1V0

Table of Contents

I. Outline of 1V0

1V0 lets you "feel a phantastic early mainframe".

It is licensed by its creater, Nino Ivanov, under the GNU AFFERO GENERAL PUBLIC LICENSE, Version 3, 19 November 2007, https://www.gnu.org/licenses/agpl-3.0.en.html.

The system is a sort of "assembler-like interpreter" on a virtual machine and not unlike a "BASIC interpreter with immediate mode" or a "Lisp interpreter". The 1V0 system (pronounced like the Bulgarian name "Ivo") is inspired a bit by the idea of a "mainframe" of the late 1940s or early 1950s. At the time, computers were rather "personal" - without any "time-sharing" and other complication - and "immediate" - your really computed with _numbers_, not strings, objects, etc., and you did not too fancy I/O. Often, these computers had large or variable bit-lengths, for as primordial as they were, they were supposed to let you do real engineering or mathematical work. Thus, here, you set the numbers into a machine, let it operate, and get some result back - which might not even be "printed" but rather just stored at certain memory locations. The aim of the system is to make possible to conduct interactive algorithmic computation on nearly ANY hardware. This makes it possible to use basically ANYTHING to devise algorithms, which, due to their centricity on numbers, should with ease be useable anywhere at all, and portable to any other programming language with relative ease. Thus, 1V0 may help any hobbyist turn his Arduino into a "computer", and facilitate algorithmic experimentation also on old mobile phones, which might, in turn, be helpful to advance understanding of computation for children, the poor, and people in developing countries. 1V0 is presented in C (and this is the reference implementation), in Java (for "big" machines), as a Java midlet (for old cellphones) and for Arduino-styled microcontrollers, tested on Arduino UNO, Arduino MEGA2560, Arduino DUE (that should work on ARM in general; unfortunately, a reset always clears the instruction memory due to lack of EEPROM emulation), Digispark TINY85 and Wemos D1 (ESP8266).

"Phantastic" refers to the fact that it is inspired by thelikes of early IBM, LGP, UNIVAC and similar machines, and in the way results are stored after operations it is inspired by Intel 8080 or 8086 chips, but it does not imitate anything specifically; instead, it is a purely original creation.

II. Design principles

From this philosophy, several design principles evolved, which make it possible to let 1V0 to be extremely portable and operate on very, very small machines like an ATTiny85-chip with 512 byte SRAM and 512 byte EEPROM as well as on possibly any mobile phone that would run a Java MIDP 1.0 midlet:

a. - Instructions and code are separated: Particularly on microcontrollers, this seems to be an advantage. Mixing code and data sounds interesting in theory, but only causes chaos in practice - as data has a good chance of overwriting instructions, and instructions have a good chance of being misread as data. Keeping these two worlds separate turns out beneficial for debugging. And it makes it possible to place them in different memory locations - particularly, at least the instructions into EEPROM (and flash, if it emulates EEPROM). So there exists an "instruction memory" and a "data memory". (The terms "instruction", "operation" and "command" are used herein somewhat interchangeably, but more precisely, an operation is a part of an instruction - and an instruction with address is a command. Apart from an operation, it also contains the operand data addresses.) - The decision to store instructions and not data in EEPROM was made on the basis that re-programming does not occur as often (done manually by the user) as changing data (done automatically during execution by the machine).

b. - Computation, not "fiddling", and from here, "large numbers" and "non-integers": 1V0 has only one data type - the floating (or fixed-point, depending on the implementation) number. - When dealing with numbers, one is usually interested in quickly computing floats of non-integer nature or at least fixed-point numbers, and surely one would not like to be limited by only integer numbers of a maximum size of 255. That is why the only data type is the "float" (whereby next to floating point numbers also double precision implementations or fixed-point numbers are to be understood). Floats are set into the system, computations are done on floats or ranges of floats, and floats are returned.

c. - Every instruction is n bytes long (n typically is 4). Each data location is n bytes long (n is the length of the float). Every instruction is set by giving (i) its address in the instruction memory, (ii) the operation that shall be performed when reaching it, (iii) a first datum, expressed by its location in the data memory (usually where the result will be stored, too) and (iv) a second datum. Every datum is set by saying in what

location what value shall be held, and the value is itself a float.

d. - All input and output is given and read as numbers; moreover, all instructions are given and read as numbers. This makes operation by nothing but a numeric keypad and a numeric display possible. (Exceptions to this exist, but the exceptions can be simply deleted from the code - they more serve the amenity of use rather than core functionality.) This makes operation on very low-end devices - like old cell-phones with a numeric keypad - a lot easier than having to write out long-winded commands (as is the difficulty in using a BASIC-interpreter on a feature-phone).

e. - "Goto is good": A conditional goto, which jumps to a chosen instruction location if a chosen data location's value is zero (or at least a very, very small number) is the single branching and decision-making operator. Unconditional jumps are performed obviously by making sure the location is indeed holding a zero. Goto is at the heart of most computations, and its purity is rather useful in constrained memory conditions, where fidgeting with loops and structures may feel more unwieldy.

f. - The code is "personal": 1V0 is not made for large, collaborative endeavours, but rather, for little, personal one-off programs. These should be operational quickly - if for simple things the user has to think too much, the task will rather be done by hand with a calculator.

g. - "Fail late": It is better for a little, one-off program to "perform", even if the user let it have "bugs", rather than have crystal-clear perfection before anything of use is accomplished. "Fail late" programs naturally make it more difficult to create "large" programs, because bugs will be "tolerated" a long time and "best effort" solutions will be presented, instead of alerting the user of his mis-design. But this makes fast advance in the beginning possible - and as programs are small and personal, that is the more important accept. - This is e.g. expressed in the fact that an unknown instruction number will be interpreted as "no operation" (skipping it instead of terminating), that the square root of negative numbers is given as the square root of their initial absolute value with the sign attached, that a division by zero or a natural logarithm of zero gives zero (which facilitates jumps), and similar design decisions. (A "fail early" type of language, in contrast, is Haskell: by the time you let your Haskell program compile, you can have only pragmatic errors, i.e. deep conceptual misunderstandings, as all syntactic and semantic - due to the type checking - errors will be eliminated. Then again, Haskell - as opposed to Lisp - is not exactly flexible to use.)

h. - CISC, not RISC: For a human being, it is better to have "many operations at hand", as here, the code will rather directly

implement the SEMANTIC idea of the solution of a given problem, rather than the SYNTACTIC idea to puzzle together all sorts of micro-operations, for long stretches of code without any evident purpose. Hence, having MANY instructions (64, but you can implement sensibly also 256 or any other number) is preferred.

i. - Operations on memory locations and number ranges, not on "variables" and "loops": "normal people" use LibreOffice Calc, Microsoft Excel or similar tools to satisfy their larger computing needs - there, data is placed into columns, and the columns are computed upon, in a most general way. "Computing with columns" is somehow mankind's most primordial way of doing computations. What matters to people are numbers in places, and possibly series of such numbers at such consecutive locations. "Normal people" do not really want to "name" variables, "dimension" arrays or "do loops" on them. - Hence, 1V0 focuses on data locations and ranges of such locations as such. Whether a single number or a range will be used depends solely on the specific operation applied (one of up to 64 in the reference implementation). All the numbers are just at their memory locations, waiting for something to be done with these locations.

j. - Saving your work: As mentioned, 1V0 stores on microcontrollers with EEPROM (or an emulation thereof) the instructions in EEPROM. So when you turn it off and on again, your program is preserved.

k. - Tracing your program: 1V0 has a trace facility, by default being off. If you turn it on, it will print on console each operation it is executing. This allows you to follow the program flow.

l. - The solution to the Halting Problem: Alan Turing famously formulated that you cannot predict "how long a program will run" without access to its source code, because, quite simply put, some joker may have placed a "10 GOTO 10" somewhere. - To solve this, 1V0 features a run limit: after a pre-set number of operations, 1V0 stops and informs the user that the run limit has been exhausted. A run limit of 0 would allow an infinite run (i.e. lifts the restriction), but is not the default. If your program does something "infinitely", it is doing either something very smart or something very stupid, and more often than not, it is the second thing. This is not like "10 GOTO 10" in BASIC, where the user can press some sort of interruption, because particularly on some weak hardware, such interruptions may not be available.

m. - No sane person "counts from 0".

n. - Limits of ranges should be inclusive.

While the focus has been on a word-size of 32 bit, also 64-bit-sizes etc. are possible.

III. General principles of operation

The system is operated through a serial port or similar device and has two modes of operation, an "order acceptance mode" and an "execution mode". Instructions are stored in EEPROM where available (otherwise in RAM), data are stored in RAM.

The "order acceptance mode" is the "face" of the system to the user. There, the user enters operations and data. This mode is fully interactive, and the system does not compute. It is merely serving the "organisation" and a "preparation" of a later computation. (Adjusting your paper tape, relay memory and blinkenlights, if you so will.) The default order acceptance operation is to tell four things: the location of a command in the command memory (like a line number in BASIC or FORTRAN), the operation which therein shall be stored, the first data address this operation will operate upon, and the second data address this operation will operate upon. (What, SPECIFICALLY, is done, depends on the individual operation in question).

Mind you, you operate with data ADDRESSES there, and NOT immediately with data - i.e. when you want to compute 1 + 2 = 3, you might instruct 12 (command address), 9 (plus), 16 (data address 16 which contains a 1), 53 (where the 2 is contained), and when the operation is performed, data address 16 (the first operand) will contained the result. (This is evidently inspired by ADD AX,BX.)

Larger platforms invite you to also give a comment, as a fifth argument. You may press there * [ENTER]. If your comment contains a "#"-sign, the command will be cancelled and instead a no-operation will be performed. This is a safety measure. The comment itself is not saved, it is only printed on the console. On the other hand, particularly when working with microcontrollers, even this ephemeral documentation may greatly improve your overview over what you are actually doing, as you may "scroll up" in your terminal emulator. - Note, there is generally no such "safety" when you work with order acceptance instructions (negative command addresses) - these interactions are immediate.

Positive command addresses invite for further command entry in the described way.

Negative command addresses - as such are impossible - will instead trigger certain setup prompts, and specifically, -1 to list and instruction range (not unlike "LIST" in BASIC), -2 to list a data range (this "raw" examination of memory contents is indeed the most comfortable way of handling "output" - you just "look what it did"), -3 to enter instructions as "raw numbers" (see below; this is not very useful and practically useless in the default implementation, but may be adjusted to read instead instructions form some input stream or file system), -4 to enter data as "raw numers" (that turns out to be very comfortable in practice, and in cellphone Java midlets, it is the ONLY way to enter data), -5 to zero out (clear) an instruction range (in microcontrollers that is necessary as EEPROM often contains each byte a value of 255 and not of 0), -6 to clear a data range, -7 to turn on tracing, -8 to turn off tracing, -9 to set the run limit (particularly if you need an infinite run you use that to set it to 0), and -10 to exit the 1V0 system, on platforms where such exit is sensible (not on microcontrollers: exit to WHAT?). Not implemented commands are simply ignored.

A command address of 0 means the next operation will be immediately executed. No "program" will run, just one single operation. This is how you can use 1V0 "calculator style". After execution, 1V0 will return immediately to the order acceptance mode, UNLESS the operation was a jump to a higher address (see below).

To enter execution mode, the user can thus enter 0 (do immediately), 1 (jump), 0 (if datum 0 is 0 - which it is), 1 (jump to command address 1 - or any other, this could be 1412 or any other). While the system operates, it is NOT interactive. It executes "in a batch" all the instructions that are given. (Exceptions to this are commands for data input and data output during run, which, however, are not implemented in all platforms). Thereby, the system basically executes instruction after instruction, and branches as required, until it terminates or the run limit is exhausted, which often is set to five times the data memory size. (So if you have an "infinite" loop, just patiently wait a moment.) Once execution mode terminates, the system again returns to order acceptance mode - not unlike a Lisp-, BASIC- or Python-interpreter.

Every instruction is composed of 32 bits in the default implementation: 6 highest bits for the operation (hence the 64 instructions), 13 bits for the address of the first operand, 13 bits for the address of the second operand. Instruction 0 is reserved for the immediate operations mode. Datum 0 is reserved and should always be 0. It is used as a sort of "black hole" and is not supposed to hold user data. (In practice, its use is not unlike that of /dev/null in Unixoids.)

Thus, the system can perform, in its reference implementation, any number of instructions for up to 8191 pieces of data, though mostly, the full data range is not used (not enough memory on the smaller microcontrollers). The commands are not "explicitly" limited by the data positions, but as the jump instruction supplies a data address as command address, there is an implicit limitation to the same range (which can be circumvented by the operation IADR, see below, but that is not "pleasant" for the user).

The system operates by fetching integers, in which the operation together with the operands is stored, extracting the operation to be performed and the operands to be subjected to it, executing the operation, and advancing to the next instruction in the instruction memory. (And as data are stored elsewhere, instructions do not "care" how big each datum is.) Making instructions "occupy one integer" thus makes encoding, decoding and executing instructions extremely straightforward.

In various platforms, various of the operations may be omitted; to see the capability of a speficic platform, best look at its respective source code. (The system is VERY easily adjustable: you literally just change little sections of code under if-then-else-clauses which decide what exactly shall be done for any given command.) Unknown operations will simply be skipped. - The design decision to simply skip unknown operations eases compatibility between platforms. - One platform is a little different, namely the ATTiny85: As space was extremely tight there anyway, instructions are instead encoded in 16-bit integers, with only 12 operations possible on only 63 data positions (apart from position 0). (It was NOT lack of SRAM, but limited flash space (!) that really made things difficult to implement there.) Instructions have been by default artificially limited to 63, too (the range of "normal" jumps there), though obviously, all in all four times as many would be possible in 512 byte EEPROM (if the user gets accustomed to resorting to IADR, see below).

Whenever limits are supplied for any range, be it in order acceptance mode or order execution mode, the limits are considered inclusive. That is, 1 to 5 is really 1,2,3,4,5, and not as in other languages e.g. excluding 5.

IV. Specific operations: order acceptance mode

You are invited to enter a command address. Depending on this, further interactions take place. After each interaction, you are again invited to enter a command address, unless you exit the system by entering a command address of -10 (not available on all platforms, particularly microcontrollers). The exact prompts for each interaction shown differ from platform to platform; here, the reference system in C is used for the demonstration. On most platforms, the prompt for a new interaction appears immediately, only on cellphones you have to "click your way on" with buttons, as the screen is too small to rely on continuous printing interaction.

a. - Entering a positive number

This is taken as the command address. There, an integer is to be recoreded which represents the operation to be conducted together with the operands. You are then asked for the operand; and then for each of the addresses of the two data operands. Finally, you are asked for a comment. (Comments on Java should end with an asterisk. In C, this is just "nice to do".)

An example interaction may look like this, though on simpler platforms, the prompts may be shortened and comments may not be available.

```
CMD ADDRESS: 27
OPERATION  : 12
DATA ADDR 1: 86
DATA ADDR 2: 86
CMNT (OPT) : Turn data at 86 to 1 by dividing it by itself. *
00027 DIVS 0086>+0.000000E+00>+0.000000E+00
0086>+0.000000E+00>+0.000000E+00
```

This means, for command address 27 we want a division (12), where the datum at location 86 divides itself. The final line is a status report, what command has just been accepted. (It is also a nice visual separator between commands, which otherwise may become hard to follow.) Each datum is printed in the following fashion:

```
0086>+0.000000E+00>+0.000000E+00
```

This is rooted in the idea that data can also hold addresses. (Addresses are either rounded or taken as the integer part of the float.) Here, it is shown "the first datum address is 86, and and

it holds a value of 0, and if that value were itself an address, it would point to another value, which happens to be also 0". If you had at data address 11 a value of 1, and at data address 1 a value of 57, all others being zero, then

a print of "datum address 11" would look like

0011>+1.000000E+00>+5.700000E+01

and a print of datum 1 would look like

0001>+5.700000E+01>+0.000000E+00

This property of "interpreting data also as pointers" is useful if data has been pre-entered with the command -4, and particularly, if data shall be used to hold addresses. Negative values are in this regard, as pointers, evaluated according to their absolute value.

A command address can be overwritten at will. You do not necessarily have to enter command addresses consecutively. And given that instruction 0 is seen as "no operation", you do not even have to enter commands contiguously, and so you can "leave space" between commands, if desired. The command is merely entered, but not executed.

b. - Entering zero

Works exactly as above, however, the command is immediately transferred to execution mode and executed. You return to order acceptance mode only after the execution.

The "results" of the command, i.e. any changes to data memory, are NOT automatically shown. They are shown only if data memory is inspected manually or if during execution an output command has been given. Whether the user wishes to inspect data or not, and which data, is not part of this interaction (but see -2).

An example interaction to square a number (multiply - 11 - it with itself) might look like:

```
CMD ADDRESS: 0
OPERATION  : 11
DATA ADDR 1: 1
DATA ADDR 2: 1
CMNT (OPT) : square the datum at 1 *
00000 MULS 0001>+5.700000E+01>+0.000000E+00
0001>+5.700000E+01>+0.000000E+00
```

```
RUNLIMIT = 2001
```

The run limit indication shows how many operation cycles are left and is not of importance here, but it does show that you desired a program execution (albeit of a program with only one - because immediate - instruction).

c. - Entering -1

Show a range of instruction memory. This prints the operands and their data in the desired range.

A sample interaction may be:

```
CMD ADDRESS: -1
LIST INSTRUCTION RANGE, INCLUSIVE LIMITS.
FIRST COMMAND ADDRESS FROM WHERE ON TO LIST: 7
SECOND COMMAND ADDRESS AND TO WHERE TO LIST: 11
INSTRUCTIONS:
00007 10=MINS 0005>+0.000000E+00>+0.000000E+00
0001>+1.000000E+01>+0.000000E+00
DECIMAL REPRESENTATION: 671129601
00008 03=OUTP 0005>+0.000000E+00>+0.000000E+00
0005>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 201367557
00009 01=JUMP 0005>+0.000000E+00>+0.000000E+00
0011>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 67149835
00010 01=JUMP 0000>+0.000000E+00>+0.000000E+00
0004>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 67108868
00011 03=OUTP 0004>+0.000000E+00>+0.000000E+00
0004>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 201359364
00012 00=NOOP 0000>+0.000000E+00>+0.000000E+00
0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
00013 00=NOOP 0000>+0.000000E+00>+0.000000E+00
0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
```

The printout shows, if the platform is not too low, also a mnemonic to the instruction, for easier human comprehension. It is not a bad idea, after you enter your program, to again check whether you entered it correctly.

The "decimal representation" shows what number would have to me entered under -3 in order to give that instruction in "machine code".

A range of 0 to 0 prints all.

d. - Entering -2

Show a range of data memory:

```
LIST DATA RANGE, INCLUSIVE LIMITS.
FIRST DATA ADDRESS FROM WHERE ON TO LIST: 1
SECOND DATA ADDRESS AND TO WHERE TO LIST: 13
DATA:
0001: +3.249000E+03 +0.000000E+00 +0.000000E+00 +0.000000E+00
+0.000000E+00
0006: +0.000000E+00 +0.000000E+00 +0.000000E+00 +0.000000E+00
+0.000000E+00
0011: +1.000000E+00 +0.000000E+00 +0.000000E+00 +0.000000E+00
+0.000000E+00
```

Not all platforms display scientific notation. In fact, cellphones and some microcontrollers do not. I much prefer it, though. It looks so "early-mainframy".

Beware that on systems with fixed point, you will only see integers, and you have to discount the decimal places - on ATTiny85, 00, on MIDP 1.0 enabled cellphones, 0000. Thus, 12345 means 123.45 on an ATTiny85 and 1.2345 on a primitive cellphone.

A range of 0 to 0 again prints all.

e. - Entering -3

Enter a range of instructions as integers - while possible, this is not very useful. But that is indeed your "machine code" if you will. The value 67108863 means "No operation, each on data 8191" and is not saved.

Example:

```
CMD ADDRESS: -3
ENTER FULL INSTRUCTIONS AS DECIMAL NUMBERS, 67108863 TO TERMINATE.
START ADDRESS FROM WHERE ON TO ENTER: 37
00037: INSTRUCTION: 12345678
00038: INSTRUCTION: 90123456
```

```
00039: INSTRUCTION: 111111111
00040: INSTRUCTION: 67108863
```

This is on purpose more flexible than the next possible entry and has no pre-determined "end" as you might not yet be sure how many instructions you will thus "patch in".

f. - Entering -4

Enter a data range. Now this is straightforward, and you will grow to like it.

Example:

```
CMD ADDRESS: -4
ENTER DATA AS FLOATING POINT NUMBERS WITHIN INCLUSIVE RANGE
LIMITS.
DATA ADDRESS FROM WHERE ON TO ENTER: 6
DATA ADDRESS UNTIL WHERE TO ENTER  : 13
0006: DATUM: -57.1
0007: DATUM: -8.811
0008: DATUM: 130.6
0009: DATUM: 8.627
0010: DATUM: 0
0011: DATUM: 0
0012: DATUM: 1311
0013: DATUM: 1306
```

On systems which allow floating point, enter floats. On system which operate with fixed point, bear in mind how many positions are used as "decimal places". These are two positions on an ATTiny85 and four positions for a MIDP 1.0 midlet. Accordingly, to enter a "31" on an ATTiny85 system, enter 3100, and on a primitive cell phone, enter 310000. As long as you enter the numbers correctly, the system will itself keep track of adjustments in multiplication and division later on.

If you want to enter only one datum, supply the same value for the beginning and the end of the range.

g. - Entering -5

Clearing an instruction range, limits inclusive, is straightforward:

```
CMD ADDRESS: -5
```

```
CLEAR INSTRUCTION RANGE, INCLUSIVE LIMITS.
FIRST COMMAND ADDRESS FROM WHERE ON TO CREAR: 83
SECOND COMMAND ADDRESS AND TO WHERE TO CLEAR: 88
INSTRUCTIONS CLEARED.
```

The respective instructions are set to "no operation" and would be simply skipped upon execution.

To clear only one instruction, make both ends of the range the same. Beware, a range of 0 to 0 CLEARS EVERYTHING. This is a good idea to do on microcontrollers in order to rid the EEPROM of any previous data that might undesirably be interpreted as instructions.

h. - Entering -6

Clear a data range.

```
CMD ADDRESS: -6
CLEAR DATA RANGE, INCLUSIVE LIMITS.
FIRST DATA ADDRESS FROM WHERE ON TO CLEAR: 0
SECOND DATA ADDRESS AND TO WHERE TO CLEAR: 0
DATA CLEARED.
```

Beware, this example CLEARS ALL DATA MEMORY, as a range of 0 to 0 was chosen.

Apart from that, to clear only one datum, make both ends of the range the same.

i. - Entering -7

Turn on tracing during execution.

```
CMD ADDRESS: -7
TRACE ON
```

Results will be visible in the next execution, also for immediate commands, e.g.:

```
CMD ADDRESS: 0
OPERATION  : 9
DATA ADDR 1: 1
DATA ADDR 2: 1
CMNT (OPT) : show an immediate trace *
```

```
00000 PLUS 0001>+0.000000E+00>+0.000000E+00
0001>+0.000000E+00>+0.000000E+00
00000 09=PLUS 0001>+0.000000E+00>+0.000000E+00
0001>+0.000000E+00>+0.000000E+00
```

This shows adding zero to zero at data position 1. The last line is the trace, the line above it is just the instruction receipt confirmation. (When execution is not immediate, these will not be show next to each other as here.)

j. - Entering -8

Turn off tracing - similar to the above example:

```
CMD ADDRESS: -8
TRACE OFF
CMD ADDRESS: 0
OPERATION  : 9
DATA ADDR 1: 1
DATA ADDR 2: 1
CMNT (OPT) : same thing without tracing *
00000 PLUS 0001>+0.000000E+00>+0.000000E+00
0001>+0.000000E+00>+0.000000E+00
```

The last line of above is not shown.

k. - Entering -9

Set the run limit. Set it to 0 to abolish any run limit.

Example:

```
CMD ADDRESS: -9
SET RUNLIMIT, UP TO 65535, 0=INFINITE, TERMINATING WHEN 1: 60000
```

l. - Entering -10

Exits.

V. Specific operations: execution mode

The mnemonics are actually only used for print-outs of operators. Instead, the system really works only with the number of each operation.

Each operation at any command address has the form:

OPERATOR, FIRST-DATA-ADDRESS, SECOND-DATA-ADDRESS, or briefly, opr, datadr1 and datadr2. (Command addresses are generally omitted in this display.)

Where the first and the second data addresses specify a range, the range limits are taken to be both inclusive (unless, where specified, the range limits are defined differently).

0=NOOP

No operation. Skipped. The operands are not evaluated. Could be used to keep space free for later instructions or to temporarily inactivate an instruction and let it be skipped.

Keep in mind that the equivalent of 0 8191 8191 or decimal 67108863 is used to terminate command entry in the command -3 in order acceptance mode, so such a command could not be entered there.

Example:

0, 0, 0

Entering NOOP can be useful when you started writing a command at the wrong command address. The command there will be overwritten, but at least no data will be modified.

1=JUMP

If the value at the first data address is 0, jump to the instruction that has the same command address as the second data address. Note the irregularity versus other commands: the jump is TO THE SECOND DATUM _A_D_D_R_E_S_S_ TAKEN DIRECTLY AS COMMAND ADDRESS and NOT, as it would be regularly, TO THE _V_A_L_U_E_ HELD BY THE SECOND DATUM INTERPRETED AS COMMAND ADDRESS. This irregularity is for the sake of easier readibility - otherwise absolutely all jumps would have "computed" targets and that is

harder to comprehend to the reader. It also limits the normal jump range to the possible data address range.

Example:

1, 0, 1

"Jump, if the datum at data address 0 is 0, to the first command address and execute from there" - a common way to begin execution of your program, as the datum at address 0 is 0.

2=IADR

Indirect addressing trigger - the VALUES at the operand addresses supplied with this instruction are taken as the operand ADDRESSES for the NEXT instruction, substituting and hence ignoring whatever data addresses were supplied in the next instruction.

For instance, if the datum at address 1 contains the value 0, the datum at address 4 contains the value 5, the datum at address 5 contains the value 0, and the datum at value 3 contains the value 63, then the following program does NOT loop infinitely (each operation given in the form command address, operator, first data address, second data address):

1, 2, 4, 3
2, 1, 1, 1

The second line SHOULD jump back "infinitely" (actually: until terminated by the exhaustion of the run limit) to the first instruction. But the first instruction modifies the second instruction to actually be executed as if it were:

2, 1, 5, 63

And as the value at 5 is 0, the jump triggers to instruction 63, and then the program "glides out" and terminates on the NOOPs in command memory.

3=OUTP

Output data in a range - if both range limits are the same, output only the datum at a single data address. A shorthand for outputting only one datum is giving the second data address as 0.

Example:

3, 4, 5

Output the data at the addresses 4 and 5.

4=INPT

Input data in a range - works the same as OUTP, but for data entry. Giving twice the same address, or the second address as 0, outputs a single datum.

Example:

4, 5, 7

Input the data at the addresses 5, 6, and 7.

5=SADR

Set address as value. The first data address receives as value the supplied second data address.

If the datum at address 6 contains 0, then

5, 6, 7

will let the datum at address 6 contain the value 7.

6=SVAL

Set value - namely, set the value of the first data address to the value of the second data address, i.e. the thing which other programming languages express as A=B or A:=B or thelike.

If data address 7 contains the value 0, and data address 8 contains the value 9, then

6, 7, 8

lets both 7 and 8 contain the value 9. The value is indeed copied, and not "pointed to"; i.e. if the value at data address 8 is later changed to 10, then data address 7 will still continue to hold a value of 9 and not be changed to hold 10.

7=IAAS

This and the next instruction are useful for a more natural expression of certain loops.

Indirect Address Assignment is like SADR with one more level of indirection and works like this:

Assume the first data address contains the value X.

X is then interpreted as a data address, and the value at the data address X is set to the second data address.

Example:

If data address 24 contains the value 3, then

7, 24, 25

will set the data address 3 to hold the value 25.

8=IVAS

The Indirect Value Assignment interprets the value held at the first operand's address as an address, and sets this address to hold the value held by the second operand's address. (This is like the above, just it sets the value, not the address of the second operand.)

Example:

If data address 24 contains the value 3, and data address 25 contains the value 6, then

8, 24, 25

will set the data address 3 to hold the value 6.

9=PLUS

Add two single numbers (not ranges of numbers), pointed to by the first and the second data operands. Store the result in the value of the first operand.

Example:

If data address 88 contains the value 11 and data address 89 contains the value 4, then

9, 88, 89

makes the data address 88 to contain the value 15.

10=MINS

Subtract a single number from another single number, each pointed to by the data operands. Store the result in the value of the first operand.

Example:

If data address 88 contains the value 11 and data address 89 contains the value 4, then

10, 88, 89

makes the data address 88 to contain the value 7.

11=MULS

Multiply a single number with another single number, each pointed to by the data operands. Store the result in the value of the first operand.

Example:

If data address 88 contains the value 11 and data address 89 contains the value 4, then

11, 88, 89

makes the data address 88 to contain the value 44.

12=DIVS

Divide a single number by another single number, each pointed to by the data operands. Store the result in the value of the first operand. If the divisor is 0, the result is set to 0.

Example:

If data address 88 contains the value 11 and data address 83 contains the value 0, then

12, 88, 83

makes the data address 88 to contain the value 0.

13=POXY

Raise X to the power Y. X is thereby taken always for its absolute value, and whatever sign it had before, is re-attached to the result. Thus, POXY of -3 to the 2nd is -9, not 9. The base is taken as the value at the first supplied data address, the exponent is taken as the value at the second supplied data address. The result is stored as the value of the first data address.

Example:

If data address 13 contains the value 2 and data address 12 contains the value 10, then

13, 13, 12

will let data address 13 contain the value of 1024. (Note how a command address, an operator number and a data address are entirely separate entities.)

14=LOXY

Logarithm to the base X, being the value under the SECOND data address, of Y, being the value under the FIRST data address. Both values are taken for their positive absolute value. If the base is zero, the result is given as zero. The computation makes use of the fact that log x (y) = (ln y) / (ln x). The result is stored in

the value of the first operand (thus overwriting the argument of the logarithm, but not the base of the logarithm).

Example:

If data address 1 contains 9 and data address 2 contains 3, then

14, 1, 2

will let data address contain 2.

15=IFRA

Give the integral and fractional part of a number stored as a value under the first supplied data address. After the operation, the first supplied data address will contain the integer part and the value under the second supplied data address will contain the fractional part. If you set the second data address to be 0, then in essence the fractional part gets discarded and the number held as value under the first data address is merely turned into an integer.

Example:

If data address 1 contains 2.22222, then

15, 1, 3

will result in data address 1 containing 2 and data address 3 containing 0.22222.

16=REMN

Determine the remainer of a division between the value at the the first data address divided by the value at the second data address (which, if it is zero, gives a result of zero). The result is stored at the first data address.

Example:

If data address 1 holds 9 and data address 3 holds 2, then

16, 1, 3

will let data address 1 hold 1 (as 2*4 = 8 and the remainder is 1), with the second value remaining unchanged.

17=AMNT

The "amount" is sought - deals with percentage calculations. The basis is the equation, amount = (1 + (percentage / 100)) ^ period. (That is, the percent are given as e.g. 25 and not as a share of 0.25)

The first data address holds as value the period, the second data address holds as value the percent.

Afterwards, the first data address holds the increase of the amount, based on the idea that the initial amount is "1".

Here, the first data address contains the value of the periods and the second data address contains the percent. The result (amount) is stored under the first data address.

Thus, if data address 1 contains 5.6 (periods) and data address 2 contains 11 (percent),

17, 1, 2

lets data address 1 contain ca. 1.8 (i.e. the growth of "1" over 11 periods at 5.6 percent).

18=PERD

The "period" is sought - deals with percentage calculations. The basis is the equation, amount = (1 + (percentage / 100)) ^ period. (That is, the percent are given as e.g. 25 and not as a share of 0.25)

The first data address points to the final amount (as grown from an original "1") and the second data address contains the percent. The result (period) is stored under the first data address.

Thus, if data address 3 contains 2.3 (the final amount) and data address 4 contains 3 (the percent),

18, 3, 4

returns in data address 1 a period of about 28.18.

19=PCNT

The "percentage" is sought - deals with percentage calculations. The basis is the equation, amount = (1 + (percentage / 100)) ^ period. (That is, the percent are given as e.g. 25 and not as a share of 0.25)

The first data address points to the final amount (as grown from an original "1") and the second data address contains the period. The result (percent) is stored under the first data address.

Thus, if data address 5 contains 1.728 (the final amount frown out of 1) and data address 6 contains 3 (the period),

19, 6, 5

returns in data address 1 a percentage of 20, i.e. 20% interest would cause such a development.

20=SWAP

Swap the values between the two data addresses. While this is exceedingly trivial, it is an annoying task to novices. After the command each data address holds the previous other data address' value as its own.

21=FACT

Deliver the factorial of the rounded absolute value of a number.

The number whose factorial is to be determined is pointed at by the second data address. The first data address will store the result of the operation. Factorials of numbers above 34 (and on smaller platforms, such smaller value) will be given as zero.

If data address 2 points at -14.12 as a value, then

21, 1, 2

results in data address 1 pointing at a value of +8.717829E+10 (or 87178291200).

22=COPY

Copy an address range.

This command is somewhat complex, because evidently, optimally FOUR values need to be supplied, namely the beginning of and end of each of two ranges (or three, if the second range is somehow anchored to the size of the first), but only TWO data addresses can be supplied.

The assumption here is that the operands of the command specify the beginning (first operand) and the end (second operand) of the DESTINATION range. The SOURCE range is determined by the value of the first operand (beginning) and by the value of the second operand (ending), each interpreted as an address. Upon copying of the range, these values will be written by actual range values. If the ranges are of unequal length, copy either revolvingly or partially, depending on whether the source falls short or is too long. If first the higher, then the lower address is supplied, it is assumed that a reversal of the range is desired.

Example:

Assume the data address range 1-10 contains the numbers 101-110 in ascending order.

Assume the datum at address 20 contains the value 10. Assume the datum at address 35 contains the value 1.

Then

22, 20, 35

fills the range between and inclusive datum address 20 and datum address 35 with the following values:

110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 110, 109, 108, 107, 106, 105.

23=FRIS

Fill range from initial value by a step (not much unlike how in a spreadsheet you select two neighbouring cells and fill a range with their difference).

The data addresses describe the range to be filled. The first data address contains as value the initial value. The second data

address contains as value the step to be used in filling the range.

If data address 31 contains the value 6.27 and data address 40 contains the value 12.13, then

23, 31, 40

lets the range between 31 and 40 contain the following numbers:

6.27, 18.4, 30.53, 42.66, 54.79, 66.92, 79.05, 91.17999 (a rounding error), 103.31, 115.44.

24=MNMX

Deliver the minimum and the maximum value within a range (based on real values, not absolute values). The first data address' value is seen as the beginning address of the range and the second data address' value is seen as the end address of the range. After the run, the first data address will contain the largest number and the second data address will contain the smallest number.

Thus, if the range between datum address 20 and datum address 35 with the following values:

110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 110, 109, 108, 107, 106, 105, and

data address 15 contains the value 35, and data address 16 contains the value 20, then

24, 15, 16

lets data address 15 contain the value 110, and data address 16 contain the value 101.

25=SORT

Sort a range and replace the original range by the sorted range. The first data address defines where the lowest value, the second data address defines the position where the highest value is to be stored.

Thus, if the data address range 1-10 contains the numbers 101-110 in ascending order, then

25, 10, 1

reverses the the range and the address range 1-10 will now contain the numbers 110-101 in descending order.

26=CORS

Coupled range sorting - this is a facility for sorting one range according to another. For instance, you have employee age and salary for employees of a certain type in two ranges of equal size, and you would like to sort the one range by age and see how the second range, that of the salaries, would change according to that age sorting. (Evidently, JUST sorting the age or JUST sorting the salaries is NOT what you want - you want to keep the COUPLING of each pair of elements in the respective ranges.)

The first data address contains as value the address of end of the first range. The first range is supposed to begin on the address just after the first data address (above or below).

The second data address contains as value the address of end of the second range. The second range is supposed to begin on the address just after the first data address (above or below).

It is required that both ranges be of the same length. The second range is sorted "according to" the first range.

If the second data address is zero, then CORS works like 25=SORT, and sorts the range of the first data address plus 1 until the value at the first data address taken as final range data address.

Assume the following data addresses and ranges contain the following values:

51-60: 42, 28, 61, 28, 27, 39, 31, 41, 60, 52

73-82: 45000, 75000, 23000, 112000, 80000, 33500, 43167, 21050, 66004, 28000

50: 60

72: 82

Then

26, 50, 72

will result in the following range contents:

51-60: 27, 28, 28, 31, 39, 41, 42, 52, 60, 61

73-82: 80000, 75000, 112000, 43167, 33500, 21050, 45000, 28000, 66004, 23000

Notice how in each case, the person aged 31 earns 43167.

27=TURN

Turn a range "upside down", that is, whichever values were contained at the beginning of the range find their place at the corresponding end of the range and vice versa. If the second supplied data address is zero or same as the first data address, do nothing.

Thus, if the data address range 1-10 contains the numbers 101-110 in ascending order, then

27, 1, 10

will make the range contain the numbers 110-101 in descending order, and another

27, 1, 10

will again reverse the range to its original condition of containing the numbers 101-110 in ascending order. (As opposed to the 25=SORT example above, the values are only "turned", but not sorted; any sorting is purely accidental and present in the inital range.)

28=SUMR

Sum up a range. The first data address contains a value, which value is the address of the beginning of the range. The second data address contains a value, which value is the end of the range. The result of the sum is stored as a value under the first data address.

If the data address range 1-10 contains the numbers 101-110 in ascending order, and datum 11 holds the value 1 and datum 12 holds the value 10, then

28, 11, 12

results in the data address 11 holding the value of 1055, the sum of 101 until 110 inclusive.

29=SUSQ

Deliver the sum of squares and the square rooted sum of squares of a range; very similar in operation to SUMR. The first data address contains a value, which value is the address of the beginning of the range. The second data address contains a value, which value is the end of the range. The result of the sum is stored as a value under the first data address.

If the data address range 1-10 contains the numbers 101-110 in ascending order, and datum 11 holds the value 1 and datum 12 holds the value 10, then

28, 11, 12

results in the data address 11 holding the value of 1055, the sum of 101 until 110 inclusive.

30=IXTH

Turn each X within a range to 1/Xth. If the value is zero, 1/Xth of it is assumed to be zero, too. The original range is replaced by these values.

31=ABSR

Deliver the absolute value of each value of a range, replacing the original range.

32=SQRT

Replace each element of a range by the square root of the absolute value of this element.

33=SQUA

Replace each element of a range by the value of raising this element to its second power. Here, -3 ^ 2 really gives 9 and not -9; see in contrast 13=POXY.

34=CBRT

Replace each element of a range by the cubic root of the value of this element.

35=CUBE

Replace each element of a range by the value of raising this element to its third power.

36=LNRN

Replace each element of a range by the natural logarithm of the absolute value of this element; ln(0) is given as 0.

37=EXPR

Replace each element of a range by raising the number e (ca. 2.718282) to the power of this element, i.e. replace each X by e ^ X.

38=RADE

Assume each value in a range is given in radians and substitute it with the equivalent value in degrees, that is, substitute each X by X * 180 / pi. Fractions are given not in minutes and seconds, but simply in decimal parts of the degree.

39=DERA

Assume each value in a range is given in degrees (and decimal parts of degrees; not minutes and seconds) and substitute it with the equivalent value in radian, that is, substitute each X by X * pi / 180.

40=SIND

Replace each element of a range, assuming it is given in degrees (of decimal fraction, i.e. not with minutes and seconds) by the value of its sinus.

41=COSD

Replace each element of a range, assuming it is given in degrees (of decimal fraction, i.e. not with minutes and seconds) by the value of its cosinus.

42=TAND

Replace each element of a range, assuming it is given in degrees (of decimal fraction, i.e. not with minutes and seconds) by the value of its tangens; tangens of 90 and 270 degrees is given to be zero.

43=ASND

Replace each element of a range by its arcussinus value in degrees (of decimal fraction, i.e. not with minutes and seconds).

44=ACSD

Replace each element of a range by its arcuscosinus value in degrees (of decimal fraction, i.e. not with minutes and seconds).

45=ATND

Replace each element of a range by its arcustangens value in degrees (of decimal fraction, i.e. not with minutes and seconds).

46=MSTD

Compute the mean and standard deviation of a range.

When called, the first data address contains as value the address of the beginning of the range and the second data address contains as value the address of the ending of the range. After the end of the computation, the first data address will contain the average or mean, and the second data address will contain the standard deviation (computed over n-1, i.e. on a sample).

Example:

If the data address range 1-10 contains the numbers 101-110 in ascending order, data address 18 contains 1 and data address 19 contains 10, then

46, 18, 19

results in data address 18 containing as value 105,5 (the average or mean), and data address 19 containing as value 3.02765 (the standard deviation on a sample).

47=ZERO

Replace each element of a range by zero. (This can be obtained in many way, but it is clearer to say it explicitly. This is like clearing a data range with -6 in the order acceptance mode, just this occurs during execution, e.g. to clear some workspace section.)

48=RAND

Fill a range with random numbers. The value stored at the first data address signifies the lower bound and the value stored at the second data address signifies the higher bound of the generated numbers. Both these values are being overwritten as the numbers are generated.

(The seed for the random number generation is taken as the datum at data address 1 in the C implementation, though maybe I should change that with some sum of instructions and data ranges.)

If data address 51 contains the value 12 and data address 60 contains the value 14, then the range 51-60 may contain after

48, 51, 60

a set of values not unlike these:

13.35516, 13.39943, 12.09040, 12.61702, 12.62410, 12.55335, 13.87267, 12.53588, 13.64661, 12.04021.

49=RUND

Replace each element x in a range with round(x) (in C); in some simplified implementations, this is just a truncation of any fractional part.

50=CEIL

Replace each element x in a range with ceil(x) (in C).

51=TANH

Compute the hyperbolic tangent of a range - that is, for each element x, replace it with tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x)).

52=DTNH

Compute the first derivative of the hyperbolic tangent for each element of a range - that is, for each element x, replace it with derivtanh(x) = 1 - tanh(x) * tanh(x). This function is useful should you wish to do anything with neural networks.

53=PLUR

Add a range to another range. This and the following three operations are range-range-operations.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the address end of the second range. The range begins one address before or one address after the supplied second operand address.

The result is stored in the first range. The ranges are applied to each other in ascending or descending order, depending whether the begin and end addresses of the ranges were given in ascending or descending order.

Example:

Let the data address range 21-30 contain the values

11, 12, 13, 14, 15, 16, 17, 18, 19, 20.

Let the data address 20 contain the value 30 (the end of the first range which begins at 21).

Let the data address range 40-49 contain the values

1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000.

Let the data address 50 contain the value 40 (the end of the second range, in descending order, which begins at 49).

Then performing

53, 20, 50

results in the following values within the first range of data addresses 21 to 30:

2011, 1912, 1813, 1714, 1615, 1516, 1417, 1318, 1219, 1120.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

54=MINR

Subtract a range from another range. Analogous to above.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the address end of the second range. The range begins one address before or one address after the supplied second operand address.

The result is stored in the first range. The ranges are applied to each other in ascending or descending order, depending whether the begin and end addresses of the ranges were given in ascending or descending order.

Example:

Let the data address range 21-30 contain the values

2011, 1912, 1813, 1714, 1615, 1516, 1417, 1318, 1219, 1120.

Let the data address 31 contain the value 21 (the end of the first range, in descending order, which begins at 30).

Let the data address range 40-49 contain the values

1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000.

Let the data address 50 contain the value 40 (the end of the second range, in descending order, which begins at 49).

Then performing

54, 31, 50

results in the following values within the first range of data addresses 21 to 30:

911, 712, 513, 314, 115, -84, -283, -482, -681, -880.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

55=MULR

Multiply a range by another range. Analogous to above.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the address end of the second range. The range begins one address before or one address after the supplied second operand address.

The result is stored in the first range. The ranges are applied to each other in ascending or descending order, depending whether the begin and end addresses of the ranges were given in ascending or descending order.

Example:

Let the data address range 21-30 contain the values

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Let the data address 20 contain the value 30 (the end of the first range which begins at 21).

Let the data address range 40-49 contain the values

1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000.

Let the data address 39 contain the value 49 (the end of the second range which begins at 40).

Then performing

55, 20, 39

results in the following values within the first range of data addresses 21 to 30:

1100, 2400, 3900, 5600, 7500, 9600, 11900, 14400, 17100, 20000

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

56=DIVR

Divide a range by another range, whereby division by zero gives zero. Analogous to above.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the address end of the second range. The range begins one address before or one address after the supplied second operand address.

The result is stored in the first range. The ranges are applied to each other in ascending or descending order, depending whether the begin and end addresses of the ranges were given in ascending or descending order.

Example:

Let the data address range 21-30 contain the values

200, 190, 180, 170, 160, 150, 140, 130, 120, 110

Let the data address 31 contain the value 21 (the end of the first range, in descending order, which begins at 30).

Let the data address range 40-49 contain the values

1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000.

Let the data address 39 contain the value 49 (the end of the second range which begins at 40).

Then performing

56, 31, 39

results in the following values within the first range of data addresses 21 to 30:

0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

57=PLUN

Add to a range a number. This and the following three operations are range-single-number-operations.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the the number.

The result is stored in the first range, processing element by element with the supplied number.

Example:

Let the data address range 21-30 contain the values

11, 12, 13, 14, 15, 16, 17, 18, 19, 20.

Let the data address 20 contain the value 30 (the end of the first range which begins at 21).

Let the data address 17 contain the value 16.

Then performing

57, 20, 17

results in the following values within the first range of data addresses 21 to 30:

27, 28, 29, 30, 31, 32, 33, 34, 35, 36.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

58=MINN

Subtract from a range a number. Analogous to above.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the the number.

The result is stored in the first range, processing element by element with the supplied number.

Example:

Let the data address range 21-30 contain the values

11, 12, 13, 14, 15, 16, 17, 18, 19, 20.

Let the data address 20 contain the value 30 (the end of the first range which begins at 21).

Let the data address 7 contain the value 11.

Then performing

58, 20, 16

results in the following values within the first range of data addresses 21 to 30:

-6, -5, -4, -3, -2, -1, 0, 1, 2, 3.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

59=MULN

Multiply a range by another range. Analogous to above.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the the number.

The result is stored in the first range, processing element by element with the supplied number.

Example:

Let the data address range 21-30 contain the values

-6, -5, -4, -3, -2, -1, 0, 1, 2, 3.

Let the data address 31 contain the value 21 (the end of the first range, in descending order, which begins at 30).

Let the data address 7 contain the value 11.

Then performing

59, 31, 7

results in the following values within the first range of data addresses 21 to 30:

-66, -55, -44, -33, -22, -11, 0, 11, 22, 33.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

60=DIVN

Divide a range by a number, whereby division by zero gives zero. Analogous to above.

The first operand contains as value the address end of the first range. The range begins one address before or one address after the supplied first operand address.

The second operand contains as value the the number.

The result is stored in the first range, processing element by element with the supplied number.

Example:

Let the data address range 21-30 contain the values

2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

Let the data address 31 contain the value 21 (the end of the first range, in descending order, which begins at 30).

Let the data address 3 contain the value -2.

Then performing

60, 31, 3

results in the following values within the first range of data addresses 21 to 30:

-1, -2, -3, -4, -5, -6, -7, -8, -9, -10.

As seen, the address immediately before and the address immediately after a range may turn out useful for such operations, and it appears adviseable to keep these two respective data addresses available.

61=PROB

This is a look-up function for Gaussian distributions. For a given standard deviation distance (in standard deviation units) from a mean of zero, determine the covered probability. This is a range function, and each element of the range is assumed to be such a standard deviation distance.

If the data address range 21 to 30 contained the numbers:

0.5, -1, 3.465, 2.1, -1.923, 1.25, 2.333, -0.421, 0, -3.5,

then

61, 21, 30

turns the contents of that range into:

0.6915, 0.1587, 0.9997, 0.9821, 0.2740002 (rounding error), 0.8944, 0.9901, 0.3372, 0.5, 0.0001999736 (rounding error)

This function does not operate with interpolation, but merely tries to match the known value closest to the value it has been called upon.

62=STDD

This is a look-up function for Gaussian distributions and the reverse to the above. For a given probability, determine the standard deviation distance from a mean of zero. This is a range function, and each element of the range is assumed to be such a covered probability distance.

If the data address range 21 to 30 contained the numbers:

0.6915, 0.1587, 0.9997, 0.9821, 0.2740002, 0.8944, 0.9901, 0.3372, 0.5, 0.0001999736,

then

62, 21, 30

turns the contents of that range into:

0.5, -1, 3.39, 2.1, -1.92, 1.25, 2.33, -0.42, 0, -3.47.

This function does not operate with interpolation, either, but merely tries to match the known value closest to the value it has been called upon.

63=USER

Just like the Apollo Guidance Computer had a sort of "escape hatch" to access further memory in Block II, 1V0 has this special function which by default does nothing and which is left for special uses needed by the users of the system on various platforms. It is thus "reserved". On microcontrollers, perhaps USER could be used to implement GPIO functions. Or generally, file system functionality, etc, etc. Given that 8192*8192 addresses can be submitted as arguments, this function allows for another 67 108 864 possible operations. And given 2=IADR, these can be freely set, too.

other=NOKO

No known operation - if no operation matches a given operator, this is what is printed as operation. This may occasionally happen to actually defined operators, if they are, however, not implemented on a certain platform. On a fully equipped system with 64 operations, from 0 to 63, this will not be shown.

VI. Design choice regarding fixed-point systems and jump addresses

The system has one incompatibility or asymmetry across platforms with regard to such that employ fixed point: the jump command.

The jump command receives as second operator a "data address" that is to be interpreted as an "instruction address". This address could also be set by IADR. Now, should an instruction address of, say, 24, be given as 2400 or as 24? - Because only addresses up to 63 are even expressible, I decided that the address should be given WITHOUT any fixed-point-adjustment. I.e. a jump to "24" has the same meaning on all platforms, even if on ATTiny85, the number 24 alone means 0.24, and on MIDP 1.0 enabled featue-phones it is 0.024. This is also more intuitive and becomes an issue only when the jump address needs to be COMPUTED (and is not instead just pre-set as a datum in e.g. a jump table).

The other general alternative would have been to make all jumps indirect: jump not to the second data address, but to the address expressed by the value contained by the second data address. - I found that too convoluted. Such jumps should be the exception, as in an instruction list, you no longer even necessarily where a jump is going to. They are still possible by using IADR and letting IADR thus determine the jump conditions out of values. But not making this the default is indeed a premeditated design decision.

VII. Available commands per platform

Described is which commands are available per platform (which may be further simplified within the platform), noting that 63=USER is reserved and unimplemented everywhere, and left to the user to devise. Further note that microcontrollers do not have the order acceptance command -10, as an "exit" is not sensible. These exceptions are to be noted when interpreting "ALL".

a. - C

ALL

b. - JAVA

ALL

Variant without statistical look-up tables: further lacking 61=PROB and 62=STDD.

c. - Arduino UNO

ALL except 16=REMN, 17=FACT, 24=MNMX, 25=SORT, 26=CORS, 27=TURN, 34=CBRT, 35=CUBE, 47=ZERO, 48=RAND, 50=CEIL, 51=TANH, 52=DTNH, 61=PROB, 62=STDD

Operating through EEPROM. Operating through the serial console.

d. - Arduino MEGA2560

ALL

Operating through EEPROM. Operating through the serial console.

e. - ARM (tested on Arduino DUE)

ALL

Operating through RAM. Operating through the serial console. This system, not using EEPROM, is extremely general and should work on a multitude of platforms, including RISC-V etc.

f. - ESP8266 serial console variant

ALL

Operating through EEPROM emulation. Operating through the serial console.

g. - ESP8266 WiFi variant

ALL

Operating through WiFi instead of the serial console. Setup your access server with the name, port and password you devise in the scetch and listen for the inbound connection of the board, e.g. with netcat (also available in some variants of busybox on Android).

h. - ATTiny85

Order acceptance mode: ALL except -3 (entry of entire instructions as integers), -7 and -8 (turn tracing on and off).

Order execution mode: ONLY 0=NOOP, 1=JUMP, 2=IADR, 5=SADR, 6=SVAL, 8=IVAS, 9=PLUS, 10=MINS, 11=MULS, 12=DIVS are available.

For space reasons, safety measures and checks are nearly all removed.

i. - IVO MIDP 2.0 CLDC 1.1 midlet

The user interface of order acceptance mode looks different and goes screen-by-screen. The inital screen shows the available commands thus:

"WELCOME TO 1V0: -1show_instructions -2show_data -4enter_data 0NOOP 1JUMP 2IADR 5SADR 6SVAL 7IAAS 8IVAS 9PLUS 10MINS 11MULS 12DIVS 13POXY 14IFRA 15REMN 17AMNT 19PCNT 20SWAP 21FACT 22COPY 23FRIS 24MNMX 25SORT 26CORS 27TURN 28SUMR 29SUSQ 30IXTH 31ABSR 32SQRT 33SQUA 34CBRT 35CUBE 37EXPR 38RADE 39DERA 40SIND 41COSD 42TAND 46MSTD 47ZERO 49RUND 50CEIL 53PLUR 54MINR 55MULR 56DIVR 57PLUN 58MINN 59MULN 60DIVN"

However, the full set of order acceptance mode commands is available.

j. - IVOnano MIDP 1.0 midlet

The user interface of order acceptance mode looks different and goes screen-by-screen. The inital screen shows the available commands thus:

"WELCOME TO 1V0: 0000PRECISION -1show_instructions -2show_data -4enter_data 0NOOP 1JUMP 2IADR 5SADR 6SVAL 7IAAS 8IVAS 9PLUS 10MINS 11MULS 12DIVS 13POXY 17AMNT 19PCNT 22COPY 23FRIS 26CORS 28SUMR 30IXTH 32SQRT 34CBRT 46MSTD 53PLUR 54MINR 55MULR 56DIVR 57PLUN 58MINN 59MULN 60DIVN"

However, the full set of order acceptance mode commands is available.

VIII. Notes on platforms

a. - C

Executable is produced e.g. on Linux with:

```
gcc -o 1V0 1V0_YYYYMMDD.c  -lm
```

On Windows, tcc can be used. No fancy libraries are used.

b. - Java

Executable is produced e.g. thus:

rename 1V0_YYYYMMDD.java or 1V0_YYYYMMDD_no_stat.java to IVO.java; then do

```
javac IVO.java
```

and execute with

```
java IVO
```

For distribution, furthermore create a file, IVO.mf, and place these two lines into it:

```
Manifest-Version: 1.0
Main-Class: IVO
```

Thereafter do:

```
jar cmf IVO.mf IVO.jar IVO.class
```

This will create the file IVO.jar, and you can execute it with:

java -jar IVO.jar

- The supplied file is called here actually IVO_Java.jar, to distinguish it from IVO.jar, which is the midlet.

c. - Microcontrollers

I am using the Arduino IDE and it is really the straightforward way you are used to. Select the board, paste the source, and compile and upload as you would usually do.

The Arduino DUE has been the test platform for the ARM code. The ARM code is really the most portable of these and should work on pretty much any more powerful microcontroller, perhaps with some adjustment of the data and instructions memory sizes.

The Arduino UNO code compiles extremely narrowly - 32254 out of 32256 bytes of flash are used. It is imaginable that future developments may make a compilation impossible - then you can, essentially, consider "what to throw out". I would suggest 4=INPT (occupying then only 31542 flash bytes), as you can set your values prior to run in order acceptance mode with -4 anyway. (3=OUTP would hurt more - as you might actually like to have it to see intermediate results, e.g. for debugging.)

As EEPROM library for the ATTiny85 I used:

https://github.com/PaulStoffregen/EEPROM

The ATTiny85 uses fixed point with two decimal places, and so when you enter e.g. 10, you enter it as 1000, when you enter 0.43, you enter it as 43, and so forth. The computational adjustments are done automatically, you just need to take care of correct entry and interpretation. Note, however, that addresses in 1=JUMP and 2=IADR are to be given as unadjusted values, that is, a jump to address 37 is really a jump to "37" and not "3700".

The ATTiny85 has its instruction range artificially limited to 63. The EEPROM would, however, allow up to four times as much. The reason for the limitation is the supply of an absolute jump address, which is done by giving a data address of the desired size, and data addresses there only go up to 63. If you set instruction range to be larger, then more far-away jumps can only be realised through IADR and thus indirect addressing.

The ATTiny85, due to size limitations, has some useability quirks: Negative numbers get their minus AFTER the number. It is simply easier to negate the entire result when a minus is read than to keep space for an extra variable to hold a -1 and then to multiply with it or thelike. Also, numbers are terminated with an asterisk. So to enter "-14.12", you actually enter "1412-*" (note the two fixed point positions, too).

The ATTiny85 always traces all its execution and a data print-out with 2-* shows the entire range of data without range requests.

d. - Midlets

The midlets can be compiled e.g. with Sun's (now Oracle's) Wireless Toolkit 2.2. Refer here to IVO.java and IVOnano.java.

For IVOnano, select to build a MIDP 1.0. IVOnano uses fixed point with four decimal places, and so when you enter e.g. 10, you enter it as 100000, when you enter 0.4321, you enter it as 4321, and so forth. The computational adjustments are done automatically, you just need to take care of correct entry and interpretation. Note, however, that addresses in 1=JUMP and 2=IADR are to be given as unadjusted values, that is, a jump to address 37 is really a jump to "37" and not "370000".

For IVO, select to build a MIDP 2.0 with CLDC 1.1. This operates with floats, just like the larger systems.

To load them into your WAP-browser, point your browser to the JAD-file, NOT the JAR-file. (WAP-browsers load midlets through the address supplied in the JAD-file, if you point them to the JAR-file directly, they might not load it.)

e. - Source code comments

Admittedly, I had not fully foreseen the ease of portability of the 1V0 system. - For understanding the code, you best refer to the C, Java, Arduino MEGA2560, ARM or ESP8266 sources. Particularly ATtiny85 and the midlets do not really have too well-commented code, as there, it was just too much of a constant battle against space and functionality constraints.

IX. Sample interactions and source code

Finally, we show an example interaction, once in the C variant, and the same again in the ATTiny85 variant, and thereafter it is proposed how source code of the system may be saved.

Sample runs

We shall enter the system, clear the instructions (not necessary in the C variant, but necessary in the EEPROM supporting variants when you do not want ot re-use an old program but write a new one), and we shall write a program that adds the numbers from 10 to 23, giving 231 at datum position 4. We shall make use of interactive output, and we shall show how memory can be examined in the end, too, to demonstrate that "interactive output" is not actually needed and that you can use the system purely in a sort of "batch mode". When entering the program, we shall show it is possible to jump also in an area BEFORE your actual program start, if it contains NOOPs (which is the case after having cleared the instruction memory).

Firstly, we enter the data, then the instructions. This way, you can see how the instruction confirmations actually point at each respective datum, and which may help you visually see what happens. Also the use of the comment facility is shown.

```
./1V0
SYSTEM READY
CMD ADDRESS: -5
CLEAR INSTRUCTION RANGE, INCLUSIVE LIMITS.
FIRST COMMAND ADDRESS FROM WHERE ON TO CREAR: 0
SECOND COMMAND ADDRESS AND TO WHERE TO CLEAR: 0
INSTRUCTIONS CLEARED.
CMD ADDRESS: -4
ENTER DATA AS FLOATING POINT NUMBERS WITHIN INCLUSIVE RANGE LIMITS.
DATA ADDRESS FROM WHERE ON TO ENTER: 1
DATA ADDRESS UNTIL WHERE TO ENTER  : 3
0001: DATUM: 10
0002: DATUM: 23
0003: DATUM: 1
CMD ADDRESS: 3
OPERATION  : 6
DATA ADDR 1: 4
DATA ADDR 2: 1
CMNT (OPT) : We will store the output at datum 4, set the inital value there. *
00003 SVAL 0004>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
CMD ADDRESS: 4
OPERATION  : 9
DATA ADDR 1: 1
DATA ADDR 2: 3
CMNT (OPT) : Increase the beginning by the step of 1. *
00004 PLUS 0001>+1.000000E+01>+0.000000E+00 0003>+1.000000E+00>+1.000000E+01
```

```
CMD ADDRESS: 5
OPERATION  : 9
DATA ADDR 1: 4
DATA ADDR 2: 1
CMNT (OPT) : result becomes result plus begin (which is increased stepwise) *
00005 PLUS 0004>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
CMD ADDRESS: 6
OPERATION  : 6
DATA ADDR 1: 5
DATA ADDR 2: 2
CMNT (OPT) : begin testing whether the loop (run) is over - fetch the end value*
00006 SVAL 0005>+0.000000E+00>+0.000000E+00 0002>+2.300000E+01>+0.000000E+00
CMD ADDRESS: 7
OPERATION  : 10
DATA ADDR 1: 5
DATA ADDR 2: 1
CMNT (OPT) : if the test value minus the stepped begin value is 0 - we are done*
00007 MINS 0005>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
CMD ADDRESS: 8
OPERATION  : 3
DATA ADDR 1: 5
DATA ADDR 2: 5
CMNT (OPT) : demonstrate interactively the test value on platforms which allow*
00008 OUTP 0005>+0.000000E+00>+0.000000E+00 0005>+0.000000E+00>+0.000000E+00
CMD ADDRESS: 9
OPERATION  : 1
DATA ADDR 1: 5
DATA ADDR 2: 11
CMNT (OPT) : if 5 is 0 and we are done - eject and "glide out" the computation*
00009 JUMP 0005>+0.000000E+00>+0.000000E+00 0011>+0.000000E+00>+0.000000E+00
CMD ADDRESS: 10
OPERATION  : 1
DATA ADDR 1: 0
DATA ADDR 2: 4
CMNT (OPT) : otherwise do an absolute jump to continue looping and adding *
00010 JUMP 0000>+0.000000E+00>+0.000000E+00 0004>+0.000000E+00>+0.000000E+00
CMD ADDRESS: 11
OPERATION  : 3
DATA ADDR 1: 4
DATA ADDR 2: 4
CMNT (OPT) : this is the printing of the final result once all has been added *
00011 OUTP 0004>+0.000000E+00>+0.000000E+00 0004>+0.000000E+00>+0.000000E+00
CMD ADDRESS: -1
LIST INSTRUCTION RANGE, INCLUSIVE LIMITS.
FIRST COMMAND ADDRESS FROM WHERE ON TO LIST: 1
SECOND COMMAND ADDRESS AND TO WHERE TO LIST: 15
INSTRUCTIONS:
00001 00=NOOP 0000>+0.000000E+00>+0.000000E+00 0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
00002 00=NOOP 0000>+0.000000E+00>+0.000000E+00 0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
00003 06=SVAL 0004>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
DECIMAL REPRESENTATION: 402685953
00004 09=PLUS 0001>+1.000000E+01>+0.000000E+00 0003>+1.000000E+00>+1.000000E+01
DECIMAL REPRESENTATION: 603987971
00005 09=PLUS 0004>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
DECIMAL REPRESENTATION: 604012545
00006 06=SVAL 0005>+0.000000E+00>+0.000000E+00 0002>+2.300000E+01>+0.000000E+00
DECIMAL REPRESENTATION: 402694146
00007 10=MINS 0005>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
DECIMAL REPRESENTATION: 671129601
```

```
00008 03=OUTP 0005>+0.000000E+00>+0.000000E+00 0005>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 201367557
00009 01=JUMP 0005>+0.000000E+00>+0.000000E+00 0011>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 67149835
00010 01=JUMP 0000>+0.000000E+00>+0.000000E+00 0004>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 67108868
00011 03=OUTP 0004>+0.000000E+00>+0.000000E+00 0004>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 201359364
00012 00=NOOP 0000>+0.000000E+00>+0.000000E+00 0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
00013 00=NOOP 0000>+0.000000E+00>+0.000000E+00 0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
00014 00=NOOP 0000>+0.000000E+00>+0.000000E+00 0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
00015 00=NOOP 0000>+0.000000E+00>+0.000000E+00 0000>+0.000000E+00>+0.000000E+00
DECIMAL REPRESENTATION: 0
CMD ADDRESS: 0
OPERATION  : 1
DATA ADDR 1: 0
DATA ADDR 2: 1
CMNT (OPT) : Execute - we jump to 1 and not 3, but it does not matter (NOOPs) *
00000 JUMP 0000>+0.000000E+00>+0.000000E+00 0001>+1.000000E+01>+0.000000E+00
RUNLIMIT = 5001
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+1.200000E+01
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+1.100000E+01
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+1.000000E+01
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+9.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+8.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+7.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+6.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+5.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+4.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+3.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+2.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+1.000000E+00
DATA OUTPUT OF THE RANGE 0005 TO 0005:
OUTPUT: ADDRESS=0005 DATUM=+0.000000E+00
DATA OUTPUT OF THE RANGE 0004 TO 0004:
OUTPUT: ADDRESS=0004 DATUM=+2.310000E+02
RUNLIMIT = 4801
RUNLIMIT = 4601
RUNLIMIT = 4401
RUNLIMIT = 4201
RUNLIMIT = 4001
SYSTEM READY
CMD ADDRESS: -2
LIST DATA RANGE, INCLUSIVE LIMITS.
FIRST DATA ADDRESS FROM WHERE ON TO LIST: 1
SECOND DATA ADDRESS AND TO WHERE TO LIST: 4
```

```
DATA:
0001: +2.300000E+01 +2.300000E+01 +1.000000E+00 +2.310000E+02 +0.000000E+00
CMD ADDRESS: -10
SYSTEM RUN TERMINATED
```

Note that the instructions start from 3, yet we jump to 1 to begin... but due to the NOOPs on freshly initialised memory, that is just fine. - The advantage here is, you can start to write your program "further up", and resort to adding code in the lower addresses when you figure out that you forgot some important initialisation setup.

The same computation is now to be shown on an ATTiny85. This is a straight copy of the interaction in the Arduino IDE.

Note that the numbers are printed by the system after entry "as they should look like", but actually were entered at the Arduino IDE serial monitor as described with asterisks as terminators of each number. (If you happen to forget the asterisk, and just enter a number, and nothing appears - just enter only an asterisk, and it will be accepted.) The commands are entered as in the "big" version, to demonstrate a good general compatibility. EEPROM, though, is cleared (with 5-*) at the end of the program entry before run in the respective ranges.

```
C:-4
:1
:3
1:1000
2:2300
3:100
C:3
O:6
:4
:1
C:4
O:9
:1
:3
C:5
O:9
:4
:1
C:6
O:6
:5
:2
C:7
O:10
:5
:1
C:8
```

```
O:3
:5
:5
C:9
O:1
:5
:11
C:10
O:1
:0
:4
C:11
O:3
:4
:4
C:-5
Z
:1
:2
C:-5
Z
:12
:63
C:0
O:1
:0
:1
0-10-1
1000
2000
3-6-4-1
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
```

10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3

5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
10-10-4
4-9-1-3
5-9-4-1
6-6-5-2
7-10-5-1
8-3-5-5
9-1-5-11
11-3-4-4
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000
36000
37000
38000
39000
40000
41000
42000

43000
X
C:-2
2300-1
2300-2
100-3
23100-4
0-5
0-6
0-7
0-8
0-9
0-10
0-11
0-12
0-13
0-14
0-15
0-16
0-17
0-18
0-19
0-20
0-21
0-22
0-23
0-24
0-25
0-26
0-27
0-28
0-29
0-30
0-31
0-32
0-33
0-34
0-35
0-36
0-37
0-38
0-39
0-40
0-41
0-42
0-43
0-44
0-45
0-46
0-47
0-48

```
0-49
0-50
0-51
0-52
0-53
0-54
0-55
0-56
0-57
0-58
0-59
0-60
0-61
0-62
0-63
```

Note here the line stating "23100-4", i.e. the actual sum of 231 is at data address 4, as it should be.

Also note the line marked "X" and signifying the exhaustion of the runlimit (which was set at 126) - subsequently, it has been increased tenfold, and under the new setting, the computation would just "glide out" through the remaining NOOPs in instruction memory.

Source code example

From this, it could be proposed to keep source code in a form not unlike the one that is anyway used for entering it into the system, with the following proposal being made for entering e.g. a program that check whether a given number is a prime number. (This prosal is solely designed for demonstration and not for efficiency and should work on all 1V0 variants, including ATTiny85.)

TITLE: CHECK PRIME

Task:

Determine is the number held in datum[0] prime or not. Give as result a 1 (yes) or 0 (no) in datum[10], and if not, give in datum[11] and datum[12] the first factors discovered.

General description:

A number is set in datum[1], and after program run, datum[10] contains either 1, if the number is prime, or 0, if it is not. Data input and output are handled by -4 and -2 in order execution mode.

Whether a number is prime or not is by trying repeated subtraction of each number from 2 until at least half the sought number. If any such repeated subtraction delivers a rest of zero, then the analysed number is not prime.

Specifically, the integers from 2 until and, if odd, going beyond, n/2 are tested as divisors, by way of repeated subtraction from the dividend (which is the number to be analysed). (More proper would be until the square root of n - until that time, either the factors have been found or the number must be prime.) If by any of this integers repeated subtraction without a rest is possible, then the number is not prime, and the step-size of the repeated subtraction times the subtraction cycles gives the analysed number. Each time a repeated subtraction does not render "a division without rest", another repeated subtraction, this time with a larger subtrahend (increased by 1) is attempted. If n/2 are reached and still no repeated subtraction without a rest is possible, then the number analysed must be prime and datum[10] is set accordingly.

Computation ends when either it is determined that n/2 is reached without any factors - and then the number is prime - or until, before this, factors for a division without a rest are found - and then the analysed number is not prime. For this, check at the end of the run data address 10 - if it is 1, the number is prime, if it is 0, it is not.

Data addresses, where [X] designates the value stored at X:

User-set:
1: the number to be tested (INPUT)
2: 2 (needed to run through
3: 1 (the step size)

System-set:
4: half the number to be tested
5: gradually increased subtrahend (computatons ends when it reaches [4]/2)
6: copy of [4], repeated subtraction target
7: counter for subtractions; [5]*[7] are checked whether they match [1]
8: temporary holder of [1] for equality test to the initial number
9: temporary holder of [4] for termination test (saying [1] is indeed prime)
10: result if the number is prime - 0 if not, 1 if yes (OUTPUT)
11: if the number is not prime, one factor of it (OUTPUT)
12: if the number is not prime, another factor of it (OUTPUT)

Commands:	Comments:
4,10,10,10	Set the result, datum[10], to zero. PROGRAM START ADDRESS. *
5,6,5,3	Start the repeated subtraction for divisibility test with 1.*
6,6,4,1	(Determine the end of trying numbers - until n/2, for if *
7,12,4,2	) until then no division has ruined the prime, none will. *
8,6,6,1	Set up repeated subtraction, on a copy of the number tested.*
9,9,5,3	Increase the subtrahend (repeated jumpt target). *
10,6,7,3	Set the count of subtraction attempts to 1. *
11,6,9,4	(If the increased subtrahed is greater or equal to the max *
12,10,9,5	limit of n/2 (actually, square root of n), then the number *
13,1,9,26	) must be flagged as prime and computation is terminated. *
14,10,6,5	(If repeated subtraction delivers <=0, end this cycle and *
15,1,6,18	) check whether the number was divided exactly (no prime). *
16,9,7,3	(Otherwise increase the subtraction cycle counter by 1 and *
17,1,0,14	) try another subtraction and counting further cycles of it.*
18,6,11,7	(Record the first factors (in case it is not prime) which *
19,6,12,5	) divided the number without leaving a rest. *
20,11,7,5	Multiply subtraction step times subtraction cycles. *
21,6,8,1	To compare with the above, copy the number being tested. *
22,10,8,7	Difference between number tested and steps-times-cycles. *
23,11,8,8	Square it (i.e. make it positive and diminish small errors).*

```
24,1,8,27         If it is 0, the division was exact; no prime number, end.   *
25,1,0,8          Jump to testing another division, increasing the subtrahend.*
26,9,10,3         Jump target in case the number is prime: set datum[10] to 1.*
27,0,0,0          Jump target in case the number is NOT prime: no prime flag. *
28,1,0,63         End computation by jumping to the end of instruction memory.*
```

Sample run:

(Optional)

X. Dedication

This system has been created in loving memory of my father, and with deepest gratitude, whom I miss every single day and who will forever continue to inspire me.