# SK-702 User Manual

Congratulations on your purchase of the SK-702 electronic computing machine. You have taken your first steps into the bright future of computing! This document is intended to guide you on those first steps. It has been translated from the original Russian.

As a Turing complete device, the capabilities of the SK-702 are quite literally limitless. However, this document is necessarily limited, being printed on a finite amount of inert paper. So you will no doubt have questions that are not answered herein. Please mail all such questions to Матрёша Полупроводник (Matryoshka Semiconductor).

Матрёша Полупроводник
пр. Вернадского, 7
Москва, Россия
119296

Of course, the ultimate teaching device for the SK-702 is the SK-702 itself. If you prefer a more interactive learning experience, try the popular EduQuest software. The recently released version 3.0 of this program has a didactic module specifically designed for, and ported to, the SK-702 architecture.

Otherwise, there's no better way to learn than to simply dive in and start programming. If you are unsure about some detail of an instruction's operation, try it out and see what happens.

At Matryoshka Semiconductor, we are excited to welcome you to the future of computing. Good luck on your journey!

# Architecture

SK is short for Стековой компьютер (Stekovoi Kompyuter), or Stack Computer. The name is quite apt, as the beating heart of your SK-702 is its stack memory. This innovative architecture simplifies the internal construction of the device, allowing it to be drastically smaller than its competitors. In fact, the SK-702 is Matryoshka Semiconductor's smallest and lightest model ever, barely larger than an upright piano. Truly a modern marvel!

***Technical detail:*** *Astute readers may be aware that pure stack machines are not Turing complete. Rest assured that Matryoshka Semiconductor has included a few instructions that bend the rules a little bit, and allow the entire SK line of computers to be fully Turing complete. At least in the sense that all real-world computers are Turing complete, where we ignore the fact that physical computers necessarily have a limited memory.*

Imagine a deck of cards. The rules are you can take the top card from the deck and read it, or you can add another card to the top of the deck, but you're not allowed to take cards from the middle of the deck (you can't even look at them!). In technical jargon, taking the top card is called popping from the stack, and adding a card to the top is called pushing to the stack.

Most of the instructions the SK-702 can execute are concerned with manipulating the stack in this way. Take the ADD instruction as an example, which adds two values together. First, it pops the top value off the stack, say 12, then it pops another value, say 34, then it adds those values together, 46, and pushes the result back onto the stack.

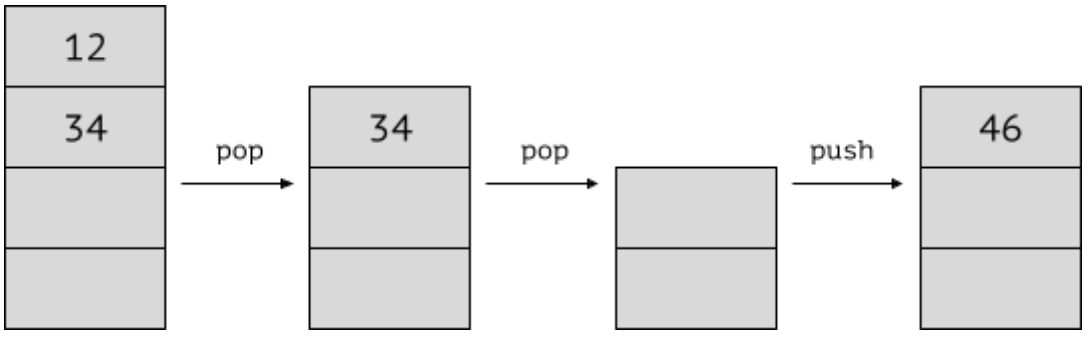


Figure 1: How the ADD instruction manipulates the stack

By careful manipulation of the stack, virtually any computation can be performed. Knowing what's in your stack at each step of the program is the key to writing bug free code.

Each stack slot stores a signed 8-bit integer. In less technical language, stack values can be any whole number from -127 to 127. There's also a special value called NAN. NAN is an error value that is used in various ways, for example if you try to divide by zero, or pull data from a port that doesn't have any more data.

***Technical detail:*** *Stack slots can also contain program addresses. This feature enables subroutines, and will be explained in more detail in the Advanced Control Flow section.*

The SK-702 has a comparatively large memory, and can hold up to 128 values in its stack. This should be more than enough for any practical computation.

# Program Structure

Like most modern computers, the SK-702 executes its instructions sequentially, one at a time, from top to bottom (with a few exceptions that will be explained later). The SK-702 is programmed using ordinary text (the future era has no need of punch-cards!), and that text is processed one line at a time.

Every instruction is 3 letters long, and must appear on its own line (you can't put two instructions on the same line). All instructions are uppercase, and case sensitive. Some instructions, like PSH (push), take an argument. The argument (or arg for short) must be written on the same line, and be separated from the instruction by a space. Let's look at an example.

```
PSH 34
PSH 12
ADD
```
Example 1

Example 1 shows a very simple program containing 3 instructions. First, a PSH instruction pushes a 34 to the stack, then another PSH pushes a 12, then the ADD instruction adds them together (as shown earlier in figure 1). Once the SK-702 has no more instructions to execute, it simply stops. At the end of this program, the stack contains only 46.

Not all lines of the program are instructions. Labels are another critically important element. They are used to give a name to a particular location in the program. Certain instructions, like JMP, can cause the execution to jump to a label, instead of continuing ordinary sequential execution.

Unlike instructions, labels can be longer than 3 letters. They can contain letters (upper or lower case), numbers, or underscores. Labels are case sensitive, and their name must be unique. When defining the label, write a line containing the name, followed by a colon character.

```
this_is_a_label:
PSH 34
PSH 12
ADD
JMP this_is_a_label
```
Example 2

Example 2 shows a simple loop. The JMP instruction takes a label as an argument, and causes the execution to return to the line where the label was defined. In this case, the program performs the same

addition as example 1, then jumps back to the start and repeats the operation.

***Technical detail:*** *Example 2 also shows a bug that can occur in stack machines. Each time the loop repeats, another 46 is added to the stack. The stack will grow and grow until it hits the limit. This is called a stack overflow. It is usually a mistake for the program to return to the same place, with a differently sized stack.*

As well as instructions and labels, a program can also contain comments. Comments have no functional effect, but can be useful to explain your code to other programmers (or to your future self!). Comments begin with a hash character, and the SK-702 will simply ignore the rest of the line. Empty lines are also ignored, so feel free to space out sections of your code to improve the clarity.

```
# This is a comment.
PSH 34
PSH 12  # This is also a comment.
ADD
```
Example 3

***Technical detail:*** *Many programmers compete to have the shortest or fastest programs. For the purposes of these competitions, instructions are all that matter. Labels and comments do not contribute to program size, and do not take time to execute (indeed, they are not executed at all).*

# Instruction Set

This section gives a complete list of instructions available on your SK-702. Each instruction's documentation begins with a short summary line like this:

INS arg [-x, +y]

This shows the instruction's name, what type of argument it takes, if any, and how it manipulates the stack. -x means it pops x values off the stack, and +y means it adds y values to the stack. If an instruction does both pops and pushes, all the pops happen before the pushes.

## Stack Manipulation

The basic stack manipulation instructions.

### Push

PSH value [0, +1]

Pushes a literal value to the stack. Takes one numeric arg, from -127 to 127, or NAN.

### Pop

POP [-1, 0]

Pops a value from the stack. Takes no argument.

## Unary Arithmetic

Unary arithmetic instructions consume one value from the stack, and produce one value. They take no arguments.

### Negate

NEG [-1, +1]

Pops the top value from the stack, negates it, then pushes it back to the stack. The negative of NAN is NAN.

### Increment

INC [-1, +1]

Pops the top value from the stack, increments it (adds 1), then pushes it back to the stack. Incrementing NAN yields NAN. Incrementing 127 exceeds the value limit, so also yields NAN.

### Decrement

`DEC [-1, +1]`

Pops the top value from the stack, decrements it (subtracts 1), then pushes it back to the stack. Decrementing NAN yields NAN. Decrementing -127 exceeds the value limit, so also yields NAN.

## Binary Arithmetic

Binary arithmetic instructions consume two values from the stack, and produce one value. They take no arguments.

### Add

`ADD [-2, +1]`

Pops two values from the stack, adds them, then pushes the result to the stack. If either input is NAN, the result is NAN. If the result of the addition exceeds -127 or 127, this also yields NAN.

### Subtract

`SUB [-2, +1]`

Pops two values from the stack, subtracts the top value from the second value, then pushes the result to the stack. If either input is NAN, the result is NAN. If the result of the subtraction exceeds -127 or 127, this also yields NAN.

### Multiply

`MUL [-2, +1]`

Pops two values from the stack, multiplies them, then pushes the result to the stack. If either input is NAN, the result is NAN. If the result of the multiplication exceeds -127 or 127, this also yields NAN.

### Divide

`DIV [-2, +1]`

Pops two values from the stack, divides the second value by the top value, then pushes the result to the stack. Since the SK-702 only deals with integers, a division that would result in a fraction is rounded down to the nearest integer less than the true result. For example, 7 divided by 2 is 3. If either input is NAN, the result is NAN. If the top value is 0, this also yields NAN.

### Modulo

`MOD [-2, +1]`

Pops two values from the stack, calculates the second value modulo the top value, then pushes the result to the stack. The modulo is also known as the remainder: x modulo y is the remainder of x divided by y. For example, 7 modulo 2 is 1. If either input is NAN, the result is NAN. If the top value is 0, this also yields NAN.

***Technical detail:*** *DIV and MOD satisfy this invariant:*
*let d = x DIV y*
*and m = x MOD y*
*then x = dy + m*
*and m ∈ [0, y) if y > 0, or (y, 0] if y < 0*

### Compare

`CMP [-2, +1]`

Pops two values from the stack, compares them, then pushes the result to the stack. If the top value is greater than the second value, the result is 1. If the top value is less than the second value, the result is -1. If they are equal, the result is 0. If either input is NAN, the result is NAN.

## Communication

The SK-702 communicates with the outside world using ports. Without ports, computing anything would be rather pointless. Ports can be connected to just about anything. For example, you could connect your SK-702 to your coffee machine, and write a program to brew the perfect cup. Or you could connect it to an array of temperature and humidity sensors and compute tomorrow's weather. Some researchers believe it may even be possible to connect multiple computing devices together into some sort of world-wide web, but this is unlikely to have any practical benefit.

The communication instructions take a port ID argument. Each port ID corresponds to one of the plugs on the back of the device. There are 64 ports in total, with IDs ranging from 0 to 63, though it's unlikely that you'll need to use all of them. Depending on what you plug into the port, the port may be read-only, or write-only, or read-write.

### Send

`SND port [-1, 0]`

Pops a value from the stack and sends it to the given port.

### Receive

`RCV port [0, +1]`

Receives a value from the given port and pushes it to the stack. If there is no more data to receive, NAN is pushed to the stack instead.

## Control Flow

These instructions affect how the execution flows around the program. They all take a label argument, and jump to that label if their condition is satisfied.

### Jump

`JMP label [0, 0]`

Unconditionally jumps to the given label.

### Jump if greater than

`JGT label [-1, 0]`

Pops a value from the stack and jumps to the given label if that value is greater than 0.

### Jump if less than

`JLT label [-1, 0]`

Pops a value from the stack and jumps to the given label if that value is less than 0.

### Jump if equal

`JEQ label [-1, 0]`

Pops a value from the stack and jumps to the given label if that value is equal to 0.

### Jump if equal to NAN

`JEN label [-1, 0]`

Pops a value from the stack and jumps to the given label if that value is equal to NAN.

# Advanced Stack Manipulation

Earlier in this document it was stated that the SK-702's stack architecture means that the only way to read data is to pop it off the stack, and the only way to store data is to push it to the top of the stack. This was a modest simplification in order to facilitate pedagogical expediency (aka a lie).

The SK-702 is not a pure stack machine. The following instructions bend the rules of the ordinary stack architecture in order to make your job as a programmer easier. Matryoshka Semiconductor humbly hopes that the remarkable convenience of these tools makes up for the earlier untruth.

Some of these instructions take a stack index argument, and the others pop a stack index from the stack. Stack indices start at 0 for the top slot, up to a maximum of 127 (the bottom slot in the unlikely event that the stack is totally full).

## Copy

`CPY stack_index [0, +1]`

Copies the value at the given stack index, and pushes it to the stack.

## Swap

`SWP stack_index [0, 0]`

Swaps the value in the top slot with the value at the given stack index.

## Copy N-th

`CPN [-1, +1]`

Pops a value from the stack and interprets it as a stack index, then copies the value at that stack index, and pushes it to the stack.

## Swap N-th

`SWN [-1, 0]`

Pops a value from the stack and interprets it as a stack index, then swaps the value in the top slot with the value at that stack index.

***Technical detail:*** *The CPN and SWN instructions index into the stack after popping the top value. So if the top stack slot is 0, and the second slot is x, then CPN will pop the 0, and effectively do a CPY*

*0 instruction, copying the x to the top of the stack. Afterwards the top slot and the second slot will both be x.*

## Advanced Arithmetic

These instructions, also known as bitwise operators, work directly with the bits of the values, rather than the numbers they represent. So it is necessary to give a brief outline of how the numbers are encoded in binary. If in doubt, the debugger has a mode to view the values in the stack and ports in binary rather than decimal.

Each stack slot is made up of 8 bits. The numbers from 0 to 127 are simply encoded using their binary representation.

```
decimal    128 64 32 16  8  4  2  1
      0  =   0  0  0  0  0  0  0  0
      1  =   0  0  0  0  0  0  0  1
      2  =   0  0  0  0  0  0  1  0
      3  =   0  0  0  0  0  0  1  1
      4  =   0  0  0  0  0  1  0  0
         ...
    127  =   0  1  1  1  1  1  1  1
```

To find the bit pattern of a negative number, from -1 to -127, find the positive number, flip its bits, then add one. This is known as two's complement representation. For example, 3 is 00000011, so to find -3 we flip the bits, 11111100, and add 1, 11111101. Note that all negative values begin with a 1, and all positive numbers begin with a 0.

```
decimal    128 64 32 16  8  4  2  1
     -1  =   1  1  1  1  1  1  1  1
     -2  =   1  1  1  1  1  1  1  0
     -3  =   1  1  1  1  1  1  0  1
     -4  =   1  1  1  1  1  1  0  0
         ...
   -127  =   1  0  0  0  0  0  0  1
```

The final bit pattern, 10000000, represents NAN. The way the bitwise instructions handle NAN is a bit different to the other arithmetic instructions. In ordinary arithmetic, if any of the inputs are NAN, the result is NAN. However, for the bitwise instructions, NAN is just another bit pattern with no special meaning.

### Not

NOT [-1, +1]

Pops the top value from the stack, flips all its bits, and pushes the result to the stack.

NOT 01010111 = 10101000

## And

AND [-2, +1]

Pops the top two values from the stack, "and"s together every bit, and pushes the result to the stack. That is, for each pair of bits, if both are 1 the result bit is 1, otherwise the result bit is 0.

00110101 AND 01010111 = 00010101

## Or

ORR [-2, +1]

Pops the top two values from the stack, "or"s together every bit, and pushes the result to the stack. That is, for each pair of bits, if both are 0 the result bit is 0, otherwise the result bit is 1.

00110101 ORR 01010111 = 01110111

Note that the instruction is called ORR, rather than OR. This quirk is because every instruction must be exactly 3 letters long.

## Xor

XOR [-2, +1]

Xor is short for exclusive or. Think of "a xor b" as "a or b, but not both". This instruction pops the top two values from the stack, "xor"s together every bit, and pushes the result to the stack. That is, for each pair of bits, if the bits are different the result bit is 1, otherwise the result bit is 0.

00110101 XOR 01010111 = 01100010

## Left shift

LSH [-2, +1]

Pops the top two values from the stack, shifts the bit pattern of the second slot to the left by the value of the top slot, and pushes the result to the stack. If the shift value is negative, the bit pattern is shifted to the right instead. Bits that fall off the left or right of the value are simply dropped. Shifting by anything beyond 7 or -7 will always yield 0.

```
00110101 LSH  1 = 01101010
00110101 LSH  3 = 10101000
00110101 LSH -4 = 00000011
```

Another way of thinking about this instruction is that it multiplies or divides by a power of two. That is, x LSH y multiplies x by $2^y$.

Note that unlike the other bitwise instructions, left shift only cares about the bit representation of the second stack slot. The top stack slot is treated as an ordinary value. For this reason, shifting by NAN yields NAN, but NAN itself can be shifted (shifting 10000000 by -2 yields 00100000).

***Technical detail:*** *The LSH instruction does not perform sign extension. So shifting a negative value by a negative amount will result in a positive value, as the leading bits will be zeros.*

## Advanced Control Flow

The following instructions extend the ordinary jump instructions by storing program addresses on the stack. Program addresses take up a single stack slot, just like a value. However, it is an error to try to use an address as a value. Most instructions that consume slots from the stack only accept values (eg ADD, NEG, INC).

However, the stack manipulation instructions don't actually inspect the value of the stack slot, so can move slots around whether they are values or program addresses: POP, CPY, SWP, CPN, SWN.

The two instructions in this section are primarily used to jump into and out of subroutines. A subroutine is simply a section of code that is intended to be reused. If you find yourself writing the same piece of code multiple times in your program, it might save some instructions if you pull them out into a subroutine:

```
my_subroutine:
# Put whatever instructions you need in here.
# Once your subroutine is done, use the RET
# instruction to return to the call site.
RET

# Call your subroutine using JSR.
JSR my_subroutine
# The RET instruction will take us back here
# once the subroutine is finished.
```

### Jump to subroutine

`JSR label [0, +1]`

Pushes the current program address to the stack, then unconditionally jumps to the given label. This is used to jump into a subroutine.

### Return from subroutine

`RET [-1, 0]`

Pops a program address from the stack and jumps to it. This is used to return from a subroutine, back to the JSR instruction it came from.

## Miscellaneous Instructions

### Stack size

`STK [0, +1]`

Pushes the current stack size to the stack. For example, if the stack is empty, a 0 is pushed to the stack.

### Halt

`HLT [0, 0]`

When the SK-702 reaches this instruction, it stops execution. Most programs end by simply reaching the bottom of the code and running out of instructions, but if you need to end the program in the middle of the code, use HLT.

### Breakpoint

`BRK [0, 0]`

When the SK-702 reaches this instruction, it pauses execution so that the program state can be examined by the programmer. This is similar to the HLT instruction, but allows execution to resume once the inspection is complete. This instruction is very useful for debugging.

### Bell

`BEL [0, 0]`

Plays a beep sound. This instruction is occasionally useful for debugging.

## No-op

`NOP [0, 0]`

NOP is short for no-op, which is short for no operation. This instruction does nothing. There is no practical reason to use this instruction. It is included for historical reasons.

***Technical detail:*** *In other programming environments, no-ops are used to introduce delays into the program. This is more efficient than doing busy-work like repeatedly adding 0, because the CPU is typically designed to enter a low power state when executing a no-op. However, since the SK-702 has blocking I/O, the author can't see any reason to add these sorts of delays to your code.*

# Errors

There are two different kinds of errors that can occur in your programs: compile errors, and runtime errors. Compile errors are typos in your program that are caught before your program even begins running, such as misspelling an instruction. Runtime errors are those that occur while your program is executing, such as stack overflows. This section explains all of these errors, and how to fix them.

## Compile Errors

### Invalid instruction

```
PSH 123   # Valid
PSK 123   # Misspelt instruction
psh 123   # Must be uppercase
PUSH 123  # Must be 3 characters long
```

You've typed an invalid instruction. All instructions are 3 uppercase letters. Check the spelling of the instruction. It's also possible that you were trying to write a comment, but forgot the # at the start, or trying to define a label but forgot the : at the end.

### Disabled instruction

```
MUL  # Usually valid, but disabled during certain tasks
```

You've typed a valid instruction, but it is disabled at the moment. Certain tasks disable some instructions to increase the difficulty. For example, the EduQuest software sometimes disables instructions to force you to think outside the box. Check the task description to see what instructions are disabled, and try to think of another way of solving the problem.

### Too many arguments / Not enough arguments

```
PSH 123      # Valid
PSH          # Not enough arguments
PSH 123 456  # Too many arguments
ADD 123      # Too many arguments
```

You've given the instruction either too many or not enough arguments. All instructions expect a specific number of arguments, either zero or one. Check the documentation of the instruction for details.

## Expected a number

```
PSH 123  # Valid
PSH NAN  # Valid
PSH abc  # Expected a number
PSH nan  # Expected a number
```

You've typed an invalid number. Numbers must only contain digits (0 to 9) except for a possible - at the start, or be NAN (case sensitive). Check the spelling of the number.

## Need -127 to 127, or NAN

```
PSH 123   # Valid
PSH NAN   # Valid
PSH 999   # Too large
PSH -999  # Too negative
```

You've tried to pass a number outside the instruction's range. Specifically the "value" range: from -127 to 127 (inclusive), or NAN. This range includes all the possible values that can be stored in the SK-702's stack. The PSH instruction is the only one that expects this range. Check for typos in the number.

## Need 0 to 127

```
CPY 123  # Valid
CPY 999  # Too large
CPY -3   # Negatives are not allowed
CPY NAN  # NAN is not allowed
```

You've tried to pass a number outside the instruction's range. Specifically the "stack_index" range: from 0 to 127 (inclusive). This range includes all the valid stack indexes, with 0 being the top of the stack, and 127 being the bottom of the stack when it is totally full. This range is used by the CPY and SWP instructions to refer to the stack slot they operate on. Check for typos in the number.

## Invalid send port ID / Invalid receive port ID

```
SND 0    # Valid
SND 999  # Too large
SND -3   # Negatives are not allowed
SND NAN  # NAN is not allowed
SND 47   # Ports may be disabled for the current task
```

You've tried to use a invalid port ID. Valid port IDs are always between 0 and 63 (inclusive), but even ports inside that range may be invalid. Some ports are read-only or write-only, and certain

ports can be disabled in certain contexts. For example, the EduQuest software disables all but a few ports, to ensure that you don't interfere with important hardware while you're learning. Check the task description to see what ports are available.

### Invalid label

```
my_label:  # Valid
my label:  # Labels can't contain spaces
my-label:  # Labels can't contain invalid characters
:          # Labels can't be empty
```

You've defined an invalid label. Labels must use 1 or more characters, all of which are either an upper or lowercase letter, or a number, or an underscore. Check that you're not using invalid characters.

### Duplicate label

```
my_label:  # Valid
my_label:  # Duplicate label
```

You've defined the same label more than once. Labels must be unique throughout your program. Try renaming the label to something unique.

### Undefined label

```
my_label:     # Valid
JMP my_lable  # Misspelt label
```

You've referenced a label that doesn't exist (for example, in a JMP instruction). Check the spelling of the label (they're case sensitive).

### Too many instructions

This error simply means that your program is too long for the SK-702 to run. It is very unlikely that you'll ever see this error in a real program, as the limit is several thousand instructions.

## Runtime Errors

TODO

# Quick Reference

TODO: Single page reference sheet containing all instructions and error states

Other TODOs

- Every example needs a label, and a box around it
- Background design for cover
- Page design for other pages
- Styling for headings and subheadings
- Page numbers
- Table of contents