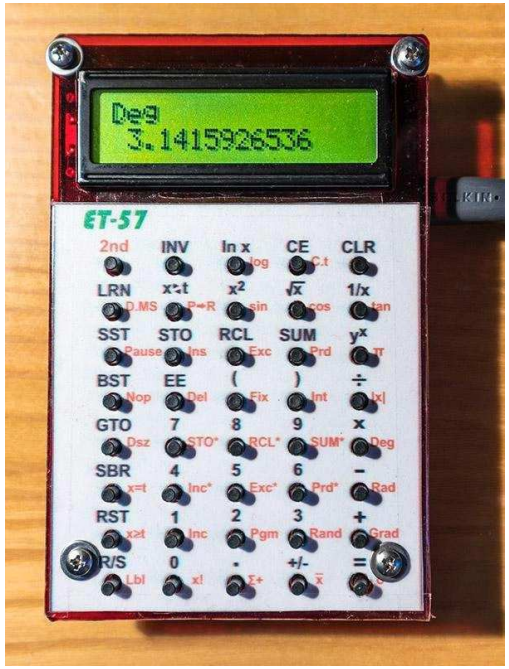
PROGRAMMABLE CALCULATOR

ET-57



User's Guide

ET-57 Programmable Calculator

User's Guide

Calculator version 201101

March 2023

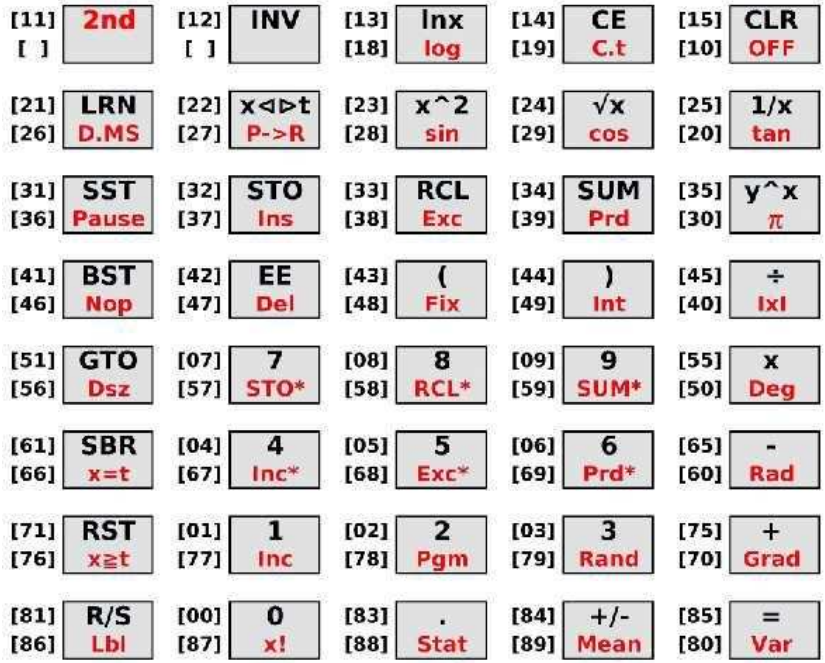
from the original manual by Miroslav Nemecek
translation and adaptation by Pierre Houbert

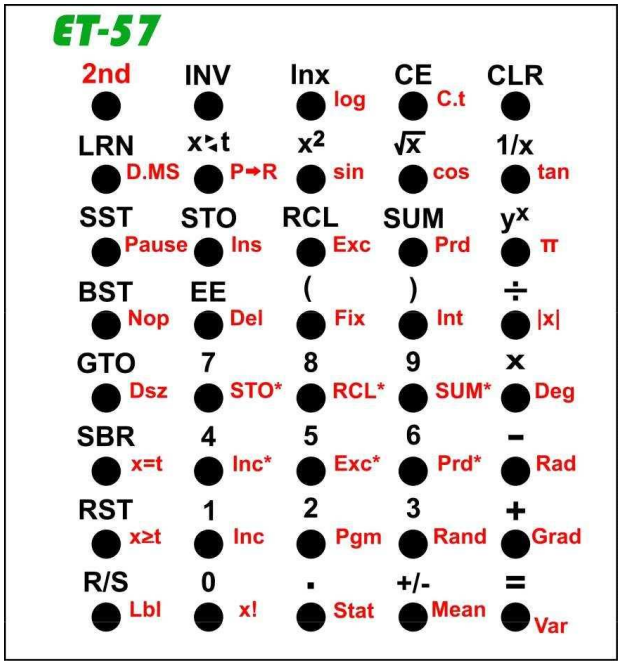
website: http://www.breatharian.eu/hw/et57/

Summary

1. Keyboard layout

For each key, the basic meaning is shown on the first line and the alternative meaning on the second line of the screen (after pressing the 2nd key).





2. Features

Summary : Calculation accuracy of 17 mantissa digits (BCD code), 11 mantissa digits displayed. 80 registers (RAM), 500 program steps (EEPROM), ARmega8 processor, LCD screen.

- ATmega8 processor (8 MHz, 8 KB ROM, 1 KB RAM, 512 B EEPROM)
- Supply voltage 5 V (from USB charger or USB port)
- Calculations in BCD code
- 40 key keyboard
- Two-line LCD display (2 x 16 alphanumeric characters)
- 17-digit calculation accuracy
- Accuracy of base registers 15 digits
- Accuracy of extended registers 13 digits
- Data display to 11 significant digits
- Scientific display mode with exponent on 2 digits, from - 99 to +99
- 10 program spaces
- 50 program steps per space (500 steps total)
- 10 labels per program space
- Function calls and jumps between program spaces
- User program stored in EEPROM (without battery)
- 10 base registers (accessible by direct addressing)
- 70 extended registers (accessible by indirect addressing)
- Indexing access to variables
- Exponential and logarithmic functions
- Trigonometric functions
- Statistical functions
- Factorial
- Random number generator
- Fully open source hardware and software
- Controlling an external device via the ISP connector
- Calculator code entirely written in AVR assembler

3. Description

The **ET-57** calculator is conceptually based on the famous **TI-57** calculator, developed in 1977 by Texas Instruments.
It tries to maintain compatibility with programs written for the **TI-57** while extending functionality by utilizing the capabilities of the processor being used.
The functionality is extended by more program spaces (10 spaces for a total of 500 steps), more data registers (80), direct and indirect addressing, factorial, random number generator...

The calculator is intended for former users of the **TI-57** calculator, those interested in retro technology, and as a teaching aid to familiarize themselves with the principles of calculator use and programming.
For this reason, they strive to simplify the design as much as possible, consisting only of microswitches, a processor, an LCD screen, a connector for the external power supply and a few small components.
The **ET-57** is a tool made for experimenting, teaching, or being used in the office with external power from a USB charger or USB port.

In addition to the basic **ET-57** variant with an ATmega8 processor, the calculator can also be available in the hardware variant of the **ET-58** calculator, with ATmega88, ATmega168 or ATmega328 processors.
This **ET-57B** variant differs from the base variant in that it allows the calculator to be turned off (OFF button) and LCD screen contrast control, but does not provide access to external devices (the **ET-58** calculator does not include an ISP connector).

4. How to use the calculator

The **ET-57** calculator is equipped with a 2-line alphanumeric LCD display, 40 microswitches and a processor.

Since the calculator is not battery powered (it is intended for desktop use with external power from a USB charger or USB port), it does not include a power switch. feed.
The user program is stored in the EEPROM memory, the content of which is retained even without power supply.
By disconnecting the power supply, the calculator is reset, the registers, the contents of the display and the operations started are erased, only the contents of the user program (in the EEPROM) remain.

After connecting the power, the calculator name will be displayed on the calculator screen for 1 second, along with a 6-digit code representing the date of the calculator's firmware version.
For example. "ET-57 201101" means firmware date (build) 11/1/2020.

The calculator in the ET-57B variant (reprogrammed **ET-58** calculator processor) can be turned off by pressing **2nd CLR** (function **OFF**) and turned on by pressing **CLR**.
By pressing **INV 2nd CLR** (LCD function) followed by a number from **0** to **9**, the contrast of the display can be controlled.

5. Deviations from the **TI-57**

Although the **ET-57** calculator software strives for maximum compatibility with the original **TI-57** calculator, deviations may occur and some programs may need to be modified upon import.
Here are the known discrepancies that may need to be taken into account :

Greater accuracy

The original **TI-57** calculator operates with an internal precision of 11 digits (11 BCD digits of the mantissa, 2 digits of the exponent, 1 digit of the sign, 7 bytes in total) and displays a maximum of 8 digits of the mantissa.
The **ET-57** calculator calculates internally with a mantissa precision of 17 digits (10 bytes per number).
It stores the result of the calculation in base registers (i.e. registers **R0**..**R9** and **X**) with a precision of 15 digits (9 bytes). When stored in an extended register (**R10**..**R79**), data is rounded to 13 digits (8 bytes).

The original **TI-57** calculator uses the CORDIC method for function calculations, which allows relatively quick and easy calculations using only basic operations (shift, addition, subtraction) and table values.
The CORDIC method is used in calculators and internally in processors.
In contrast, the **ET-57** calculator uses the Taylor series for calculations, which is more suitable for the type of processor used.

As a consequence of the above, the original **TI-57** calculates with a precision of 9 to 10 digits, the additional 1 to 2 digits of the mantissa include calculation inaccuracies. ET-57 calculates functions with a precision of 15 digits (internally it calculates to 17 digits, by storing in the register the result is rounded to 15 valid digits).

Higher accuracy is usually not a problem, it can appear, for example, with a random number generator or when comparing the results of calculators.

Calculation repetition

The **ET-57** calculator, after pressing the [=] key, repeats the last entered arithmetic operation, while on the original **TI-57**, repeatedly pressing the [=] key causes an error indication , which some programs use to indicate an error.
In such cases, it is necessary to provide an error indication in another way, for example with the sequence [CLR] [1/x] .

6. Numbers format

In the original **TI-57** calculator, a number is stored in 14-digit BCD registers D13 to D0. A single digit can take values from 0 to 9. The lower two digits, D1 and D0, contain the unsigned exponent, ranging from 00 to 99. The upper 11 digits, D12 to D2, contain the mantissa digits. The mantissa is always left-aligned so that the D12 digit does not contain zeros. The highest digit D13 contains the sign flags. Bit 0 indicates a negative mantissa, bit 1 a negative exponent, and bit 2 an inverted mantissa (highest digit carry sign, mantissa in negative form). This method of interpreting numbers uses CPU support for BCD operations.

The **ET-57** calculator also uses a BCD interpretation of a number's mantissa. The BCD format provides more appropriate rounding of results for human interpretation. For example. the number 0.1 is stored in the BCD code as the digit '1' with an exponent of -1, with no loss of precision. In binary code, such a number would be expressed with the mantissa 4CCCC... (infinite number of digits), when the simple writing of the number creates a small error.

The mantissa is not stored in the **ET-57** calculator in absolute form with a separate sign (as in the original calculator), but retains the signed form, with an extension to the most significant digit. A signed digit contains the value 0 (indicating a non-negative number) or 9 (indicating a negative number). The negation of a
number means the "decimal complement" of the digits of the mantissa, or the "nines complement" (= inversion) increased by 1.

The exponent again expresses a decimal exponent, but it is stored in the first byte of the number as a binary number with a bias of 128. An exponent with a value of 0 (order of ones) is symbolized by the binary value 128. An exponent of 1 (tens) has a value of 129, an exponent of -1 (tenths) has a value of 127.
The exponent has a range of valid binary values from 29 to 227, which corresponds to a decimal exponent from -99 to +99.
In addition, 3 special cases of exponent values are used: 0 indicates zero, 28 indicates an overflow towards negative exponents (too small number) and 228 indicates an overflow towards positive exponents (too large number).

The calculator uses 3 number formats, differing in the precision of the mantissa:

1) 10-byte numbers (17-digit precision) are used in calculations. The first byte is the exponent in binary form with a bias of 128. The next 9 bytes contain the mantissa in signed form, from upper digits to lower digits. This means 1 significant digit and 17 significant digits. By displaying the mantissa in HEX form, the figure is displayed in a human-readable form, as numbers from left to right.

The mantissa is normalized so that the first digit (sign) contains 0 (a non-negative number) or 9 (a negative number). The second digit (the highest digit of the mantissa) contains a different number than the signed digit - i.e. the mantissa is left-aligned.

2) The base registers (**R0** to **R9**, as well as the **X** and LAST registers) occupy 9 bytes (15 digit precision). The first byte is the exponent, the next 8 bytes contain 15 mantissa digits and 1 sign digit.

3) The extended registers (R10 to R79) occupy 8 bytes (precision 13 digits), with a mantissa of 7 bytes, i.e. 13 digits of mantissa and 1 digit of sign.

The mantissa of the result can be displayed for debugging purposes with the **INV +** keys (disabled with **INV -**). The 16 digits of the X-register are displayed, without the exponent.

Examples of numbers (in HEX format, including an exponent with a bias of 128) :

3.14159265358979 -> 80 03 14 15 92 65 35 89 79
-3.14159265358979 -> 80 96 85 84 07 34 64 10 21
123.456 -> 82 01 23 45 60 00 00 00 00

A famous trigonometry test can be used to test the accuracy of the calculator :

If the calculation is correct, the result should again be the number 9.
The calculation quickly loses accuracy and discrepancies are common with calculators.
For the original TI-57 calculator, the result is 9.0047464 (3-digit precision), for the ET-57 calculator, the test result is 8.9999999976 (9-digit precision).

More information on the accuracy of calculators :
http://www.datamath.org/Forensics.htm

7. Keyboard

The calculator can work either in direct mode (execution), when the key codes are executed immediately, or in programming mode, when the key codes are only saved in the program, but not executed.

The calculator is controlled by a set of 40 keys arranged in 8 rows and 5 columns.
Lines are numbered from top to bottom, in order from 1 to 8.
The columns are numbered from left to right, with numbers from 1 to 5.
It is with this numbering that the key codes are stored in the program.

After pressing the 2nd key, the alternate function of the next key is used, indicated by the column number 6 to 10 (the number 10 is replaced by the number 0 in the code).

When writing the program to memory (using the LRN key), the code of the pressed key is written to the program as a pair of digits, where the first digit represents the row of the key (usually 1 to 9) and the second digit represents the column of the key (typically 1 to 5 for the basic function or 6 to 0 for an alternative function).

Numerical key codes from 0 to 9 are not stored in the program using the key coordinate, but as a decimal value from 00 to 09.

Note: In the text of the manual, the names of the keys are given without any second prefix which may be necessary to invoke the function of the key. For example. the button code Rand (random number) is called by pressing the 2nd and 3 keys.

8. On-screen indicators

The LCD display contains 2 lines of 16 alphanumeric characters.
The first line is used to display the indicators, the second line to display the number entered and the result of the operation.

Deg/**Rad**/**Grd** :
indication of the angle unit in degrees, radians or gradians.
360° = 2*PI radians = 400 gradians.
The angle measurement unit can be changed with the keys [Deg] , [Rad] or [Grad]

Fix 0 à **Fix 8** :
indicates the selected rounding of numbers from 0 to 8 decimal places. It is set with the keys [Fix] [0] to [Fix] [8]

The sequence [INV] [Fix] ([Fix] [9] also has the same meaning) deactivates the rounding of the displayed result. In this case, the rounding is not indicated on the display (it is replaced by spaces).

EE :
indicates exponent mode.
After pressing [EE] , the number is displayed in scientific, mantissa and exponent form.
The mode can be canceled by pressing [CLR] or [INV] [EE]
If exponent mode is off, nothing is shown on the screen..

2nd :
indicates that you pressed the alternate function key [2nd] .
If a key is pressed after pressing [2nd], its alternate function (displayed in the second row of the keyboard) will be executed instead of its base function.
If the key [2nd] is not pressed, or if it is pressed twice, the alternative function is not active, the basic button function is executed.
The basic state is not indicated on the display (spaces are displayed at the position).

INV :
indicates that you have pressed the key [INV] activating the inversion function.

Opération :
the last position of the 1st line is intended to indicate the active arithmetic operation: + addition, - subtraction, * multiplication, : division, \ modulo...

9. Number editor

The entered number, together with the results of the calculation, will be displayed on the 2nd line of the screen. The mantissa is displayed with a maximum of 11 digits.

1 position is reserved for the sign before the mantissa. A '-' will appear here for negative numbers, a space will be left for positive numbers.

The exponent is displayed after the mantissa (if exponent mode is active). The exponent is separated from the mantissa by a + or - sign. The exponent is displayed as 2 digits.

A decimal point is part of the mantissa. In exponent mode in scientific notation (mantissa and exponent), the decimal point always appears after the first digit. If exponent mode is not active, a decimal point is displayed after the units digit.

The key [CE] deletes the last character of the mantissa or exponent (depending on where the digits are currently written).

The key [EE] starts entering the exponent.
You can return to entering the mantissa by pressing the key [.] (period) or [INV] [EE]

The key [EE] is also used to start editing the displayed result of the operation. This can be used to remove hidden digits from a number.

Example : round a number to 4 decimal places.

[π]		3.1415926536	example of a number with decimals
[Fix]	[4]	3.1416	the display is rounded to 4 decimal places
[EE]		3.1416+00	start editing, cut hidden digits
[INV]	[EE]	3.1416	turn off exponent mode
[INV]	[Fix]	3.1416	disabling the rounding, the result stays at 3.1416

10. Numerical expressions

During calculations, the calculator maintains the priority of operations in 3 steps :

①. ^ power, √‾ square root

②. * multiplication, : division, \ modulo

③. + addition, - substraction

The calculations are first evaluated at the level ① power and square root, then ② multiplication and division, and finally ③ addition and subtraction.

Any number of parentheses can be used in an expression, up to level 7.

After performing the calculation, you can repeat the calculation of the lowest level by pressing the key [=] again.
Enter a number and press [=] to repeat the operation. The entered number is used as the first operand of the operation, the second operand remains original.

Note: Calculations are performed internally with 17-digit precision. By storing the result in the X-register, the result is rounded to 15 digits.

Example:

- 3 + 2 = 5
- 4 =6
- 1 0 =12
- 1 0 + 2 * 3 y^x 4 = 172 [*similar to 10 + (2*(3^4)) = 172*]

11. Registers Addressing

The calculator contains 10 base registers (named **R0** to **R9**) and 70 additional registers (named **R10** to **R79**), for a total of 80 data registers (**R0** to **R79**).

The base registers **R0** to **R9** have a mantissa precision of 15 digits. Base registers are used as main working registers. They are addressed by direct addressing, using the instructions :

[STO] [RCL] [SUM] [Exc] [Prd] [Inc] followed by registry indexes [0] to [9].

The base registers are also addressable by the inverse operations :

[INV] [SUM] , [INV] [Prd] , [INV] [Inc]

The instructions [INV] [STO] , [INV] [RCL] , [INV] [Exc] have the same function as the instructions without [INV] , , but instead of the basic registers **R0** to **R9**, they address the additional registers **R10** to **R19**.

Most basic registers have an additional function:
R0 ... number of elements N (Statistics), loop counter **Dsz**
R1 ... sum of y (Statistics)
R2 ... sum of y^2 (Statistics)
R3 ... sum of x (Statistics)
R4 ... sum of x^2 (Statistics)
R5 ... sum of x*y (Statistics)
R6
R7 ... registrer **T**
R8 ... index register for indirect addressing
R9 ... alternative index register for indirect addressing

The additional data registers **R10** to **R79** have a precision reduced to 13 digits. By using them, the stored data is shortened by 2 digits.

The additional registers (**R10** to **R79**) cannot be addressed by direct addressing, it is necessary to use indirect addressing :

STO* , RCL* , SUM* , Exc* , Prd* et Inc*

In indirect addressing, the register number is not part of the instruction, but is read from register **R8** (indirect addressing register). Inverses are treated the same way

Indirect addressing operations preceded by INV :

The instructions

INV STO* , INV RCL* , INV Exc*

have the same function as the same instructions without **INV**, but, in this case, the alternative index register **R9** is used instead of the index register **R8**.

Note: The base registers **R0** to **R9** can also be addressed by indirect addressing.

Example, reverse order of register contents :

1) Fill registers **R10** to **R79** with numbers 0 to 69

	Pgm 1	switch to program space 1 (and reset pointer)
	LRN	activate programming mode
00	Lbl 1	label of subroutine 1 (filling registers)
01	7 0 STO 0	register counter preparation (70 registers)
04	9 STO 8	prepare index register - 1 (pointer to **R10**-1)
06	CLR	reset **X** register
07	Lbl 9	loop start label
08	Inc 8	increment **R8** (index register)
09	STO*	stores the value of X in the register indexed by **R8**
10	+ 1 =	incrementing the **X**-register
13	Dsz	decrements **R0** and skips the next instruction if **R0**=0
14	GTO 9	continue the loop if **R0** is not yet equal to 0
15	INV SBR	end of program

2) Inversion of register contents (R10<->R79, R11<->R78,...)

16	Lbl 2	label of subroutine 2 (inversion of registers)
17	3 5 STO 0	register counter preparation (70 registers / 2)
20	9 STO 8	preparing the first index register - 1
22	8 0 STO 9	preparing the second index register + 1
25	Lbl 8	loop start label
26	Inc 8	increment **R8** (first index register)
27	INV Inc 9	decrement **R9** (second index register)
28	RCL*	read the value of the register indexed by register **R8**
29	INV Exc*	exchange with the contents of the indexed register **R9**
30	STO*	store the value in the register indexed by **R8**
31	Dsz	decrements **R0** and skips the next instruction if **R0**=0
32	GTO 8	continue the loop if **R0** is not yet equal to 0
33	INV SBR	end of program

3) Display of the contents of registers **R10** to **R79**

34	Lbl 3	label of subroutine 3 (display registers)
35	7 0 STO 0	register counter preparation (70 registers)
38	9 STO 8	prepare index register - 1 (pointer to **R10**-1)
40	Lbl 7	loop start label
41	Inc 8	increment **R8** (first index register)
42	RCL*	read the value of the register indexed by register **R8**
43	Pause	pause to display value
44	Dsz	decrements **R0** and skips the next instruction if **R0**=0
45	GTO 8	continue the loop if **R0** is not yet equal to 0
46	INV SBR	end of program

4) Program testing

LRN	exit programming mode
SBR 1	fill registers **R10** to **R79** with number 0 to 69
INV RCL 9	display contents of register **R19** (9)
SBR 3	check the contents of the registers: the numbers 0 to 69 are displayed
SBR 2	inversion of register contents
INV RCL 9	display contents of register **R19** (60)
SBR 3	check the contents of the registers: the numbers 69 to 0 are displayed

12. Programming

Writing a sequence of keystrokes to program memory is called a program. The program turns the calculator into a powerful tool.
Program memory consists of 10 independent program areas, switched by the instruction Pgm with parameters 0 to 9
Each program area contains 50 program steps, so a total of 500 program steps are available.
10 labels Lbl , numbered from 0 to 9 can be used in each program space.
Subroutines can be called or jumps made between program spaces by using the Pgm instruction in the program, this instruction does not change program space permanently, but only for an GTO or SBR command .

The programs are stored in the processor's EEPROM memory, the contents of which are retained even after the calculator's power supply has been disconnected.

The programming mode is started with the key LRN.
The content of the program is displayed on two lines of the display. The bottom line from the left indicates the current program space (**Pgm0** to **Pgm9**), followed by the current pointer in the program, i.e. address **00** to **49**.
The address is followed by the numeric code of the instruction.
This instruction code consists of 2 digits. The first digit represents the row with the keys 1 to 8, the second digit is the column with the keys 1 to 5 or, for the alternative function, the column 6 to 0. The numeric keys are displayed with the code **00** to **09**.
The instruction code can be followed by a parameter from 0 to 9. The instruction code can also be preceded by the minus sign '**-**' signifying the inverse function INV .
The top line displays the “text” format of the instruction.

Keys useful for programming :

SST	(*Single Step*)	Increments the program pointer by 1 ("next step"). The **SST** key can also be used in normal mode (run). In this mode, after pressing **SST**, the code of the current instruction is executed and the program pointer is positioned on the next step.
BST	(*Back Step*)	Decrement the program pointer by 1 ("previous step").
Ins	(*Insert*)	Inserts an empty instruction (**Nop**) at the current position of the program and shifts the rest of the program.
Del	(*Delete*)	Deletes the instruction at the current position of the program and “moves up” the next part of the program.
LRN	(*Learn*)	Exits program edit mode and returns the calculator to run mode.
GTO	(*Go To*)	The GTO instruction is normally used to move the program pointer to the specified label from 0 to 9. In the “programming” mode, the instruction cannot be used to move the pointer, because the corresponding code would be stored in the program. He must therefore exit programming mode by pressing LRN , make a jump GTO to the specified label 0 to 9 and return to programming mode by pressing LRN. In addition to jumping to a label, **GTO** allows you to jump to an absolute address in the program. The jump is made by pressing INV GTO, followed by the 2 digits of the address **00** to **49**. The jump to absolute address instruction cannot be stored in the program, it is only used to move the pointer during programming and can therefore be entered at the both in execution and in programming. ***Note**: Jumping to an absolute address is done in a different way (with a different key sequence) on the **ET-57** and the original **TI-57**. On the original **TI-57**, the **GTO** key was pressed first, then the **INV** key, and finally the 2-digit address. For the **ET-57**, you must first press **INV**, then **GTO** and finally the 2 digits of the address.*
RST	(*Reset*)	Similar to **GTO**, cannot be used directly in program mode, but can be used in run mode to return the program pointer to address 00 in the current program space.
R/S	(**Run**/*Stop*)	Starts the program or stops the program (used in run mode).

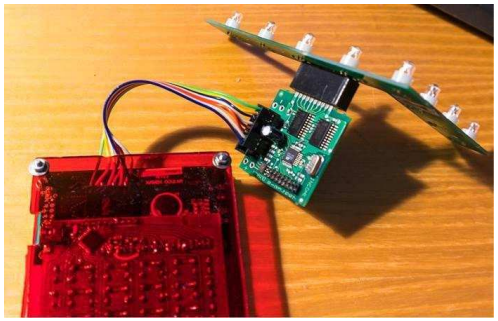
13. External Devices and Ports

The **ET-57** calculator allows connection of an external device using the ISP connector, which is otherwise used to program the calculator's processor. Communication takes place via the SPI protocol. The ISP connector is an 8-pin KONPC-SPK-8 connector with the following pin assignment:

- 1 **SCK** (serial clock, transmitted by computer)
- 2 **MISO** (calculator data input, device data output)
- 3 **MOSI** (calculator data output, in-device data input)
- 4 orientation key, pin missing, blinded in connector so it can't be inserted
- 5 /**RESET** (processor reset during ISP programming)
- 6 **GND** (ground, 0V)
- 7 **VCC** (power supply, +5V)
- 8 unused, connected to +5V in calculator, but in future it may not be used or may be used for SS signal

The connection cable connects the same pins both in the calculator and in the device (no signal crossing).
Note: The SS signal on the slave side must be connected to GND.

The device can be controlled using a 256-byte addressable port network. A negative number -1 to -256 representing the port address 0 to 255 is stored in the **R8** register of the computer. The **STO*** instruction sends the number 0 to 255 to the selected port. Conversely, the **RCL*** instruction reads the value of the selected port as a number between 0 and 255.



Similarly, the **INV STO*** and **INV RCL*** instructions can be used, in which the **R9** register with the port address is used instead of the **R8** register. Instructions other than **STO*** and **RCL*** do not allow access to device ports.

If the device is not connected or in another case of a communication error, when using the **STO*** and **RCL*** instructions, the program will stop and indicate an error (flashing 'E'). For **INV STO*** and **INV RCL*** instructions, the communication error is ignored, the program does not indicate the error and continues to run. Typically, when debugging the program, the **STO*** and **RCL*** instructions are used first to ensure that communication problems are reported. After debugging the program, instructions with INV are used to ensure that the program will work without interruption for a long time, even in the event of short-term equipment failures.

Communication takes place using the SPI communication protocol, with a clock frequency of 250 kHz. The ECU acts as a master (control unit), the external device is a slave (subordinate unit). The master (computer) sends serial data on the MOSI line, from the high bit to the low bit. It sends a clock signal to the SCK line, with data sampling on the rising edge.

At the same time, the slave (device) returns data on the MISO line. The transfer of one byte (8 bits) takes 32 µs. The master adds a 10 us delay after sending each byte so that the slave has time to evaluate the received byte.

The synchronization of the communication between master and slave is not controlled by the SS signal, but by software, by resetting the reception on the slave side. At the start of each transmission, the master sends synchronization byte 0x53 (the letter 'S', at bit level 01010011b). If the transfer is not synchronized, the data on the slave side is shifted and the slave receives a value other than 0x53. In such a case, the slave sends byte 0x55 as an indication of a synchronization error, resets the connection and after a delay of 50 us initiates a new connection. If the slave receives the correct synchronization byte 0x53, it responds with the same value of 0x53 and continues communication.

The master sends synchronization bytes 0x53 several times. If it receives a correct 0x53 response from the slave, it continues communication. If it receives a byte of 0xff or 0x00, it treats it as a device not connected flag and aborts communication with an error indication. If there are other responses, it adds a delay of 100 us and retries the synchronization attempt.

After a successful synchronization, the master continues by sending the command 0x52 (letter 'R') to read the port from the device or 0x57 (letter 'W') to write the port to the device. After the command, the master sends a byte with the port address from 0 to 255. After sending the address, the master sends the third byte of the command - data. In the case of a write to the port, it will send data 0 to 255 to write to the port. In the case of a read from the port, the master sends a 0xFF byte, while the slave sends a byte, it responds with a byte of the data read from the port.

In all other cases, when the slave byte is not specified, the slave responds by repeating the byte received from the master (echo).

Example of communication (master/slave) :

- Master writes byte 0x24 to port 0x01

0x53 / 0x ?? ...master sends 5x SYNC commands for synchronization, response from slave is initially undefined, it is ignored
0x53 / 0x53 ...the slave responds correctly with 0x53, but the master ignores the response for now, as it may be the rest of the previous communication
0x53 / 0x53
0x53 / 0x53
0x53 / 0x53 ...master detects correct response 0x53, continue
0x57 / 0x53 ...master sends command to write 0x57, slave keeps echoing
0x01 / 0x57 ... the master sends port address 0x01, the slave has detected a write command, sends the echo of the previous byte and reads the address
0x24 / 0x01 ...the master sends data 0x24 to be written to the port

- Master rereads data from port 0x01

0x53 / 0x01 ... master sends 5x SYNC commands again for synchronization
0x53 0x53 0x53 0x53 / 0x53 0x53 0x53 0x53 ... connected slave
0x52 / 0x53 ... the master sends a command to read 0x52
0x01 / 0x52 ... master sends port address 0x01, slave prepares port data for sending
0xFF / 0x24 ... master sends 0xFF and reads slave data 0x24

Example source code to handle Slave communication :

```
u8 SPIData[256];        // SPI slave data
u8 SPIInx;      // index of SPI command
u8 SPICmd;      // 1st byte - SPI command
u8 SPIAddr;     // 2nd byte - SPI address

#define SPICMD_SYNC         0x53    // synchro command ('S')
#define SPICMD_READ         0x52    // read command ('R')
#define SPICMD_WRITE        0x57    // write command ('W')
#define SPICMD_ERR          0x55    // invalid synchronization ('U')

ISR(SPI_STC_vect)     // SPI interrupt
{
        u8 d = SPDR;   // read data byte
        if (SPIInx == 0) // receive command
        {
                // check known commands
                if (    (d != SPICMD_SYNC) &&
                        (d != SPICMD_READ) &&
                        (d != SPICMD_WRITE))
                {
                        // error, try to re-synchronize
                        SPDR = SPICMD_ERR; // report error to master
                        SPI_Term(); // terminate SPI
                        _delay_us(50); // short delay 50 us
                        SPI_SlaveInit(); // re-initialize SPI
                        return;
                }

                // shift to next byte of the command
                if (d != SPICMD_SYNC)
                {
                        SPICmd = d;
                        SPIInx = 1; // next index = address
                }
        }
        else if (SPIInx == 1)     // receive address
        {
                SPIAddr = d; // save address
                SPIInx = 2; // next index = data

                // read data
                if (SPICmd == SPICMD_READ) SPDR = SPIData[d];
        }
        else    // receive data
        {
                // next index = command
                SPIInx = 0;

                // write data
                if (SPICmd == SPICMD_WRITE) SPIData[SPIAddr] = d;
        }
}
```

14. Keys and instructions

Each instruction has a BCD program code, a title and a sequence of presses on one or more keys to express it.

00...09 [0] [9] 0...9 - Basic digits

Base digits are used to enter digits in the range 0 to 9. They are used to enter the mantissa of a number, enter the exponent, memory register number, tag number, and others.

The numbers are stored in the program with the code 00 to 09.

10 [2nd] [CLR] OFF - Stop calculator (ET-57B)

The **2nd CLR** (**OFF**) sequence is used to turn off the calculator.
The calculator can be turned back on by pressing the **CLR** key alone.

By specifying the **INV** prefix before the **OFF** statement, the contrast of the LCD display can be adjusted.
The instruction requires a numeric code from 0 to 9 as a parameter.
0 sets the lowest display contrast (light font on light background), 9 sets the highest display contrast (dark font on dark background).

The **OFF** and **INV OFF** functions are only available for the **ET-57B** variant of the calculator.
The **ET-57** calculator cannot be turned off with a button, nor can the contrast of the display be controlled.

11 [2nd] 2nd - Alternate function

The **2nd key** is used to change the meaning of the next key to an alternate function.
After pressing **2nd**, the alternate function of the next button is then executed.
A second press on **2nd** returns to the basic functions. (cancels the first press on **2nd**).

The code for the **2nd key** (11) is not saved in the program, it is the alternative code for the next key which is then saved.

Example:
2 lnx ... calculate the natural logarithm of the number 2 [0.6931...]
2 2nd lnx ... decimal logarithm of the number 2 (**log** instruction) [0.3010...]

12 [INV] INV - Inverse of a function

The **INV** key, pressed before another key, will cause that other key to have its function reversed.
In some special cases, **INV** will not cause the inverse of the following function but an additional alternative function.
A second press on **INV** returns to the basic functions. (cancels the first press on **INV**).
The **INV** button code is not saved in the program. It is stored as an instruction indicator with the minus sign (**-**) before the instruction code.
Some instructions do not accept the **INV** prefix, which is then ignored and the instruction code is stored without the prefix in the program.

Example:
1 0 sin ... calculate the sine of 10 [0,1736...]
0 . 1 2 INV sin ... calculates the arcsine of 0.12 [0.3464...]

13 [lnx] lnx - Natural logarithm and exponent

lnx calculates the natural logarithm of the displayed number. This natural logarithm uses Euler's constant as its base with the value 2.718281828459. If the **INV** button is pressed first, the inverse function, the natural exponent, is executed.
The argument of the **lnx** function must be a non-zero positive number. In the case of zero, the display will flash with the value -9.9999+99, as an error indication.
For a negative number, the absolute value of the number is calculated and the display flashes again with an error indication.

The argument to the **INV lnx** function can be a positive or negative number, ranging from approximately -227 to +227. A number outside this range will result in data overflow and indicate an error.

Example:
5 lnx ... calculates the natural logarithm of 5 [1.6094...]
5 INV lnx ... calculates the natural exponent of 5 [148.413...]

14 [CE] CE - Error correction

CE can be used to cancel the '**E**' error indication, manifested by flashing of the display.
When entering a number, the last character entered is deleted by pressing the **CE** key.

In scientific notation, if the mantissa is being entered, the last character of the mantissa is removed. If the exponent is being entered, the last character of the exponent is deleted. If an exponent with a value of 0 is removed, the exponent is canceled and the input reverts to the mantissa.

15 [CLR] CLR - Clearing the display

CLR performs several initialization operations.
It resets started arithmetic operations, resets error indication,
turns off **EE** exponent mode, resets the **X**-register, and starts editing a new number with a default value of 0.
The **CLR** key does not reset the **T** register or the data registers.
In the **ET-57B** calculator variant, the **CLR** key is used to turn the calculator on (or off).

18 [2nd] [lnx] log - Decimal logarithm and exponent

log calculates the decimal logarithm of the displayed number.
The decimal logarithm uses the number 10 as its base. If the **INV** key is pressed first, the inverse function, the decimal exponent, is executed.
The argument to the log function must be a non-zero positive number. In the case of zero, the display will flash with the value -9.9999+99, as an error indication.
For a negative number, the absolute value of the number is calculated and the display flashes again with an error indication.
The argument of the **INV log** function can be both a positive number and a negative number, ranging from -99 to +99. A number outside this range will result in data overflow and an error indication.

Example:
5 log ... calculates the decimal logarithm of 5 [.69897...]
5 INV log ... calculates the decimal exponent of 5 [100000]

19 [2nd] [CE] C.t - Clear register T

The **C.t** key can be used to clear the **T** register (i.e. **R7** register).

> Entering the **INV** prefix before the key **C.t** clears all registers **R0** to **R79** (including the **T** register).

20 [2nd] [1/x] tan - Tangent

The **tan** function calculates the tangent of an angle. The angle is entered in the units defined by the **Deg**, **Rad** or **Grad** switches.
Typing the **INV** prefix before the **tan** statement will perform the opposite function (arctangent).
The result is an angle in the currently defined angular measurement.

Example:
5 0 tan ... calculates the tangent of 50 [1.19175...]
5 0 INV tan ... calculates the arctangent of 50 [88.85423...]

21 [LRN] LRN - Programming

LRN enables or disables programming mode.

22 [x<>t] x<>t - Exchange of X and T registers

With the **x<>t** key, it is possible to switch between the **X** and **T** registers.
The **X**-register is the working register and also the display content.
Register **T** is an auxiliary (temporary) register, corresponds to register **R07**. It is used to compare numbers and to convert polar and Cartesian coordinates.
The **X** register is reset by the **CLR** key.
The **T** register is reset by the **C.t** key.

23 [x²] x^2 - Square of a number

The **x^2** function calculates the square of a number, or the multiple of a number by itself.

24 [√x] √x - Square root of a number

√x calculates the square root of a number. The number must not be negative. If a negative number is calculated, the square root of the absolute value of the number is calculated and the error indication '**E**' is activated (display flashes).

25 [1/x] 1/x - Inverse of a number

The **1/x** function calculates the inverse of a number.
If the number is zero, the value 9.9999+99 is displayed and the error indication '**E**' is activated (display flashes).
This application of **0 1/x** is often used in programs to activate error indication and to signal abnormal operation of the program.

26 [2nd] [LRN] D.MS - Conversion of minutes and seconds

The **D.MS** instruction is used to convert time expressed in hours/mins/seconds or angle expressed in degrees/minutes/seconds to a decimal number. The original number ***HH.MMSS*** or ***DD.MMSS***, is entered with the number of hours, or degrees, in the integer position, then with the number of minutes in the first two decimal places and finally with the number of seconds in the next two decimal places.
The result of the function is a decimal number representing the number of hours, or degrees, expressed as a decimal number ***DD.DDDD***.
By specifying the **INV** prefix before the **D.MS** instruction, the reverse operation is performed - time or angle expressed using a decimal number is converted to hours/ minures/seconds or degrees/minutes/seconds .
The decimal number ***DD.DDDD*** representing hours or degrees will be returned as ***HH.MMSS*** or ***DD.MMSS***, with the number of hours, or degrees, in the integer part, and the number of minutes in the first two decimal places followed by the number of seconds in the next two decimal places. If the result is not an integer number of seconds, the decimal seconds are padded with additional decimal digits.

Example :

1 2 . 3 0 2 3 D.MS ... conversion to decimals [12.50638...]
+ 3 . 4 5 1 2 D.MS ... conversion to decimals [3.75333...]
= INV D.MS ... resulting time [16.1535]
Sum of time:
12 hours, 30 minutes and 23 seconds
+ 3 hours, 45 minutes and 12 seconds
= 16 hours, 15 minutes and 35 seconds

27 [2nd] [x<>t] P->R - Conversion polar / cartesian

P->R converts the coordinates of the polar expression (radius/angle) into Cartesian coordinates (abscissa and ordinate).
Before the operation, the **T** register (i.e. auxiliary register **R7**) contains the radius and the **X**-register (content of the display) contains the angle. The angle is given in the selected angular measurement (**Deg**, **Rad** or **Grad**).
After the operation, the **T** register (auxiliary register **R7**) contains the abscissa X, the **X** register (display content) contains the ordinate Y.
By specifying the **INV** prefix before the **P->R** instruction, the inverse operation is performed, the conversion from Cartesian coordinates to polar.
Before the operation, the **T** register contains the abscissa X, the **X** register (display) contains the ordinate Y.
After the operation, the **T** register contains the radius and the **X**-register (display) contains the angle. The angle is given in the selected angular measurement.

Example :

1 0 x<>t ... enter radius 10 in register T
3 0 ... entry of the 30° angle in the X-register
P->R ... conversion from polar to Cartesian coordinates. Y=5
x<>t ... exchange X and T displays coordinates X = 8.6602...

8.6602... **x<>t** ... X abscissa input
5 ... Y ordinate input
INV P->R ... Cartesian to polar conversion, angle = 30°
x<>t ... swap **X** and **T**, show radius 10

28 [2nd] [x²] sin - Sine

The **sin** function calculates the sine of an angle. The angle is entered in the units defined by the **Deg**, **Rad** or **Grad** switches.
Typing the **INV** prefix before the **sin** statement will perform the opposite arcsine function. The result is an angle in the defined angular measurement.
The angle calculated by the arcsine function is between -90° and +90°. The input value of the arcsine function must be between -1 and +1. If it is outside the specified range, the display is limited to the valid range and an "**E**" error is indicated (display flashes).

29 [2nd] [√x] cos - Cosine

The **cos** function calculates the cosine of an angle. The angle is entered in the defined unit **Deg**, **Rad** or **Grad**.
Typing the **INV** prefix before the cos statement performs the opposite arccosine function. The result is an angle in the defined angular measurement.
The angle calculated by the arccosine function is between 0° and +180°. The input value of the arccosine function must be between -1 and +1. If it is outside the specified range, the display is limited to the valid range and an "**E**" error is indicated (display flashes).

30 [2nd] [y^x] PI - Ludolf's number

The **pi** key is used to enter the constant "Ludolf's number", which has a value of 3.14159265358979.

31 [SST] SST - Advance one step in program

The **SST** (Single Step) key increments the program address pointer by 1 in programming mode.
In run mode, the program statement, on which the pointer is positioned, is executed, allowing the program to be run step by step for debugging purposes.
Caution : in this case of step by step test of the program if a subroutine is called, the return of the subroutine cannot occur correctly (the calculator does not memorize the return address of the subroutine).

32 [STO] STO - Store a number in a register

STO (Store) stores the displayed number in data register **R0** to **R9**. A register number from 0 to 9 is entered as an instruction parameter.
The **INV** prefix before the **STO** instruction performs a similar function, but instead of registers **R0** through **R9**, the number is stored in registers **R10** through **R19**. Registers **R10** to **R19** belong to the group of extended registers with a precision reduced to 13 digits. Recording deletes the last 2 digits of the mantissa.

33 [RCL] RCL - Recall a number from a register

RCL (Recall) is used to recall a number from data register **R0** to **R9** to the display. A register number from 0 to 9 is entered as an instruction parameter.
Placing the **INV** prefix before the **RCL** instruction performs a similar function, but instead of the **R0** to **R9** register, the number is read from the **R10** to **R19** register. Registers **R10** to **R19** belong to the group of extended registers with a precision reduced to 13 digits. The mantissa is completed by 2 digits 0 at the end.

34 [SUM] SUM - Add and subtract into a register

SUM adds the displayed number (**X** register) to register **R0** to **R9**. A register number from 0 to 9 is entered as an instruction parameter.
By specifying the **INV** prefix before the **SUM** instruction, the opposite function is performed - subtracting a number from the data register **R0** to **R9**.

35 [y^x] y^x - Power and root

The instruction **y^x** raises the number Y (the first operand, in the stack of operations) to the power expressed by another number X (the second operand, the number on the display).
If the **INV** prefix is pressed first, the inverse operation, the Xth root, is performed. The first operand of Y must be a non-negative number. If this is the lowest level of the expression, the calculation can be repeated for another first operand by pressing **=** repeatedly.

Example:
3 y^x 7 ... raise 3 to power 7 [2187]
2187 INV y^x 7 ... 7th root of 2187 (2187^(1/7)) [3]

36 [2nd] [SST] Pause - Delay

The **Pause** command stops program execution for 0.25 seconds and displays the contents of the register **X**.

37 [2nd] [STO] Ins - Inserting an empty step in the program

The **Ins** key, used in programming mode, inserts an empty **Nop** instruction at the program pointer position. The following steps are shifted downwards.

38 [2nd] [RCL] Exc - Exchange X with a register

Exc exchanges the displayed number (**X** register) with the contents of data register **R0** to **R9**. A register number from 0 to 9 is entered as an instruction parameter.
Placing the **INV** prefix before the **Exc** instruction performs a similar function, but instead of using registers **R0** through **R9**, the number is swapped with registers **R10** through **R19**. Registers **R10** to **R19** belong to the group of extended registers with a precision reduced to 13 digits. When saving, the last 2 digits are removed from the mantissa, and when loading, the mantissa is completed with 2 digits 0 at the end.

39 [2nd] [SUM] Prd - Multiply and Divide in a register

Prd is used to multiply the data registers **R0** to **R9** by the number displayed (register **X**). A register number from 0 to 9 is entered as an instruction parameter.

By entering the **INV** prefix before the **Prd** instruction, the inverse function is performed by dividing the data register **R0** to **R9** by the displayed number.

40 [2nd] [÷] IxI - Absolute value

The **IxI** function adjusts the number to the absolute value (removes the negative sign from the number).

If the **INV** prefix is specified before pressing **IxI**, the absolute value is not applied but a sign test is performed on the number. If this number is less than 0, the result of the operation is -1, if this number is greater than 0, the result is +1. If the number is 0, 0 remains.

41 [BST] BST - Back step in program

The **BST** (*Back Step*) key in programming mode decrements the program address pointer by 1.

42 [EE] EE - Exponent mode

Press **EE** to activate exponent mode.

If the key is pressed while entering a number, it switches to entering the exponent.

At the same time, the display mode in scientific notation with an exponent is activated.

If the key is pressed outside the entry of a number, the display mode in scientific notation with an exponent is activated and the entry of the exponent of the number is launched. This feature is often used to remove hidden digits from a number, because when you start typing, only the displayed digits are written to the display, not the full exact value of the number.

Press the **INV** prefix before pressing **EE** to exit exponent display mode. Another way to exit exponent display mode is to press the **CLR** key.

Example:

pi Fix 4 EE INV EE INV Fix ... round PI number to 4 decimal places [3.1416]

43 [(] (- Left parenthesis

The character **(** opens the calculation of part of the expression.

Parentheses can be used up to 7 levels.

44 [)]) - Right parenthesis

The character **)** closes the calculation of part of the expression.

45 [÷] : - Divide

The sign **:** divides the first operand by the second operand. If this is the lowest level of the expression, the calculation can be repeated for another first operand by pressing **=** repeatedly.

Pressing the **INV** prefix before pressing **:** executes the inverse function, i.e. the modulo **mod** operation which returns the remainder after division. The modulo operation divides the first operand Y (in the stack) by the second operand X (on the display), converts the result to an integer, multiplies the second operand X by it, and subtracts from the first operand Y. The result is the remainder after division. The result has the same sign as the first operand.

Example :

2 . 2 : 0. 5 = ... divide 2.2 by 0.5 [4.4]

2 . 2 INV : 0 . 5 = ... calculates the remainder of 2,2 divided by 0,5 [0,2]

2 . 2 +/- INV : 0 . 5 = ... calculates the remainder of -2,2 divided by 0,5 [-0,2]

2 . 2 INV : 0 . 5 +/- = ... calculates the remainder of 2,2 divided by -0,5 [0,2]

2 . 2 +/- INV : 0 . 5 +/- = ... calculates the remainder of -2,2 divided by -0,5 [-0,2]

46 [2nd] [BST] Nop - No operation

The **Nop** (*No Operation*) command is an empty command that does not perform any operation. It is only used to fill an unused step in the program.

47 [2nd] [EE] Del - Delete a program step

The **Del** (*Delete*) key, used in programming mode, deletes a step from the current program position. The following steps are then shifted upwards..

48 [2nd] [(] Fix - Number of decimal places

Using the **Fix** key, the number displayed on the screen is rounded to the number of decimal places

specified. The number 0 to 8 is entered as a parameter, representing the number of decimals after the decimal point 0 to 8.

In rounding mode, the number is padded from the right with zeros, up to the specified number of decimal places. Entering the sequence **INV Fix** or **Fix 9** disables rounding. In this case, the number is displayed with full precision and trailing leading zeros are suppressed.

Rounding only affects the display of the number. Internally, the number (**X** register) continues to be stored in full. If it is necessary to actually remove the hidden digits, this can be done using the **EE** key (see 42 Exponent mode, **EE**).

The rounding mode set also affects how very small numbers are displayed. If rounding is on and exponent mode is not on, the screen displays zeros for small numbers, even though the valid digits have passed the right edge of the screen. If rounding is not enabled, the calculator will switch to displaying the exponent if the exponent is less than -3.

49 [2nd] [)] Int - Integer part

The **Int** key is used to remove the digits after the decimal point of the number or to reduce the number to an integer.
The function has the same meaning as rounding towards zero.
If the **INV** prefix is used before the **Int** command, the inverse function is executed (**Frac**) by removing the integer part of the number and keeping only the decimal part.

Example :
2. 3 int ... integer part of 2.3 [2]
2. 3 +/- Int ... integer part of -2.3 [-2]
2. 3 INV Int ... decimal part of 2.3 [0.3]
2. 3 +/- INV Int ... decimal part of de -2.3 [-0.3]

50 [2nd] [×] Deg - Degrees

The **Deg** key switches trigonometric function calculations to degrees (a full angle is 360°).

51 [GTO] GTO - Go to a label (or address)

GTO allows you to perform an unconditional jump in a program. Its parameter is a numeric code from 0 to 9 corresponding to a label (**Lbl**) of the program.
By entering the **Pgm** command in the program before the **GTO** command, a jump to the label in another program area can be performed.
When the **GTO** instruction is used in run mode, the program pointer is positioned on the selected label.
By pressing the **INV** prefix before the **GTO** instruction, the pointer in the program can be moved to an absolute address, which is entered as a 2-digit numeric code from 00 to 49. This function cannot do part of the program, it is executed immediately, and makes it possible to move the program pointer both in execution mode and in programming mode.
The **GTO** key has another special function: if you hold it down while the program is running, the display will scroll through the contents of the display (**X** register) on the bottom line, and the address of the step executed on the upper line. However, this tracking slows down the program considerably.

55 [×] x - Multiply

The **x** key is used to multiply a first operand by a second operand. If this is the lowest level of the expression, the calculation can be repeated for another first operand by repeatedly pressing **=**.

56 [2nd] [GTO] Dsz - Program Loop after decrement

The **Dsz** instruction is used to execute a program sequence iteratively using a loop according to a number of passes specified in the **R0** register.
The **Dsz** function consists in the fact that it decrements (decreases by 1) the register **R0** and stops looping as soon as it reaches zero, it ignores the next command and continues the operation at the next step. If zero is not reached (ie the loop has not yet terminated), the command following the **Dsz** command is executed. Typically, Dsz is followed by the **GTO** command, which returns to the start of the loop label.

If the **INV** prefix is given before the Dsz instruction, the opposite direction of the instruction is executed - the next instruction is skipped if the decrement result is not zero. This variant is generally used at the beginning of the loop. When the loop counter reaches zero, the next statement, which is usually a loop statement **GTO**, is executed,

Dsz handles the **R0** register and works more precisely as follows :
If the value contained in register **R0**

- is greater than 0 before the operation, the register value is decreased by 1.
- is less than 0, the value is increased by 1.
- is equal to 0, the value remains unchanged.

If the result of the operation is zero or the operation has crossed the zero boundary, the operation for zero is performed according to the selected function.
Which means that if the decrement/increment result did not reach zero, the loop repeats (the next instruction is executed). With the **INV** prefix, a jump is performed on the contrary if the result of the operation reaches zero (or exceeds it).

Example *(factorial of a number)* ***:***

	[RST] [LRN]	... activate programming mode
00	[Lbl] [1]	... Program start label
01	[STO] [0]	...stores the entered number in the **R0** register
02	[1]	... initialization for product calculation
03	[Lbl] [9]	... label of the beginning of the loop
04	[(] [CE] [×] [RCL] [0] [)]	...multiplies the **X** register by **R0**
09	[Dsz] [GTO] [9]	... decrement **R0**, test if zero and loop otherwise
11	[INV] [SBR]	... end of program
	[LRN]	... exit programming mode
	[1] [2] [SBR] [1]	... test, factorial of 12 [479001600]

57 [2nd] [7] STO* - Indirect storage of a register

STO* stores the contents of the **X** (display) register in a data register in the same way as the **STO** instruction.
On the other hand, instead of using an instruction parameter as destination register number, the destination register number is contained in register **R8**.
All data registers **R0** to **R79** can be addressed indirectly in this way.
If the **INV** prefix is given before the instruction, the destination register number is contained in the **R9** register (alternative index).
Registers **R10** to **R79** belong to the group of extended registers with a precision reduced to 13 digits.
Saving deletes the last 2 digits of the mantissa.
Entering address -1 to -256 (negative number) in register **R8** (or **R9**) with instruction **STO*** (or **INV STO***) sends a byte with value 0 to 255 (display number) at port 0 to 255 (depending on the content of register **R8**, **R9**) to the connected external device. (See External Devices and Ports).
This function cannot be used with the **ET-57B** calculator variant.

58 [2nd] [8] RCL* - Indirect recall of a register

RCL* recalls a number from a data register in the same way as the **RCL** instruction.
On the other hand, instead of using an instruction parameter as the number of the original register of the number to be recovered, the number of the original register is contained in the register **R8**.
All registers from **R0** to **R79** can be addressed indirectly in this way.
If the **INV** prefix is given before the instruction, the original register number is contained in the **R9** register (alternative index).
Registers **R10** to **R79** belong to the group of extended registers with a precision reduced to 13 digits.
The mantissa is completed by 2 digits 0 at the end.
Entering the address -1 to -256 (negative number) in the **R8** (or **R9**) register with the **RCL*** (or **INV RCL***) instruction, reads a byte with the value 0 to 255 from port 0 to 255 (depending on the content of register **R8**, **R9**) of a connected external device. (See External Devices and Ports).
This function cannot be used with the **ET-57B** calculator variant.

59 [2nd] [9] SUM* - Indirect add into a register

The **SUM*** instruction adds (or subtracts with **INV**) a number to the destination register in the same way as the **SUM** instruction, only instead of the instruction parameter, the destination register number is held in the **R8** register.
Registers **R10** to **R79** belong to the group of extended registers with a precision reduced to 13 digits.
The mantissa is completed by 2 digits 0 at the end.

60 [2nd] [-] Rad - Radians

The **Rad** key toggles trigonometric function calculations in radians (a full angle is 2*pi rad).

61 [SBR] SBR - Subroutine call

The **SBR** (*Subroutine*) key is used to call a subroutine using as a parameter the numeric code from 0 to 9 of the called label.
By specifying the **Pgm** command in the program before the **SBR** command, a subroutine from another program space is called.
If the **SBR** instruction is used in run mode, the subroutine is executed immediately.
When calling a subroutine, the address of the instruction following the **SBR** instruction code is first stored in the address stack. The address stack has a capacity limited to 7 subroutines.
The subroutine ends with the **INV SBR** instruction which ensures the return of the subroutine to the calling program thanks to the original address which is taken from the address stack.
If the subroutine was launched from another program space, control returns to the original program space. If the subroutine was started from the keyboard, execution stops.

65 [-] - - Substraction

The minus sign (**-**) is used to subtract the operand entered after the sign from the operand entered before the sign. If this is the lowest level of the expression, the calculation can be repeated for another first operand by pressing **=** repeatedly.
Entering the prefix **INV** before the minus sign (**-**) deactivates debug mode (display of the mantissa of the number). Debug mode is activated by the **INV +** instruction.

66 [2nd] [SBR] x=t - Equality test

The instruction **x=t** makes it possible to compare the register **X** (content of display) with the auxiliary register **T** (loaded by the key **x<>t** , corresponds to the register **R7**).
If the registers match, the next instruction is executed. Otherwise, the program jumps to the next instruction.
If the **INV** prefix is specified before the **x=t** code, the inverse function is executed - the execution of the following command if the **X** register is different from the **T** register.

67 [2nd] [4] Inc* - Register Increment/Decrement

The **Inc*** instruction allows to increment/decrement the content of a data register in the same way as the **Inc** instruction, only instead of the instruction parameter specifying the register number, this register number is contained in the **R8** register.
By placing the **INV** prefix before the **Inc*** instruction, instead of the increment of 1 of the target register, the decrement of -1 is performed.

68 [2nd] [5] Exc* - Indirect exchange X with a register

The **Exc*** instruction is used to exchange the displayed number with the contents of a data register similar to the **Exc** instruction.
On the other hand, instead of using an instruction parameter as the number of the register containing the value to be exchanged, the number of this register is contained in the register **R8**.
All data registers **R0** to **R79** can be addressed indirectly in this way.
If the **INV** prefix is given before the instruction **Exc***, the register number is taken from the data register **R9** (alternative index).

69 [2nd] [6] Prd* - Multiply in an indirect register

The **Prd*** instruction is used to multiply or divide (with the **INV** prefix) the contents of a data register by the entered number in the same way as the **Prd** instruction.
On the other hand, instead of using an instruction parameter as the number of the register containing the value to be multiplied, the number of this register is contained in the register **R8**.
All registers from **R0** to **R79** can be indirectly addressed in this way.

70 [2nd] [+] Grd - Gradians

The **Grd** key switches trigonometric function calculations to grads (a full angle is 400 gon).

71 [RST] RST - Reset step address

The **RST** key is used to reset the program pointer, ie to reset the pointer to 0 (not 00). The selected program area remains unchanged.

75 [+] + - Addition

The plus sign (**+**) adds the operand entered after the sign to the operand entered before the sign. If this is the lowest level of the expression, the calculation can be repeated for another first operand by pressing **=** repeatedly.
Entering the **INV** prefix before the plus sign (**+**) activates debug mode (displays the mantissa of the number). Debug mode can be disabled by the **INV -** instruction.

76 [2nd] [RST] x>=t - Greater than or equal to

The instruction **x>=t** makes it possible to compare the register **X** (content of display) with the auxiliary register **T** (loaded by the key **x<>t**, corresponds to the register **R7**).
If register **X** is greater than or equal to register **T**, the instruction following the instruction **x>=t** is executed. Otherwise, the next command is ignored.
If the **INV** prefix is specified before the **x>=t** code, the inverse function is executed - the execution of the following command if the **X** register is smaller than the **T** register.

77 [2nd] [1] Inc. - Increment/Decrement a Register

The **Inc** instruction increments (increases by 1) the content of a data register **R0** to **R9**, whose number 0 to 9 is given as an instruction parameter. If the **INV** prefix is given before the **Inc** instruction, the reverse operation is performed - decrement the register (decrease by 1).

78 [2nd] [2] Pgm - Program space selection

A program area can be selected with the **Pgm** key. The program area is selected by entering the number 0 to 9 as a parameter of the **Pgm** instruction.
There are a total of 10 independent program areas available in the calculator, marked with numbers from 0 to 9.
Each program area has 50 program steps, i.e. a total of 500 program steps in all.
When switching the program area in run mode (from the keyboard), the program pointer is simultaneously reset to the start of the newly selected program area (step 00).
Entering the **Pgm** command into a program does not change the program space permanently, only temporarily for a later **GTO** or **SBR** instruction.
In this way, subroutines can be called or jumps between program areas can be made.
The **GTO** and **SBR** instructions do not have to immediately follow the **Pgm** instruction.

79 [2nd] [3] Rand - Random number generator

The **Rand** function calculates a random number greater than or equal to 0 and less than 1.

The ***LCG*** (*Linear Congruential Generator*) and the formula

Rand = (Seed - (Seed * 214013 + 2531011) mod 4294967296) / 4294967296

are used to calculate the random number.

The generator seed has a range of 32 bits. The generated number is converted to a float by dividing by 2^32. This ensures that the resulting random number is between 0 and 1, including zero but excluding the value 1.

The random number generator keeps counting each time it is used, which ensures that the sequence of numbers generated does not repeat (more precisely - the repeat interval is very large, 2^32 numbers). Each time the calculator is turned on, the generator seed is read from the EEPROM and a new value is stored. This ensures that generated sequences do not repeat after the calculator is powered on.

Example :

[RST] [LRN]		... activate programming mode
00	[Lbl] [1]	... Program start label (roll the die)
01	[(] [6] [×] [Rand] [+] [1] [)]	... generates a number between 1 and 6
08	[Int]	... integer part of the number
09	[INV] [SBR]	... end of program
[LRN]		... exit programming mode
[SBR] [1]		... test, roll the dice

The test can be completed with a counting test of the number of outputs of each number (1 to 6) and the counting of the number of throws per second.

Continued Example :

[LRN]		... activate programming mode
10	[Lbl] [2]	... Program start label (cumulative number of launches)
11	[SBR] [1]	... calls the dice rolling program
12	[STO] [8]	... stores the die number in R8 for use as an index
13	[Inc*]	... register increment indexed by die number (R8)
14	[GTO] [2]	... loop to continue counting
[LRN]		... exit programming mode
[INV] [C.t]		... initialization of all data registers
[SBR] [2]		... start counting test
[R/S]		... wait several minutes to stop the program
[RCL] [0] ... [RCL] [7]		... check the content of registers R0 to R7

After 10 minutes of testing, the **R1** to **R6** registers contain roughly similar numbers (similar case numbers), the **R0** and **R7** registers must contain 0.
(for example: 0, 2762, 2727, 2834, 2701, 2671, 2763, 0, which corresponds to the hypothesis of a generation speed of 27 dice rolls per second and a fairly homogeneous distribution between 1 and 6.)

80 [2nd] [=] Var - Scatter

The **Var** statement calculates the variance of the 'x' and 'y' values entered using the statistical function **Stat**. Pressing Var displays the variance of the 'y' values, and using **INV** before the **Var** statement displays the variance of the 'x' values.

The variance is calculated by the formula :

var = sum(y^2)/N - (sum(y)/N)^2

The square root of the variance gives the standard deviation '*s*'.

81 [R/S] R/S - Program start and stop

The **R/S** key can be used to start or stop a running program.

On startup, the program begins to run from the current program pointer (the current address can be found by switching to **LRN** programming mode).

After stopping, the program may not always be able to continue running - the subroutine's return address may be lost.

83 [.] . - Decimal point

The dot (**.**) is the separator for whole digits and decimal digits in a number.
In scientific notation, if this key is pressed while entering the exponent of a number, editing reverts to entering the mantissa of the number.

84 [+/-] +/- - Change of sign

The **+/-** key changes the sign of the number on the display.
Using it while typing the exponent of a number in scientific notation, changes the sign of that exponent.

85 [=] = - Perform the calculation

The **=** sign is used to close open arithmetic operations and to perform calculations.
By repeatedly pressing the **=** key, the last operation performed at the lowest level can be repeated. The first operand is the displayed number, the second operand is the number entered during the operation as the second operand (or second result of the intermediate calculation).
Caution: Some original **TI-57** programs, for which repetitive operations are not allowed, use the **=** key more than once to cause an error. When importing a program, it may be necessary to check and solve this case.

Example:

5 x **6** =	[30]	... 5 x 6 = 30
7 =	[42]	... 7 x 6 = 42 the second operand (**6**) is reused
9 : **3** =	[3]	... 9 / 3 = 3
12=	[4]	... 12 / 3 = 4 the second operand (**3**) is reused

86 [2nd] [R/S] Lbl - Labels

The **Lbl** instruction can be used to mark the beginning of a sequence in the program like a label.
In each program area, 10 labels can be used, denoted **Lbl 0** to **Lbl 9**. The label number is specified as numeric parameter 0 to 9 of the **Lbl** instruction.
You can jump to the program location marked with a label using the **GTO** jump instruction or the **SBR** subroutine call instruction.
With the use of the **Pgm** program space select statement, jumps and subroutine calls can also be made between program spaces.

87 [2nd] [0] x! - Factorial

The **x!** key is used to calculate the factorial. The factorial of a number is obtained by multiplying the successive values 1, 2, 3, ... up to the specified number. The number entered is an integer between 1 and 69.
(69! =
171122452428141311372468338881272839092270544893520369393648040923257279754140647424000000000000000 [99 digits])

Example :
6 x! ... factorial of the number 6 [6! = 1*2*3*4*5*6 = 720]

88 [2nd] [.] Stat - Statistics

The **Stat** instruction is used to enter data that can be used for statistical calculations (mean, variance).
The instruction uses data registers **R0** through **R5** to store the intermediate calculation. Before use, it is advisable to first reset the registers with the **INV C.t** command, which resets all data registers **R0** to **R79**.

Use of data registers :

- **R0** ... number of elements N
- **R1** ... the sum of *y*
- **R2** ... the sum of *y*^2
- **R3** ... the sum of *x*
- **R4** ... the sum of *x*^2
- **R5** ... the sum of *x***y*

and **R7** ... **T** register

When entering statistical data in pairs (x, y), the value 'x' is entered first and by pressing **x<>t** it is transferred to the **T** register (register **R7**).
Then the 'y' value is entered and by pressing **Stat** the 'x' and 'y' values are saved.
The display (in the **X** register) will show the number of 'n' elements inserted so far. The content of the **T** register (the value of 'x') is increased by 1 with the **Stat** instruction, because if the values of 'x' must increment by 1, it is not necessary to insert them, it is enough to insert the initial value of 'x' into the **T** register, then write only the 'y' values.
If it is not necessary to evaluate pairs of values (x, y), just enter the value 'y'.
If the **INV** prefix is used before the **Stat** instruction, the entered value will be subtracted. This is how you can correct a wrong value - enter the 'x' value of the wrong data, press **x<>t**, enter the 'y' value of the wrong data and press **INV Stat** to delete the wrong data. The content of register **T** (with the value 'x') is then decremented by 1.
Then the entry of new correct data can be continued. If it is not necessary to evaluate pairs of values (x, y), it is sufficient to enter only the value 'y' of the erroneous data.

Example :

INV	C.t			...initializing data registers
9	6	Stat	[1]	... 1st data 96
8	1	Stat	[2]	... 2nd data 81
9	7	Stat	[3]	... 3rd incorrect data
9	7	Stat	[2]	... cancellation of the 3rd data
8	7	Stat	[3]	... 3rd corrected data 87
7	0	Stat	[4]	... 4th data 70
9	3	Stat	[5]	... 5th data 93
7	7	Stat	[6]	... 6th data 77
Mean			[84]	... average of the entered values
Var			[81,333...]	... Deviation
√x			[9.0185...]	... standard deviation
RCL	1		[504]	... sum of all values

89 2nd +/- Mean - Mean

The **Mean** statement calculates the average of the 'x' and 'y' values entered using the statistical function **Stat**.
After pressing **Mean**, the display shows the average of the 'y' values.
Typing the **INV** prefix before the **Mean** statement calculates the average of the 'x' values.

15. Sample programs

1. Roll the Dice (ET-57 Version)

The program uses the internal random number generator.

Use :

SBR 1	... generates a random integer between 1 and 6
R/S	... next number

Program :

00	86 1	Lbl 1	subroutine start label
01	43	(	calculates (6 x Rand + 1), random number 1..6.9999
02	6	6	
03	55	x	
04	79	Rand	random number from 0 to 0.99999...
05	75	+	
06	1	1	
07	44	)	
08	49	Int	integer part of the number
09	-61	INV SBR	end of subroutine
10	51 1	GTO 1	next number with R/S

2. Roll the Dice (TI-57 Version)

Since the original **TI-57** did not have a **Rand** (random number generator) function, the latter's dice-rolling program cost twice the number of steps, plus the use of a register. (this program is also applicable, with some minor variations, to the successor models of the **TI-57**, namely the **TI-57 LCD**, **TI-57 II** and **TI-62**).

Register :

R0 ... Random number generator seed.

Use :

x [STO] [1] ... initialize the number generator with a seed = x

[SBR] [1] ... generates a random integer between 1 and 6

[R/S] ... next number

Program :

00	86 1	Lbl 1	subroutine start label
01	43	(	
02	43	(	\
03	33 0	RCL 0	| Recall seed
04	75	+	|
05	30	pi	|
06	44	)	|
07	35	y^x	> Calculation of frac((Seed + pi)^8)
08	8	8	|
09	44	)	|
10	43	(	|
11	-49	INV Int	/
12	32 0	STO 0	Stores the new seed (Seed)
13	55	x	\
14	6	6	|
15	75	+	> Dice value from 1 to 6
16	1	1	|
17	44	)	/
18	49	Int	
19	-61	INV SBR	
20	51 1	GTO 1	next number with R/S

3. LED light snake

The program controls an external device - the ERAM100 effect frame. By writing a byte to port 0, 8 LEDs are controlled. The byte written is a combination of bits lighting up individual LEDs, like the sum of the weights of the bits :

- bit 0 ... weight 1 ... LED 1
- bit 1 ... weight 2 ... LED 2
- bit 2 ... weight 4 ... LED 3
- bit 3 ... weight 8 ... LED 4
- bit 4 ... weight 16 ... LED 5
- bit 5 ... weight 32 ... LED 6
- bit 6 ... weight 64 ... LED 7
- bit 7 ... weight 128 ... LED 8

Use :

[RST] [R/S] ... start the snake :
3 bright LEDs run along the LED strip

Registers :

R5 ... save snake position
R8 ... index, -1 is the index of port 0 (output to LED)

Program :

00	7	7	bits value of snake appearance (LED 1,2,3 lights up)
01	32 5	STO 5	Stores snake's position
02	1	1	
03	84	+/-	
04	32 8	STO 8	sets the index of R8 to -1, which is the address of port 0
05	2	2	
06	5	5	
07	6	6	
08	22	x<>t	store 256 in T for pattern overflow test
09	86 9	Lbl 9	cycle start label
10	33 5	RCL 5	recall of the position of the snake
11	55	x	
12	2	2	
13	85	=	(position x 2) corresponds to a shift of 1 bit to the left
14	-76	INV x>=t	location < 256?
15	51 8	GTO 8	no position overflow, go to Lbl 8
16	65	-	
17	2	2	
18	5	5	
19	5	5	
20	85	=	rotation: - 256 (remove bit 7) + 1 (add bit 0)
21	86 8	Lbl 8	
22	32 5	STO 5	stores the snake's new position
23	57	STO*	storage at address -1 sends the byte to port 0
24	36	Pause	small pause of 0.25 seconds
25	51 9	GTO 9	loop repeat

4. Calculation of the polynomial

The program calculates the value of the polynomial

P(x) = a0 + a1*x + a2*x^2 + ... an*x^n

of order n for the given x, if the coefficients a0 to an are entered.

Registers :

R0 ... Dsz iteration counter
R6 ... number of coefficients N (= polynomial order n + 1)
R8 ... index register
R9 ... input x
R10 and following ... polynomial coefficients a0, a1, ... an (N coefficients)

Use :

Example, polynomial P(x) = 2 - 3x + x^2 :

RST R/S	... start entering the coefficients
2 R/S	... enter the coefficient a0 = 2
3 +/- R/S	... input of coefficient a1 = -3
1 R/S	... coefficient input a2 = 1
2 SBR 1	... calculation of the value of the polynomial P(2) = 0
1 +/- R/S	... calculation of the value of the polynomial P(-1) = 6
1 5 R/S	... calculation of the value of the polynomial P(15) = 182

Program :

00	15	CLR	reset
01	32 6	STO 6	reset of the number of coefficients N
02	9	9	index of the first coefficient 10 - 1 = 9
03	32 8	STO 8	index registry settings
04	86 9	Lbl 9	loop labels for entering coefficients
05	81	R/S	stop the program, wait for the next coefficient
06	77 8	Inc 8	increment index register **R8**
07	57	STO*	store the coefficient
08	77 6	Inc 6	incrementing the number of coefficients N
09	51 9	GTO 9	continue with the next coefficient
10	86 8	Lbl 8	calculation repeat label
11	81	R/S	display the result and wait for another 'x'
12	86 1	Lbl 1	start label of the polynomial quantization function
13	32 9	STO 9	save 'x' entered
14	43	(	
15	33 6	RCL 6	number of coefficients N
16	32 0	STO 0	**Dsz** counter preparation
17	75	+	
18	9	9	N + 9 = index of the last coefficient an
19	44	)	
20	32 8	STO 8	**R8** index register setting
21	58	RCL*	recall the last coefficient an
22	86 7	Lbl 7	loop start label
23	-56	INV Dsz	decrement **R0** and skip instruction when > 0
24	51 8	GTO 8	loop if **R0** differs from 0 otherwise end of loop
25	-77 8	INV Inc 8	decrement index register **R8**
#	43	(	
27	14	CE	Restore last **X** register
28	55	x	
29	33 9	RCL 9	Calculation Display * 'x' entered
30	75	+	
31	58	RCL*	addition of another coefficient ai (from index **R8**)
32	44	)	
33	51 7	GTO 7	continue the loop

5. Complex numbers

Arithmetic of complex numbers.
When entering a number, enter the real part, press **x<>t**, and enter the imaginary part.
In the **T** register (register **R7**) there will be the real part of the number, and in the **X** register (display) the imaginary part.
When reading the result, after pressing **x<>t** you will read the real part, after pressing a second time on **x<>t** you will read the imaginary part.

For 2-operand functions, specify the first operand (**X**) using **SBR 1**.
Then specify the second operand (**Y**) and call the function.
The result simultaneously becomes the new first operand (**X**).
For 1-operand functions, specify the operand (**Y**) and call the function.
The original first operand (**X**) remains unchanged.

The program occupies 2 program spaces, **Pgm 0** and **Pgm 1**.

Note : The ln(Y) and exp(Y) functions change the angular measure to radians.

Use :
Program space **Pgm 0** :

SBR 1	... input of the first operand
SBR 2	... adding the second operand X+Y
SBR 3	... subtraction of the second operand X-Y
SBR 4	... multiplication by the second operand X*Y
SBR 5	... division by the second operand X/Y
SBR 6	... negation -Y

Note: The **SBR 5** function calls the **SBR 1** function from **Pgm 1**.

Program space **Pgm 1** :

SBR 1	... inverse of 1/Y
SBR 2	... square of Y^2
SBR 3	... square root of sqrt(Y)
SBR 4	... natural logarithm of ln(Y)
SBR 5	... natural exponent exp(Y)

Example : calculates exp(((2+3i) - (1-i)) / (4+5i))

Pgm 0		... selection of program space 0
2 x<>t 3 SBR 1		... input of the first operand (2+3i)
1 x<>t 1 +/- SBR 3		... subtraction of the second operand (1-i)
x<>t	[1]	
x<>t	[4]	... intermediate result = (1+4i)
4 x<>t 5 SBR 5		... division by operand (4+5i)
x<>t	[0.5853...]	
x<>t	[0.2682...]	... intermediate result = (0.5853...+0.2682...i)
Pgm 1		... selection of program space 1
1 SBR 5		... natural exponent exp (Y)
x<>t	[1.7314...]	
x<>t	[0.4760...]	... result (1.7314... + 0.4760..i)

Registers :

R1 ... 'a', the real part of the first operand of X
R2 ... 'b', the imaginary part of the first operand of X
R3 ... 'c', the real part of the second operand of Y
R4 ... 'd', the imaginary part of the second operand Y
R7 ... **T** register

Program :

Pgm 0 ... selection of program space 0

- calculation of the subtraction of the second operand X-Y

00	86 3	Lbl 3	X-Y function label
01	61 6	SBR 6	call the negation function -Y

- calculation of the second operand X+Y

X + Y = (a + c) + (b + d)*i

02	86 2	Lbl 2	X+Y function label
03	34 2	SUM 2	add the imaginary part of Y to X
04	33 2	RCL 2	recall of the imaginary part of X
05	22	x<>t	exchange of real <> imaginary parts
06	34 1	SUM 1	add the real part of Y to X
07	33 1	RCL 1	recall of the real part of X
08	22	x<>t	exchange of real <> imaginary parts

- entering the first operand X

09	86 1	Lbl 1	X function label
10	32 2	STO 2	store the imaginary part of X
11	22	x<>t	exchange of real <> imaginary parts
12	32 1	STO 1	store the real part of X
13	22	x<>t	exchange of real <> imaginary parts
14	-61	INV SBR	

- calculation of the division by the second operand X/Y

15	86 5	Lbl 5	X/Y function label
16	78 1	Pgm 1	choice of program space 1
17	61 1	SBR 1	inverse function call of 1/Y

- calculation of the multiplication by the second operand X*Y

X * Y = (a*c - b*d) + (a*d + b*c)*i

18	86 4	Lbl 4	X*Y function label
19	32 4	STO 4	storage of the imaginary part Y (=d)
20	22	x<>t	exchange of real <> imaginary parts
21	43	(	
22	32 3	STO 3	storage of the real part of Y (=c)
23	55	x	
24	33 1	RCL 1	real part of X (=a)
25	65	-	
26	33 2	RCL 2	recall imaginary part of X (=b)
27	55	x	
28	33 4	RCL 4	recall imaginary part of Y (=d)
29	44	)	calculation (a*c - b*d) real part
30	22	x<>t	exchange of real <> imaginary parts
31	43	(	
32	33 1	RCL 1	recall of the real part of X (=a)
33	55	x	
34	33 4	RCL 4	recall imaginary part of Y (=d)
35	75	+	
36	33 2	RCL 2	recall of the imaginary part X (=b)
37	55	x	
38	33 3	RCL 3	recall of the real part of Y (=c)
39	44	)	calculation (a*d + b*c) imaginary part
40	51 1	GTO 1	store result in X

- Change operator sign -Y

41	86 6	Lbl 6	-Y function label
42	84	+/-	negation of the imaginary part
43	22	x<>t	exchange of real <> imaginary parts
44	84	+/-	negation of the real part
45	22	x<>t	exchange of real <> imaginary parts
46	-61	INV SBR	

Pgm 1 ... selection of program space 1

- inverse of operand 1/Y

1/Y = (c - d*i)/(c^2 + d^2)

00	86 1	Lbl 1	1/Y function label
01	43	(	
02	84	+/-	change sign imaginary part of Y (=d)
03	45	:	
04	43	(	
05	23	x^2	j^2
06	75	+	
07	22	x<>t	
08	32 3	STO 3	storage of the real part of Y (=c)
09	23	x^2	c^2
10	44	)	calculation (c^2 + d^2)
11	-39 3	INV Prd 3	divide c by (c^2 + d^2)
12	44	)	end of calculation of the imaginary part
13	22	x<>t	exchange of real <> imaginary parts
14	33 3	RCL 3	reminder of the new real part
15	22	x<>t	exchange of real <> imaginary parts
16	-61	INV SBR	

- squaring operand Y^2

17	86 2	Lbl 2	Y^2 function label
18	43	(	
19	-27	INV P->R	convert to polar coordinates
20	55	x	the angle will be multiplied by *2
21	22	x<>t	exchange of real <> imaginary parts
22	23	x^2	squared radius
23	86 9	Lbl 9	
24	22	x<>t	exchange of real <> imaginary parts
25	2	2	
26	44	)	angle * 2 (or angle / 2) is calculated here
27	27	P->R	convert to cartesian coordinates
28	-61	INV SBR	

- square root operand sqrt(Y)

29	86 2	Lbl 3	sqrt(Y) function label
30	43	(	
31	-27	INV P->R	conversion to polar coordinates
32	45	:	the angle will be divisible by /2
33	22	x<>t	exchange of real <> imaginary parts
34	24	Vx	square root of radius
35	51 9	GTO 9	realization of the function

- natural logarithm of the operand ln(Y)

36	86 4	Lbl 4	ln(Y) function label
37	60	Rad	switch to radians
38	-27	INV P->R	converts to polar coordinates
39	22	x<>t	exchange of real <> imaginary parts
40	13	lnx	logarithm of the radius
41	22	x<>t	exchange of real <> imaginary parts
42	-61	INV SBR	

- natural exponent of the operand exp(Y)

43	86 5	Lbl 5	exp(Y) function label
44	22	x<>t	exchange of real <> imaginary parts
45	-13	INV lnx	exponent exp(x)
46	22	x<>t	exchange of real <> imaginary parts
47	60	Rad	switch to radians
48	27	P->R	converts to Cartesian coordinates
49	-61	INV SBR	

6. Ramanujan approximation of the factorial x!

Calculating the factorial (including decimals) using the Ramanujan approximation.
Accuracy achieved: Values around 1 precision 3 digits, values around 69 (maximum) precision 10 digits.

Formula : x ! = sqrt(pi) * (x/e)^x * (((8*x + 4)*x + 1)*x + 1/30)^(1/6).

Use :

x SBR 1 ... calculates the factorial x!

x' R/S ... calculates for another number

Examples :

1 SBR 1 ... 1! = 1,00028..., the correct valmue must be 1,0

1 . 2 R/S ... 1.2 ! = 1,101987..., must be 1,101802...

1 0 R/S ... 10 ! = 3628800.3116, must be 3628800

6 9 R/S ... 69 ! = 1,7112245244+98, must be 1,7112245243+98

Register :

R1 ... entered value of x

Program :

00	86 1	Lbl 1	start of program
01	43	(	
02	43	(	
03	32 1	STO 1	store input value
04	45	:	
05	1	1	
06	-13	INV lnx	calculation of the constant e
07	44	)	calculation of x/e
08	35	y^x	
09	33 1	RCL 1	calculation of (x/e)^x
10	55	x	
11	30	pi	
12	24	Vx	pi square root calculation
13	55	x	
14	43	(	
15	43	(	
16	43	(	
17	8	8	
18	55	x	
19	33 1	RCL 1	calculation of 8 * x
20	75	+	
21	4	4	
22	44	)	calculation of (8*x + 4)
23	55	x	
24	33 1	RCL 1	
25	75	+	
26	1	1	
27	44	)	calculation of ((8*x + 4)*x + 1)
28	55	x	
29	33 1	RCL1	
30	75	+	
31	3	3	
32	0	0	
33	25	1/x	
34	44	)	calculation of ...*x + 1/30)
35	-35	INV y^x	6th root
36	6	6	
37	44	)	
38	-61	INV SBR	
39	51 1	GTO 1	repeat the calculation for another value

7. Stirling approximation of the factorial ln(x!)

Calculating the factorial (including decimals) using Stirling's approximation.
Accuracy achieved: Values around 1 3-digit precision, values around 69 (maximum x!) 10-digit precision, values around 200 15-digit precision.

Formula : ln(x!) = x*ln(x)+ln(sqrt(2*pi))-x+ln(sqrt(x+1/6+1/72/x-31/6480/x^2)).

Use :

x SBR 1 ... calculates the logarithm of the factorial ln(x!)

x' R/S ... calculates for another number

x SBR 2 ... calculates the factorial x!

x' R/S ... calculates for another number

Examples :

1 SBR 2 ... 1! = 0.99990..., the correct value must be 1,0

1 . 2 R/S ... 1.2 ! = 1.10177..., must be 1,101802...

1 0 R/S ... 10 ! = 3628800.1322, must be 3628800

6 9 R/S ... 69 ! = 1.7112245243+98, correct value

1 0 0 0 SBR 1 ... ln(1000!) = 5912.1281785, correct value

Register :

R1 ... entered value of x

Program :

- calculation of ln(x!)

0	81 1	Lbl 1	start of program ln(x!)
1	43	(	
2	32 1	STO 1	store the entered value of x
3	55	x	
4	13	lnx	calculation of x*ln(x)
5	75	+	
6	43	(	
7	2	2	
8	55	x	
9	30	pi	
10	44	)	
11	24	Vx	
12	13	lnx	calculation of ln(sqrt(2*pi))
13	65	-	
14	33 1	RCL 1	Recall x
15	75	+	
16	43	(	
17	33 1	RCL 1	Recall x
18	75	+	
19	6	6	
20	25	1/x	number + 1/6
21	75	+	
22	7	7	
23	2	2	
24	25	1/x	
25	45	:	
26	33 1	RCL 1	number + 1/72/x
27	65	-	
28	3	3	
29	1	1	
30	45	:	
31	6	6	
32	4	4	
33	8	8	
34	0	0	
35	45	:	
36	33 1	RCL 1	
37	23	x^2	number - 31/6480/x^2
38	44	)	
39	24	Vx	square root of total
40	13	lnx	
41	44	)	
42	-61	INV SBR	
43	51 1	GTO 1	repeat the calculation for another value

- calculation of x! [i.e. e^ln(x!)]

44	86 2	Lbl 2	
45	61 1	SBR 1	call function ln(x!)
46	-13	INV lnx	exponent, x! = e^ln(x!)
47	-61	INV SBR	
48	51 2	GTO 2	repeat the calculation for another value

8. Determining the zeros of a function

The program searches for zero crossings of the user function. The program occupies program spaces **Pgm 0** and **Pgm 1**. The user function is entered into program space **Pgm 2** and marked with label **Lbl 0**. The function can temporarily use register **R7** (register **T**) and registers **R10** and above.

The function first searches for a step interval dx = (xmax - xmin)/10 in which the sign of y changes. It then refines the place of passage by dividing the interval by zero until the deviation eps = dx/100000. The size of the dx step can be fixed at address 22. With a large value of the dx step, certain zero crossings can be skipped, a small value of dx slows down the search. The magnitude of the eps deviation can be set at address 27. The eps value affects the zero crossing search accuracy achievable at the cost of slowing down the search.

Use :

SBR 0 ... user function: calculation of the value y for the given x

SBR 1 ... input of the lower limit of xmin

SBR 2 ... entering the upper limit of xmax

SBR 3 ... determine the first zero crossing

SBR 4 ... determine the next zero crossing

Examples :

zeros of the function f(x) = 4*sin(x) + 1 - x

*The program for this function f(x) must be created in the **Pgm 2** space (see following pages).*

Pgm 0 ... selection of program space 0

3 +/- SBR 1 ... input of lower limit xmin = -3

3 SBR 2 ... input of upper limit xmax = 3

SBR 3 ... find the first zero crossing = -2.2100...

SBR 4 ... find the second zero crossing = -0.3421...

SBR 4 ... find the third zero crossing = 2.7020...

SBR 4 ... fourth not found = 9.999+99

Registers :

R0 ... the y value of the tested function
R1 ... lower limit xmin
R2 ... upper limit of xmax
R3 ... delta interval dx
R4 ... x the beginning of the interval
R5 ... x end of interval
R6 ... x current
R7 ... T register, temporarily available for SBR 0
R8 ... eps deviation (minimum dx)
R9 ... current lower limit of xmin2

Program :

Pgm 0 ... selection of program space 0

- Calculation of the user function value

00	86 0	Lbl 0	subroutine 'function call'
01	78 2	Pgm 2	choice of program space 2
02	61 0	SBR 0	user function call
03	-61	INV SBR	

- Entering the lower limit of xmin

04	86 1	Lbl 1	entry of the lower limit xmin
05	32 1	STO 1	stores the lower limit of xmin
06	-61	INV SBR	

- Entering the upper limit of xmin

07	86 2	Lbl 2	entry of the upper limit of xmax
08	32 2	STO 2	stores the upper limit of xmax
09	-61	INV SBR	

- Display error : next not found

10	86 9	Lbl 9	error display
11	15	CLR	
12	25	1/x	causes error 9.99999 99
13	86 8	Lbl 8	
14	-61	INV SBR	

- Subroutine to find the first zero

15	86 3	Lbl 3	first search for zero of the function
16	33 2	RCL 2	recall upper limit xmax
17	65	-	
18	33 1	RCL 1	recall lower limit xmin
19	32 9	STO 9	stores lower limit xmin
20	85	=	calculates difference (xmax - xmin)
21	45	:	
22	1	1	**<-** size of the dx step
23	0	0	
24	85	=	calculates dx = interval/10
25	32 3	STO 3	stores delta dx interval
26	45	:	
27	5	5	**<-** amplitude of the eps deviation
28	-18	INV log	constant 100000
29	85	=	calculates eps maximum deviation
30	32 8	STO 8	stores eps deviation

- Subroutine to find next zero

31	86 3	Lbl 4	another zero search function
32	33 2	RCL 2	recall upper limit of xmax
33	22	x<>t	xmax in T
34	33 9	RCL 9	recall lower limit xmin updated
35	76	x>=t	upper limit reached?
36	51 9	GTO 9	next passage not found
37	32 4	STO 4	store x interval start
38	75	+	
39	33 3	RCL 3	dx delta interval reminder
40	85	=	
41	32 9	STO 9	stores new xmin lower limit
42	32 5	STO 5	stores x at end of interval
43	61 0	SBR 0	calculates y at the end of the interval
44	32 0	STO 0	stores value of y at the end of the interval
45	33 4	RCL 4	recall x at start of interval
46	32 6	STO 6	stores new updated x
47	61 0	SBR 0	calculates y for the new updated x
48	78 1	Pgm 1	choice of program space 1
49	51 9	GTO 9	continuation of the function in space 1

Pgm 1 ... selection of program space 1

- Continuation of the search operation for the next zero

00	86 9	Lbl 9	continuation of the search for the next zero
01	19	C.t	reset T
02	66	x=t	is the value of y equal to 0?
03	51 8	GTO 8	zero value found
04	55	x	
05	38 0	Exc 0	y_current * y_initial
06	85	=	
07	-76	INV x>=t	if multiple < 0, zero found
08	51 7	GTO 7	if zero found, refine result
09	78 0	Pgm 0	switch to program space 0
10	51 4	GTO 4	continue with next interval

- Loop to refine the result

11	86 7	Lbl 7	refining precision of zero found
12	33 4	RCL 4	x at start of interval
13	75	+	
14	33 5	RCL 5	x at the end of the interval
15	85	=	
16	45	:	
17	2	2	
18	85	=	calculates center of the interval x
19	32 6	STO 6	stores new x updated
20	33 8	RCL 8	recall ps deviation
21	22	x<>t	stores eps deviation in T
22	33 5	RCL 5	recall x at the end of the interval
23	65	-	
24	33 4	RCL 4	recall x at start of the interval
25	85	=	calculates interval length
26	-76	INV x>=t	interval < epsilon?
27	51 6	GTO 6	if interval < epsilon, zero found OK
28	33 6	RCL 6	recall x updated (interval center)
29	78 2	Pgm 2	choice of program space 2
30	61 0	SBR 0	calculates y for the new x
31	55	x	
32	33 7	RCL 0	recall previous y
33	85	=	
34	19	C.t	reset T
35	66	x=t	y=0?
36	51 6	GTO 6	zero found
37	76	x>=t	if product > 0, no sign change
38	51 5	GTO 5	no sign change, move to higher x
39	33 6	RCL 6	recall x updated (interval center)
40	32 5	STO 5	shift down, the middle will be the new end
41	51 7	GTO 7	continuation of the zero refinement loop
42	86 5	Lbl 5	label to move to a higher x
43	33 6	RCL 6	recall x updated (interval center)
44	32 4	STO 4	go up, the center will be a new beginning
45	51 7	GTO 7	continuation of the zero refinement loop
46	86 6	Lbl 6	OK result
47	33 6	RCL 6	recall result
48	78 0	Pgm 0	choice of program space 0
49	51 8	GTO 8	function return

Pgm 2 ... selection of program space 2

- example of user function f(x) = 4*sin(x)+1-x

00	86 0	Lbl 0	user function label : must be **Lbl 0**
01	43	(	
02	32 7	STO 7	stores initial X
03	60	Rad	switch to radians
04	28	sin	\
05	55	x	|
06	4	4	|
07	75	+	> user function 4*sin(x)+1-x
08	1	1	|
09	65	-	|
10	33 7	RCL 7	|
11	44	)	/
12	-61	INV SBR	

9. Simpson's rule for integration

The program calculates the numerical integral of the user function by Simpson's approximation. The program is stored in the program space **Pgm 0**, the user function is created in the program space **Pgm 1** under the label **Lbl 0**.
The specified number of steps n must be an even number.

Use :

	CLR RST	... initialize operations and initialize program pointer
xmin	R/S	... entering the lower limit
xmax	R/S	... entering the upper limit
n	R/S	... input of the number of steps (an even number!) and calculation

Examples :

integral of the function f(x) = 1/(cos(x) + 2) in the nterval 0 to pi/2
*The program for this function f(x) must be created in the **Pgm 2** space (see following pages).*

Pgm 0	... selection of program space 0
CLR RST	... initialization operations and pointer reset (not 00)
0 R/S	... entering the lower limit
π ÷ 2 = R/S	... entering the upper limit
2 0 R/S	... entry of the number of steps (even!) and calculation [0.6046..]

Result for 2 steps =	0.604998903 (3 digit accuracy)
Result for 4 steps =	0.604619709 (4 digit accuracy)
Result for 10 steps =	0.604600227 (6 digit accuracy)
Result for 20 steps =	0.604599815 (7 digit accuracy)
Result for 100 steps =	0.604599788 (10 digit accuracy)
Result for 200 steps =	0.604599788 (11 digit accuracy)
Result for 1000 steps =	0.604599788 (13 digit accuracy)
Result for 2000 steps =	0.604599788 (13 digit accuracy)
Reference result =	0.604599788

Registers :

R0 ... number of steps n, loop counter
R1 ... lower limit xmin
R2 ... step increment dx
R3 ... loss of the result of the integral y

Program :

Pgm 0 ... selection of program space 0

0	32 1	STO 1	stores the lower limit of xmin
1	81	R/S	waiting for upper bound xmax
2	65	-	
3	33 1	RCL 1	recall lower limit xmin
4	85	=	calculates interval (xmax - xmin)
5	32 2	STO 2	stores the interval
6	81	R/S	waiting for the number of steps
7	32 0	STO 0	stores number of steps n
8	-39 2	INV Prd 2	calculates step increment: interval / n,
9	61 9	SBR 9	calculates current x and y
10	32 3	STO 3	stores result of integral I
11	86 8	Lbl 8	start of calculation loop
12	-77 0	INV Inc 0	decrement the **R0** index of the loop
13	61 9	SBR 9	calculates current x and y
14	55	x	
15	4	4	
16	85	=	calculates y * 4
17	34 3	SUM 3	
18	-56	INV Dsz	decrement register **R0**, jump if > 0
19	51 7	GTO 7	go to end of loop when **R0** = 0
20	61 9	SBR 9	calculates current x and y
21	55	x	
22	2	2	
23	85	=	calculates y * 2
24	34 3	SUM 3	
25	51 8	GTO 8	next step in the loop
26	86 7	Lbl 7	end of the loop
27	61 9	SBR 9	calculates current x and y
28	34 3	SUM 3	
29	33 2	RCL 2	dx interval reminder
30	45	:	
31	3	3	calculates increment dx / 3 * y
32	55	x	
33	33 3	RCL 3	loss of result y
34	85	=	
35	81	R/S	stop, result display
36	86 7	Lbl 9	displays the calculation of current x and y
37	33 1	RCL 1	lower limit xmin
38	75	+	
39	33 0	RCL 0	step counter
40	55	x	
41	33 2	RCL 2	step increment dx
42	85	=	current coordinate calculates x
43	78 1	Pgm 1	choice of program space 1
44	51 0	GTO 0	go to user function

Pgm 1 ... selection of program space 1

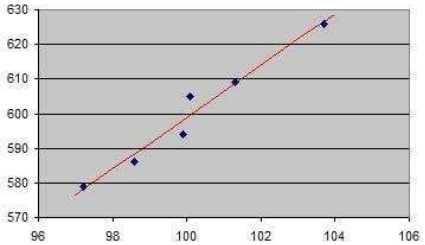
- example of user function f(x) = 1/(cos(x) + 2)

00	86 0	Lbl 0
01	60	Rad
02	29	cos
03	75	+
04	2	2
05	85	=
06	25	1/x
07	-61	INV SBR

10. Linear regression line

The program calculates the coefficients of the approximate linear regression line using the method of least squares. The pair of values (X,Y) is entered using the statistical function Stat. The regression line has the form ***y = m*x + b***.

The coefficient 'm', i.e. the slope of the line, is calculated according to the formula ***m = (sum(x*y) - sum(x)*sum(y)/N) / (sum(x^2) - somme(x)^2/ N)***.
The coefficient 'b', i.e. the displacement of the line in the Y direction, is calculated according to the formula ***b = (somme(y) - m*somme(x))/N***.



Use :

Enter the pairs (X,Y) using the key Stat

CLR RST ... program pointer initialization (step 00)

R/S ... calculates the coefficient 'm'

R/S ... calculates the coefficient 'b'

The following functions can only be called after calculating 'm' and 'b'.

x SBR 1 ... calculation of y for a given value of x

y SBR 2 ... calculation of x for a given value of y

Example :

CLR INV C.t ... clear all registers **X**, **R0** à **R79**

1 0 1 . 3 x≷t 6 0 9 Stat
... entry point 1 (101.3, 609)

1 0 3 . 7 x≷t 6 2 6 Stat
... entry point 2 (103.7, 626)

9 8 . 6 x≷t 5 8 6 Stat
... entry point 3 (98,6, 586)

9 9 . 9 x≷t 5 9 4 Stat
... entry point 4 (99.9, 594)

9 7 . 2 x≷t 5 7 9 Stat
... entry point 5 (97.2, 579)

1 0 0 . 1 x≷t 6 0 5 Stat
... entry point 6 (100.1, 605)

RST ... program pointer initialization

R/S ... calculates coefficient m = 7.4734325186

R/S ... calculates coefficient b = -148.5063762

9 7 SBR 1 ... for X = 97 Y = 576,41657811

1 0 4 SBR 1 ... for X = 104 Y = 628,73060574

5 8 0 SBR 2 ... for Y = 580 est X = 97,479488091

6 3 0 SBR 2 ... for Y = 630 est X = 104,16985425

Registers :

R0 ... number of elements N
R1 ... sum of y
R2 ... sum of y^2
R3 ... sum of x
R4 ... sum of x^2
R5 ... sum of x*y
R7 ... T-register
R8 ... calculated coefficient 'm'
R9 ... calculated coefficient 'b'

Program :

- calculation of 'm'

00	33 5	RCL 5	recall sum of x*y
01	65	-	
02	33 3	RCL 3	recall sum of x
03	55	x	
04	33 1	RCL 1	recall sum of y
05	45	:	
06	33 0	RCL 0	recall number of items N
07	85	=	(sum(x*y) - sum(x)*sum(y)/N)
08	45	:	
09	43	(	
10	33 4	RCL 4	recall sum of x^2
11	65	-	
12	33 3	RCL 3	recall sum of x
13	23	x^2	
14	45	:	
15	33 0	RCL 0	recall number of items N
16	85	=	
17	32 8	STO 8	memorization of the coefficient m
18	81	R/S	program stop, display m

- calculation of 'b'

19	33 1	RCL 1	recall sum of y
20	65	-	
21	33 8	RCL 8	recall coefficient m
22	55	x	
23	33 3	RCL 3	recall sum of x
24	85	=	
25	45	:	
26	33 0	RCL 0	recall number of items N
27	85	=	
28	32 9	STO 9	memorization of the coefficient b
29	81	R/S	program stop, display b

- Calculation of y for a given value of x

30	86 1	Lbl 1	
31	43	(	
32	14	CE	
33	55	x	
34	33 8	RCL 8	recall coefficient m
35	75	+	
36	33 9	RCL 9	recall coefficient b
37	44	)	
38	-61	INV SBR	

- Calculation of x for a given value of y

39	86 2	Lbl 2	
40	43	(	
41	43	(	
42	14	CE	
43	65	-	
44	33 9	RCL 9	recall coefficient b
45	44	)	
46	45	:	
47	33 8	RCL 8	recall coefficient m
48	44	)	
49	-61	INV SBR	