op 2nd Ind 0 1 uses the content of the register 01.

- If the register 01 contains 16, displays the partition (idem **OP 16**),
- If the register 01 contains 0, erases the alphanumeric printing buffer (idem **OP 00**).

Other functions

• **PAU** (2nd Pause) allows to preserve half a second the display of the register **x** during the execution of the program. Several pauses can follow one another for prolonging the display.

• **NOP** (2nd Nop) no operation. Instruction without any effect on the execution. Serves to insert a step so as anticipate a space between two sequences of program or to replace an instruction without provoking a gap in the numbering of steps, instead of making DEL .

Hidden verb

• **HIR** (*no key*) The TI59/58/58C hides 8 internal registers used by the system for its own functions.

The system based on the direct algebraic notation manages an AOS pile in these registers to put in on hold the numbers in the calculations to several operators to respect the priority of these operators.

Then complex functions (**STA**, **AVR**, **P/R**, **DMS**) store intermediate results in these registers as well as the statistical functions (**OP 11**, **OP 12**, **OP 13**, **OP 14**, **OP 15**) and the alphanumeric printing functions (**OP 00**, **OP 01**, **OP 02**, **OP 03**, **OP 04**).

A particular instruction exists to manipulate these registers.

Officially, this instruction does not exist :

- not a word in the **TI** documentations,
- not a key to input it into a program.

And nevertheless …

It is so necessary to use trickery to introduce this instruction with manipulations which are more similar to the juggling than to the programming.

Thus, we will create a small program…

First, we choose the ‘programming mode” (LRN)after having erased the contents of the memory program (2nd CP).

then we enter the following instructions :

2nd Lbl A STO 8 2 STO 1 1 R/S

that gives, printed with 2nd List :

```
000  76 LBL
001  11   A
002  42 STO
003  82  82
004  42 STO
005  11  11
006  91 R/S
```

We can now modify our program by deleting the step 004 then the step 002 :

- RST, LRN then SST SST SST SST for going to step 004
- 2nd Del for deleting the step 004
- BST BST for going to step 002
- 2nd Del for deleting the step 002.

We get :

```
000  76 LBL
001  11   A
002  82 HIR
003  11  11
004  91 R/S
```

We can see that the code **82** was translated into **HIR** by the printer.

Here is thus our hidden instruction which appears.

In "calculator mode", let us enter the small following calculation :

7 + 3 × 4 =

which gives us 19 because the multiplication is priority on the addition.

Now we will execute our small program by keying A on the function keys.

The number 7 appears to the display.

It is the first number of our calculation which was put on hold (stored in the AOS pile) so that the multiplication can be made first.

HIR 12 would give 3 in the display(posting), showing us that the second number of our operation was also stored in the AOS pile.

• **HIR 0**n (0 ≤ n ≤ 8) stores the content of the register **x** in the internal register n. (≈ **STO**)

• **HIR 1**n (0 ≤ n ≤ 8) recalls the content of the internal register n in the register **x**. (≈ **RCL**)

• **HIR 3**n (0 ≤ n ≤ 8) adds the content of the register **x** to the internal register n. (≈ **SUM**)

• **HIR 4**n ($0 \leq n \leq 8$) multiplies the content of the internal register n by the register **x**. (≈ **PRD**)

• **HIR 5**n ($0 \leq n \leq 8$) subtracts the content of the register **x** of the internal register n. (≈ **INV SUM**)

• **HIR 6**n ($0 \leq n \leq 8$) divides the content of the internal register n by the register **x**. (≈ **INV PRD**)

Summary table of the instructions

Code	Instr.	Keys
00	**0**	0
01	**1**	1
02	**2**	2
03	**3**	3
04	**4**	4
05	**5**	5
06	**6**	6
07	**7**	7
08	**8**	8
09	**9**	9
10	**E'**	2nd E'
11	**A**	A
12	**B**	B
13	**C**	C
14	**D**	D
15	**E**	E
16	**A'**	2nd A'
17	**B'**	2nd B'
18	**C'**	2nd C'
19	**D'**	2nd D'
20	**CLR**	CLR
21	**2nd**	2nd
22	**INV**	INV
23	**LNX**	ln$x$
24	**CE**	CE
25		
26		
27		
28	**LOG**	2nd log
29	**CP**	2nd CP
30	**TAN**	2nd tan

Code	Instr.	Keys
31	**LRN**	LRN
32	**X/T**	$x \leftrightarrows t$
33	**X2**	x^2
34	**SQR**	$\sqrt{x}$
35	**1/X**	$1/x$
36	**PGM**	2nd Pgm
37	**P/R**	2nd P→R
38	**SIN**	2nd sin
39	**COS**	2nd cos
40	**IND**	2nd Ind
41	**SST**	SST
42	**STO**	STO
43	**RCL**	RCL
44	**SUM**	SUM
45	**YX**	y^x
46	**INS**	2nd Ins
47	**CMS**	2nd CMs
48	**EXC**	2nd Exc
49	**PRD**	2nd Prd
50	**IXI**	2nd $\|x\|$
51	**BST**	BST
52	**EE**	EE
53	**(**	(
54	**)**	)
55	**/**	÷
56	**DEL**	2nd Del
57	**ENG**	2nd Eng
58	**FIX**	2nd Fix
59	**INT**	2nd Int
60	**DEG**	2nd Deg
61	**GTO**	GTO

Code	Instr.	Keys
62	**PG***	2nd Pgm 2nd Ind
63	**EX***	2nd Exc 2nd Ind
64	**PR***	2nd Prd 2nd Ind
65	*****	×
66	**PAU**	2nd Pause
67	**EQ**	2nd x=t
68	**NOP**	2nd Nop
69	**OP**	2nd Op
70	**RAD**	2nd Rad
71	**SBR**	SBR
72	**ST***	STO 2nd Ind
73	**RC***	RCL 2nd Ind
74	**SM***	SUM 2nd Ind
75	**-**	−
76	**LBL**	2nd Lbl
77	**GE**	2nd x ≥ t
78	**STA**	2nd Σ+
79	**AVR**	2nd $\bar{x}$
80	**GRD**	2nd Grad

Code	Instr.	Keys
81	**RST**	RST
82	**HIR**	
83	**GO***	GTO 2nd Ind
84	**OP***	2nd Op 2nd Ind
85	**+**	
86	**STF**	2nd St flg
87	**IFF**	2nd If flg
88	**DMS**	2nd D.Ms
89	**PI**	2nd π
90	**LST**	2nd List
91	**R/S**	R/S
92	**RTN**	INV SBR
93	**.**	•
94	**+/-**	+/−
95	**=**	=
96	**WRT**	2nd Write
97	**DSZ**	2nd Dsz
98	**ADV**	2nd Adv
99	**PRT**	2nd Prt

Comparative tests

For a same feature, several solutions of programming can appear. The cost, in number of steps, or the execution duration can influence our programming choice according to the studied case.

Sometimes, the economy of steps can be crucial, the memory being relatively limited.

Occasionally the speed of execution will be privileged as criterion of optimization.

Fortunately, considering the nature of the programs developed for this kind of machine, these concerns will be often superfluous.

Nevertheless, study the various hypotheses, for resolution of programs cases, can be useful to understand the mechanisms of the language.

Reset the registers

A great classic of programming with this kind of machine is to reset only some registers.

Indeed, to reset all the registers, all at the same time, we have the instruction 2nd CMs who answers everything the possible criteria : quickness and only 1 program step.

But to put back to zero a set of registers we shall have three choices of programming :

- Programming by decrement,
- Manipulation of partitions,
- Use of the libraries programs.

The 3 approached methods are on the basis of a reset of registers 00 to 09 and of registers 00 to 29.

<u>1st method :</u> Programming by decrement

...	n	
...	n	nn = register max (9 or 29)
...	STO	
...	00	
...	CLR	
xxx	ST*	
...	00	
...	DSZ	
...	0	
...	0x	xxx = jump address
...	xx	

2nd method : Manipulation of partitions

...	n	
...	OP	n = number of the memory group
...	17	(1 for 00 to 09, 3 for 00 to 29 [*])
...	CMS	
...	m	
...	OP	m = return to initial partition
...	17	(5 for example for 159-39 [*])

[*] concern the **TI58** and **TI58C**

3rd method : Use of the libraries programs

...	n	
...	n	nn = register max (9 or 29)
...	PGM	
...	01	
...	SBR	
...	00	
...	12	Uses Master Library ML-01

or

...	n	
...	n	nn = register max (9 or 29)
...	PGM	
...	01	
...	SBR	
...	00	
...	04	Uses Maths Utilities MU-10

Of course, these three methods do not give the same result in term of number of steps and in execution duration :

	1st method		**2nd method**		**3rd method**	
memories	**steps**	**time**	**steps**	**time**	**steps**	**time**
00 to 09	10	3,5 s	7	0,4 s	6	2,4 s
00 to 29	11	10,5 s	7	0,4 s	7	7 s

The 1st method which appears the most sensible in term of programming is nevertheless the most expensive in term of steps as well as in term of time. This method remains nevertheless the most used.

The 2nd method is the winner in duration of execution but does not offer compatibility between the **TI58**/**58C** and the **TI59** because the definitions of memory groups are not the same (See **OP 17**).

The 3rd method, not often used, is a good compromise and would deserve more attention.

Repetitive sequence

In a program, the presence of sequences of similar instructions in several places of the code is rather frequent.
The question which arises then is to know if it is sensible, or not, to convert, this repetitive sequence, in a procedure with a call, every time that it seems necessary.
Although some methods laud an excessive modularity, the purpose is not to systematize this approach but rather to consider when it can be beneficial.

The following examples are based on the principle of three instructions repeated to three different places in the same program. (2nd Int STO 0 1)

Solution 1 :

Writing of the instruction sequence as often as necessary.

3 sequences

3 instructions

⇩

9 steps

```
010  59  INT
011  42  STO
012  01  01
  .../...
032  59  INT
033  42  STO
034  01  01
  .../...
076  59  INT
077  42  STO
078  01  01
```

nbr steps = nbr sequences * nbr instructions

Solution 2 :

Calling a procedure by relative addressing (*label*).

3 calls

3 instructions

⇩

12 steps

```
010  71  SBR
011  23  LNX
   .../...
032  71  SBR
033  23  LNX
   .../...
076  71  SBR
077  23  LNX
   .../...
100  76  LBL
101  23  LNX
102  59  INT
103  42  STO
104  01  01
105  92  RTN
```

nbr steps = (nbr calls * 2) + (nbr instructions + 3)

The following summary table allows us to determine from how much of instructions and from how many calls, we can get a substantial economy of steps.

	Calls	1		2		3		4		5		6		7		8	
	Solution	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
Instructions	1	1	6	2	8	3	10	4	12	5	14	6	16	7	18	8	20
	2	2	7	4	9	6	11	8	13	10	15	12	17	14	19	16	21
	3	3	8	6	10	9	12	12	14	15	16	18	18	21	20	24	22
	4	4	9	8	11	12	13	16	15	20	17	24	19	28	21	32	23
	5	5	10	10	12	15	14	20	16	25	18	30	20	35	22	40	24
	6	6	11	12	13	18	15	24	17	30	19	36	21	42	23	48	25
	7	7	12	14	14	21	16	28	18	35	20	42	22	49	24	56	26
	8	8	13	16	15	24	17	32	19	40	21	48	23	56	25	64	27
	9	9	14	18	16	27	18	36	20	45	22	54	24	63	26	72	28
	10	10	15	20	17	30	19	40	21	50	23	60	25	70	27	80	29

We would also have been able to study a third solution …

Solution 3 :

Calling a procedure by absolute addressing (*address*)

3 calls

3 instructions

⇩

13 steps

```
010  71  SBR
011  01  01
012  02  02
   .../...
032  71  SBR
033  01  01
034  02  02
   .../...
076  71  SBR
077  01  01
078  02  02
   .../...
102  59  INT
103  42  STO
104  01  01
105  92  RTN
```

nbr steps = (nbr calls * 3) + (nbr instructions + 1)

Loop test

RST is the command which returns the execution pointer to the beginning of the program memory.

In fact, 3 possibilities allow to return to the beginning of the partition :

- RST, Of course, but this instruction also puts back flags to zero as well as the return addresses of the subroutines,
- GTO 0 0 0,
- GTO *label*.

Three simple small programs can help to compare the performances of every case.

```
000   85   +
001   01   01
002   81   RST
```

Case #1

```
000   85   +
001   01   01
002   61   GTO
003   00   00
004   00   00
```

Case #2

```
000   76   LBL
001   23   LNX
002   85   +
003   01   01
004   61   GTO
005   23   LNX
```

Case #3

Every execution is launched by RST R/S then stopped by R/S after 60 seconds.

	Count during 1 mn		
	Result (+1)	**Steps**	**Ratio**
Case #1	**538**	**3**	**179.33**
Case #2	**299**	**5**	**59.80**
Case #3	**350**	**6**	**58.33**

The test seems to prove, except RST (Case #1), that the relative addressing (Case #3) would appreciably be more successful than the absolute addressing (Case #2).

On the other hand, RST seems interesting, despite rare usage, because economical in term of steps, this instruction is of the fastest and would deserve a little more of attention.

Procedure call

The kind of addressing, absolute (address) or relative (label), is usable with all the conditional or direct jump instructions.

The loop test previously executed would tend to prove that the relative addressing would appreciably be more successful than the absolute addressing, but other comparisons bring to refine this judgment.

For every kind of addressing, 3 cases will allow us of to know more about it :

- #1 : Calling a procedure in the beginning of the program memory,
- #2 : Calling a procedure in the middle of the program memory,
- #3 : Calling a procedure at the end of the program memory.

1) relative addressing :

```
000  71  SBR
001  45  YX
002  81  RST
003  76  LBL
004  45  YX
005  85  +
006  01  01
007  92  RTN
```

Case #1

```
000  71  SBR
001  45  YX
002  81  RST
  .../...
235  76  LBL
236  45  YX
237  85  +
238  01  01
239  92  RTN
```

Case #2

```
000  71  SBR
001  45  YX
002  81  RST
  .../...
475  76  LBL
476  45  YX
477  85  +
478  01  01
479  92  RTN
```

Case #3

2) absolute addressing :

```
000  71  SBR
001  00  00
002  04  04
003  81  RST
004  85   +
005  01   1
006  92  RTN
```

Case #1

```
000  71  SBR
001  02  02
002  37  37
003  81  RST
   .../...
237  85   +
238  01   1
239  92  RTN
```

Case #2

```
000  71  SBR
001  04  04
002  77  77
003  81  RST
   .../...
477  85   +
478  01   1
479  92  RTN
```

Case #3

Each program is launched by (RST R/S) then stopped by R/S after 60 seconds.

		Count during 1 mn		
		Case #1	**Case #2**	**Case #3**
Addressing	**relative**	**224**	**66**	**32**
	Absolute	**208**	**196**	**186**

These programs prove us that both kinds of addressing are competitive for the low addresses but that the absolute addressing is faster for the high addresses.

The calculator and its statistical functions can serve us to make an analysis of tendency :

1) For absolute addressing, we will introduce our sample :

Step address **X/T** *count* **STA**

4 x⇌t 2 0 8 2nd Σ+

2 3 7 x⇌t 1 9 6 2nd Σ+

4 7 7 x⇌t 1 8 6 2nd Σ+

We can calculate various values of y (*count*) of the regression line by introducing various values of x (step *address*) followed by **OP 14**.

9 2nd Op 1 4 gives 207,…

5 9 2nd Op 1 4 gives 204,…

1 0 9 2nd Op 1 4 gives 202,…

and so on until 459.

2) For relative addressing, we will introduce our sample :

Step address **X/T** *count* **STA**

4 x⇌t 2 2 4 2nd Σ+

2 3 7 x⇌t 6 6 2nd Σ+

4 7 7 x⇌t 3 2 2nd Σ+

We can calculate various values of y (*count*) of the regression line by introducing various values of x (step *address*) followed by **OP 14**.

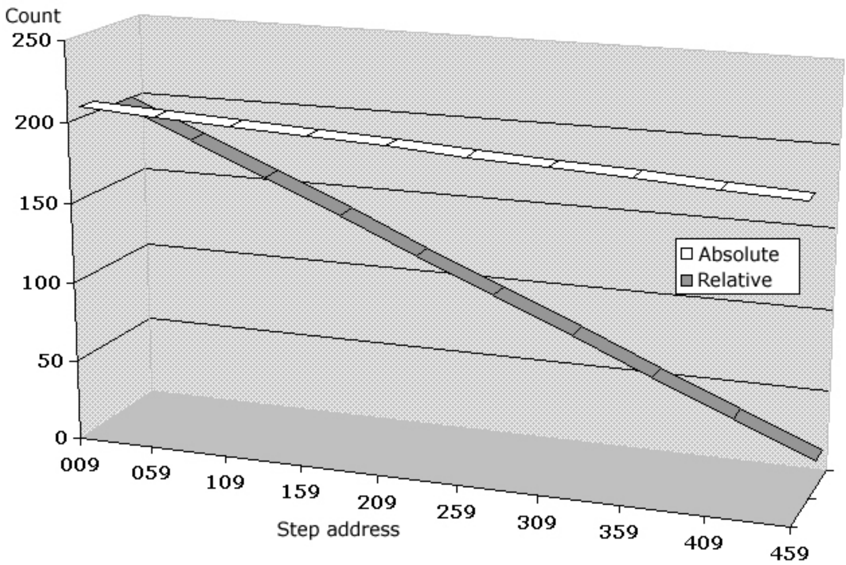
9 2nd Op 1 4 gives 200,…

5 9 2nd Op 1 4 gives 180,…

1 0 9 2nd Op 1 4 gives 160,…

and so on until 459.

We obtain the following data :



This graph confirms the performance of the absolute addressing.

Data

Data structure

Data are displayed on 10 digits with eventually the minus sign.

In the case of display in scientific notation (

)the mantissa is shown on 8 digits and the exponent on 2 digits with possible display of minus signs (mantissa and/or exponent).

In every case, the internal management of the registers stays the same: the mantissa on 13 characters, the exponent on 2 characters and 1 character to express the signs.
Let be a total of 16 characters (or 2 bytes).

Value of sign	Signs	
	Mantissa	Exponent
0	+	+
2	-	+
4	+	-
6	-	-

Data analysis

The memory of the calculator is shared between the program and the data. This partioning is modifiable (2nd Op 1 7) to distribute the memory between the program steps and the data registers :

OP 17	TI58/TI58C
0	**479-00**
1	**399-09**
2	**319-19**
3	**239-29**
4	**159-39**
5	**079-49**
6	**000-59**

By taking as reference the TI58, we notice that 480 program steps correspond to 60 registers.
A register so takes the place of 8 steps.

We have either 60 registers of 16 characters, or 480 steps of 2 characters : The TI58 thus has 960 characters of usable memory. (1920 for the **TI59**)

In the case of a partition **TI58** "**239**-**29**" (3 2nd Op 1 7) we have 480 available bytes for the program (240 steps) and 480 available bytes for the data (30 registers).

This distribution between program and data authorizes us to make an equivalence between steps and registers (for the **TI58**) :

- the register 00 correspond with steps 479, 478, 477, 476, 475, 474, 473, 472.
- the register 59 correspond with steps 007, 006, 005, 004, 003, 002, 001, 000.
- etc …

	Steps	
Reg	**from**	**To**
00	479	472
01	471	464
02	463	456
03	455	448
04	447	440
05	439	432
06	431	424
07	423	416
08	415	408
09	407	400
10	399	392
11	391	384
12	383	376
13	375	368
14	367	360

	Steps	
Reg	**from**	**to**
15	359	352
16	351	344
17	343	336
18	335	328
19	327	320
20	319	312
21	311	304
22	303	296
23	295	288
24	287	280
25	279	272
26	271	264
27	263	256
28	255	248
29	247	240

	Steps	
Reg	**from**	**to**
30	239	232
31	231	224
32	223	216
33	215	208
34	207	200
35	199	192
36	191	184
37	183	176
38	175	168
39	167	160
40	159	152
41	151	144
42	143	136
43	135	128
44	127	120

	Steps	
Reg	**from**	**to**
45	119	112
46	111	104
47	103	096
48	095	088
49	087	080
50	079	072
51	071	064
52	063	056
53	055	048
54	047	040
55	039	032
56	031	024
57	023	016
58	015	008
59	007	000

We can verify by the practice this logic of correspondence.

In “calculator mode”, we enter :

Keys	Display	
4 2nd Op 1 7	159.39	changes partition
2nd π	3.14159265	PI
STO 3 0		stores in register 30
3 2nd Op 1 7	239.29	changes partition
GTO 2 3 9 LRN	239 31	Programming mode

We will analyze steps, backwards :

The display gives us **239 31** then …

- BST gives **238 41**
- BST gives **237 59**
- BST gives **236 26**
- BST gives **235 53**
- BST gives **234 59**
- BST gives **233 00**
- BST gives **232 00**

Thus :

<table>
<tr><td></td><td colspan="13">Mantissa</td><td colspan="2">Exp.</td><td>S.</td></tr>
<tr><td>Reg. 30</td><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>9</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Steps</td><td colspan="2">239</td><td colspan="2">238</td><td colspan="2">237</td><td colspan="2">236</td><td colspan="2">235</td><td colspan="2">234</td><td colspan="2">233</td><td colspan="2">232</td></tr>
</table>

Internal registers

The internal registers, manipulable with the hidden instruction **HIR**, , are used by the AOS pile, the functioning of which is necessary to understand to avoid the conflicts between a personal use of these registers and a management made by the calculator of these same registers.

The following operation uses all the pile, thus all the internal registers :

2 x (8 - (90 / (3 * (9 - (1 + (45 / (3 * 5))))))) =

An analysis of these registers by means of a program (see following page) gives us :

```
 2.   HIR11
 8.   HIR12
90.   HIR13
 3.   HIR14
 9.   HIR15
 1.   HIR16
45.   HIR17
 3.   HIR18
```

Let be all the operands entered until find the first closing parenthesis.

000	76	LBL
001	11	A
002	02	2
003	65	*
004	53	(
005	08	8
006	75	–
007	53	(
008	09	9
009	00	0
010	55	/
011	53	(
012	03	3
013	65	*
014	53	(
015	09	9
016	75	–
017	53	(
018	01	1
019	85	+
020	53	(
021	04	4
022	05	5
023	55	/
024	53	(
025	03	3
026	65	*
027	05	5
028	54	)
029	54	)
030	54	)
031	54	)
032	54	)
033	54	)
034	54	)
035	95	=
036	91	R/S
037	76	LBL
038	12	B
039	82	HIR
040	11	11

041	42	STO
042	01	01
043	82	HIR
044	12	12
045	42	STO
046	02	02
047	82	HIR
048	13	13
049	42	STO
050	03	03
051	82	HIR
052	14	14
053	42	STO
054	04	04
055	82	HIR
056	15	15
057	42	STO
058	05	05
059	82	HIR
060	16	16
061	42	STO
062	06	06
063	82	HIR
064	17	17
065	42	STO
066	07	07
067	82	HIR
068	18	18
069	42	STO
070	08	08
071	71	SBR
072	69	OP
073	00	0
074	02	2
075	69	OP
076	04	04
077	43	RCL
078	01	01
079	69	OP
080	06	06
081	71	SBR

082	69	OP
083	00	0
084	03	3
085	69	OP
086	04	04
087	43	RCL
088	02	02
089	69	OP
090	06	06
091	71	SBR
092	69	OP
093	00	0
094	04	4
095	69	OP
096	04	04
097	43	RCL
098	03	03
099	69	OP
100	06	06
101	71	SBR
102	69	OP
103	00	0
104	05	5
105	69	OP
106	04	04
107	43	RCL
108	04	04
109	69	OP
110	06	06
111	71	SBR
112	69	OP
113	00	0
114	06	6
115	00	0
116	69	OP
117	04	04
118	43	RCL
119	05	05
120	69	OP
121	06	06
122	71	SBR

123	69	OP
124	00	0
125	07	7
126	69	OP
127	04	04
128	43	RCL
129	06	06
130	69	OP
131	06	06
132	71	SBR
133	69	OP
134	01	1
135	00	0
136	69	OP
137	04	04
138	43	RCL
139	07	07
140	69	OP
141	06	06
142	71	SBR
143	69	OP
144	01	1
145	01	1
146	69	OP
147	04	04
148	43	RCL
149	08	08
150	69	OP
151	06	06
152	91	R/S
153	76	LBL
154	69	OP
155	02	2
156	03	3
157	02	2
158	04	4
159	03	3
160	05	5
161	00	0
162	02	2
163	92	RTN

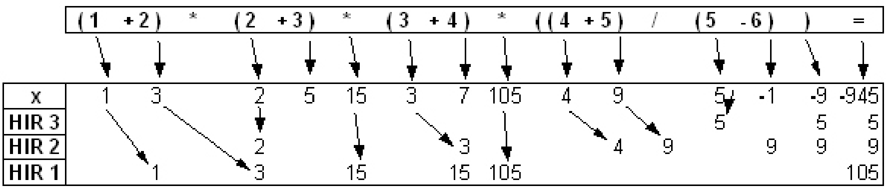
The AOS pile works in the following way :

• A number, followed by an operator, stores the register **x** (previous number or intermediate result) in the register **HIR** of rank ***r***,

• An operator, or an opening parenthesis, adds 1 to the rank ***r***,

• A closing parenthesis executes the last operator between the register **x** and the register **HIR** of rank ***r***, puts the result in the register **x**, then subtracts 1 to the rank ***r***.

<u>Example :</u>



Any intermediate result appears in the display before being put in reserve in the AOS pile.

The internal registers **HIR** are also used by the functions of alphanumeric printing.

If, in “calculator mode”, we enter :

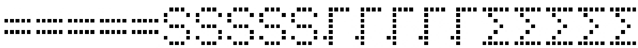
6 4 6 4 6 4 6 4 6 4 **OP 01**

3 6 3 6 3 6 3 6 3 6 **OP 02**

5 2 5 2 5 2 5 2 5 2 **OP 03**

7 7 7 7 7 7 7 7 7 7 **OP 04**

OP 05 prints :



We notice the contents of the internal registers 5 to 8 :

- **HIR 15** gives .0064646465 (6464646464000034 in internal)
- **HIR 16** gives .0036363636 (3636363636000034 in internal)
- **HIR 17** gives .0052525253 (5252525252000034 in internal)
- **HIR 18** gives .0077777778 (7777777777000034 in internal)

That's why the program of the page 106 collects the registers **HIR** to store them in the registers of data 00 - 08 before using the functions of printing.

How to practise ?

The calculators **TI59**/**58**/**58C** necessary for the practice of the language LMS are not any more marketed for several years.
Although it is sometimes possible to find a second-hand TI during second-hand trades or on web auction sites, these opportunities are rather rare and the state of machines so found is not really guaranteed, the keyboard tending to "bounce" and batteries being often defective.
Fortunately the passion of "aficionados" continued over the years, and the web offers diverse sites proposing interesting information, some manuals and other documentations, but especially some substitution solutions, emulators which work on PC or tablets (MS Dos, Windows, Android, Pocket PC).

An emulator of TI59/58/58C on platform Windows is proposed on a Web site completely dedicated to these calculators, and it is henceforth possible to devote to the pleasures of this language by downloading this free software on

http://ti58c.ift.cx

This site references most of the available emulators and also gives the links towards the other main sites dedicated to these calculators.

Summary

Index

I

L

N

O

P

R

S

T

W

X

Y