The CENTIPEDE Processor

1 INTRODUCTION

This document presents the architecture and programming syntax of a minimalistic stack-based computational and data processing system, oriented toward embedded applications in which a shorter idea-to-product route and easier aftermarket maintainability are pursued.

The architecture consists of one or more identical execution cores, each completely independent from the others. All cores share access to a common linear main memory up to 4 Gbytes both for data and code.

[CORES] defines the number of execution cores in the machine.

[RST_DEPTH] defines the depth of the internal hardware stack for return addresses. Up to 256-level deep stack is supported. The return stack is built from 32-bit cells.

[DST_DEPTH] defines the depth of the internal data stack for operations. Up to 256-level deep stack is supported. Each cell in the data stack is in fact a small data structure, consisting of a 64-bit data register and a 32-bit type/dimension register.

[VARIABLES] defines the maximum number of data variables in simultaneous use by an individual core. Supported are blocks with up to 256 variables. The variables are internal data structures, each consisting of a 64-bit value register and a 32-bit type/dimension register.

The variables array is in a sense just a block of internal registers, which a core can use as temporary data storage. Each variable has its own id (index in the variable array) and can be allocated or released by the core at any time. In addition to that, variables can have a dimension and act as linear arrays with size up to 512M elements.

A variable with dimension 0 is just a normal data register, and the value is stored directly in its value register. With dimension greater than 0, the variable is considered as an array whose start address is stored in the variable’s value register.

Accessing an unallocated variable, or allocating a variable which is currently already allocated, are considered as exceptions which can generate a system exception.

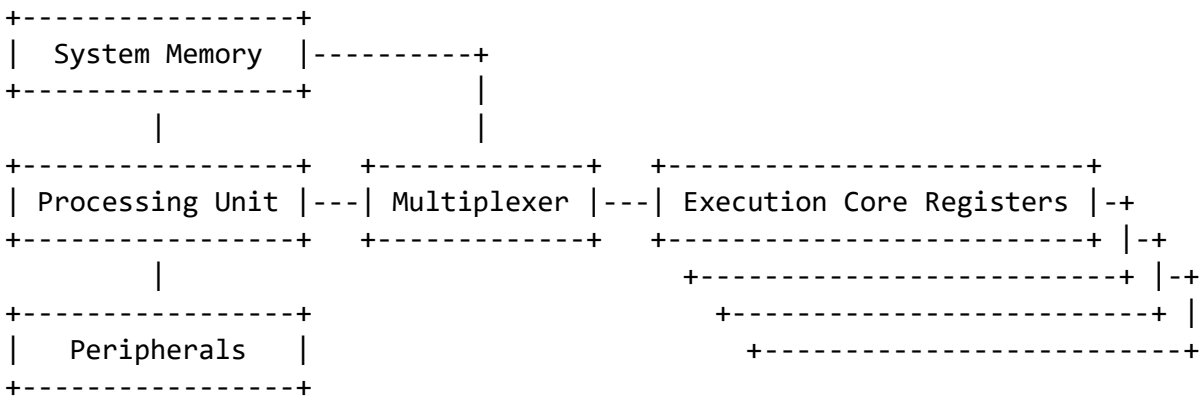
All internal registers in the cores, including the data stack, return stack, and the variables, are inaccessible from the outside world. They are only accessible by the particular core to which they belong during the execution of its instructions.

1.1 Register Model

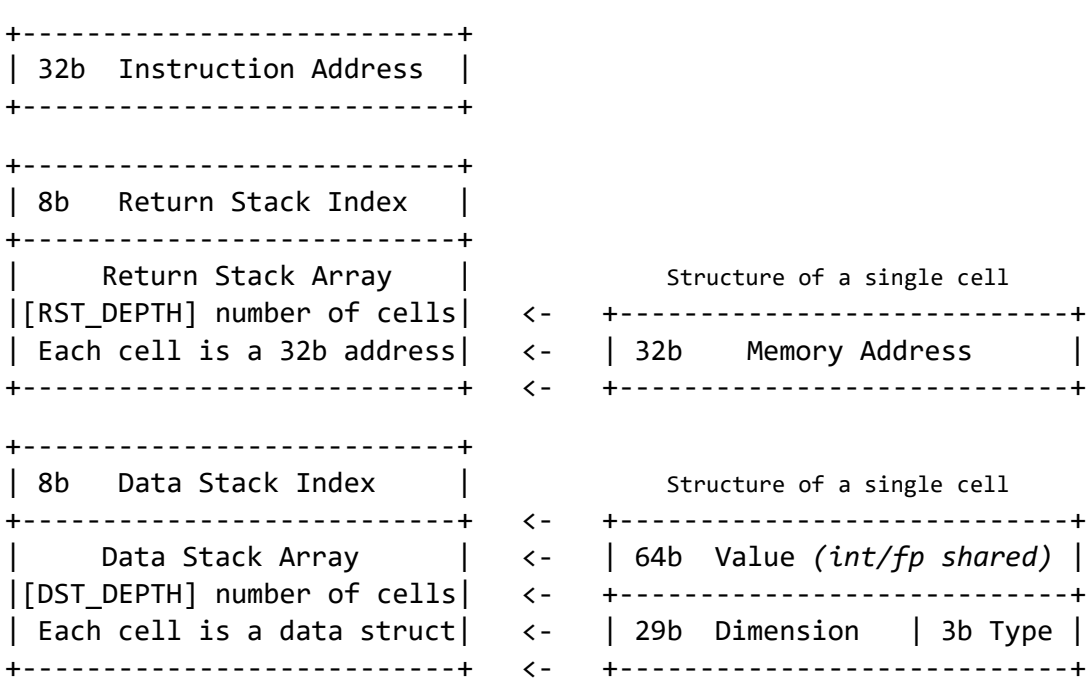
The system is built from one or more execution cores, all sharing common memory and peripheral resources.

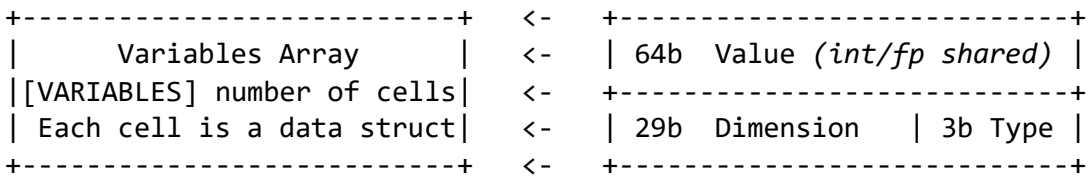
The “cores” are in fact nothing more than an organised area of fast RAM, allocated for each one of them. The single processing unit in the system gets sequentially connected to each of the active set of execution core registers through a multiplexer, which swaps the core on after execution of every instruction. During the execution of a current instruction, the multiplexer will fetch the next for next execution core and will supply it to the processing unit on the next switching cycle.

An overall diagram of the Centipede architecture is shown below (with four core register sets shown in the example):



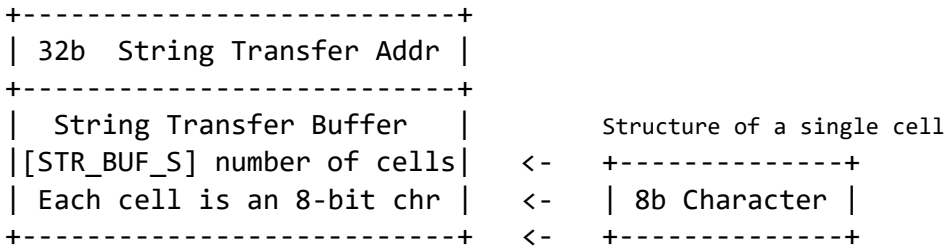
The register model of a single core is shown below:





A 64-bit value register can hold either a floating-point number or an integer number. Integer numbers are always considered as signed.

OPTIONAL (single unit, only needed in systems with support for a console)



[STR_BUF_S] defines the size of the internal string buffer in bytes for each individual core. The size depends on the maximum number of characters which are needed for expressing a number held in any of the supported data types. For 64-bit types, the size of the buffer defaults to 64 bytes.

1.2 Supported Data Types

Up to seven data types are supported in the architecture:

000: Void data type; used internally to mark free variables
001: 1-bit data type (value 0 or 1)
010: 8-bit byte
011: 16-bit integer half-word
100: 32-bit integer word
101: 64-bit integer double-word
110: 64-bit floating point number (IEEE754 double-precision FP)
110: Reserved for future extended precision floating point type

1.3 Endianness and Data Alignment

The Centipede processor uses Little-Endian byte ordering. This means that in data values which occupy more than a single byte, the lower-order bytes are stored at lower addresses. Thus, as an example in a 16-bit value, the first byte will contain the bits 0…7, and the second byte will follow with the bits 8…15.

Data alignment is required for access to memory via data types longer than a single byte. The memory address needs to be aligned in a such way, so it is divisible by the data length without a remainder. Hence, 16-bit data can be accessed only from even memory addresses, 32-bit data can be accessed only from addresses divisible by 4 with no remainder, and 64-bit data can be accessed only from addresses divisible by 8 with no remainder.

Given the rules above, all data instructions and most of the constant instructions, will require aligning.

Appropriate padding with ***nop*** instructions needs to be performed by the assembler program for data constants located within the executable code.

2 DATA STACK

Program format:

clear

Binary format:

(0x08)

Clear the data stack and reset the stack pointer.

Program format:

depth

Binary format:

(0x09)

Return into the data stack the current data stack depth before executing the instruction.

Program format:

dup

Binary format:

(0x0A)

Duplicate the top value in the data stack.

Program format:

drop

Binary format:

(0x0B)

Remove the value which is on the top of the stack.

Program format:

depth **over**

Binary format:

(0x0C)

Copy a value which is at specified depth on to the top of the data stack. Depth level 0 is the top of the stack itself, so the instruction “0 over” is a functional equivalent of the instruction “dup”. In case the depth

parameter is invalid (smaller than 0, or greater than the number of elements currently in the stack less one), a system exception will be triggered.

Program format:

depth **swap**

Binary format:

(0x0D)

Swap a value which is at specified depth with the top of the data stack. Depth level 0 is the top of the stack itself, so the instruction “0 swap” will essentially do nothing. In case the depth parameter is invalid (smaller than 0, or greater than the number of elements currently in the stack less one), a system exception will be triggered.

3 CONSTANTS

Program format:

const0
const1
cint8 value
cint16 value
cint32 value
cint64 value

Binary format:

(0x10) for const0
(0x11) for const1

(0x12) (byte)

(0x13) (byte-L) (byte-H)

(0x14) (byte-LL) (byte-LH) (byte-HL) (byte-HH)

(0x15) (byte-LLL) (byte-LLH) (byte-LHL) (byte-LHH)
(byte-HLL) (byte-HLH) (byte-HHL) (byte-HHH)

A single signed integer constant is read from the program code and placed into the data stack. Supported data sizes are 8-bit, 16-bit, 32-bit, and 64-bit.

Program format:

cfloat value

Binary format:

(0x16) (byte-LLL) (byte-LLH) (byte-LHL) (byte-LHH)
(byte-HLL) (byte-HLH) (byte-HHL) (byte-HHH)

A single 64-bit floating point constant is read from the program code and placed into the data stack.

Program format:

datai8 value, value, value …
datai16 value, value, value …
datai32 value, value, value …
datai64 value, value, value …

Binary format:

(0x1A) (count-LL) (count-LH) (count-HL) (count-HH) ((byte) ...)

(0x1B) (count-LL) (count-LH) (count-HL) (count-HH)
(((byte-L) (byte-H)) ...)

(0x1C) **(count-LL) (count-LH) (count-HL) (count-HH)**
(((byte-LL) (byte-LH) (byte-HL) (byte-HH)) ...)

(0x1D) **(count-LL) (count-LH) (count-HL) (count-HH)**
(((byte-LLL) (byte-LLH) (byte-LHL) (byte-LHH)
(byte-HLL) (byte-HLH) (byte-HHL) (byte-HHH)) ...)

Note: There is no native **datai1** instruction.

A pointer to an array of signed integer constants is placed into the data stack. All values in the array are assumed to be of the same data type. The number of values is determined by an always-present 32-bit number immediately following the instruction code. The entire constant data array is then skipped, and the execution continues with the instruction immediately following the last data value in the array.

Program format:

dataf value, value, value …

Binary format:

(0x1E) **(count-LL) (count-LH) (count-HL) (count-HH)**
((byte-LLL) (byte-LLH) (byte-LHL) (byte-LHH)
(byte-HLL) (byte-HLH) (byte-HHL) (byte-HHH)) ...)

A pointer to an array of 64-bit floating point constants is placed into the data stack. The number of values is determined by an always-present 32-bit number immediately following the instruction code. The entire constant data array is then skipped, and the execution continues with the instruction immediately following the last data value in the array.

4 VARIABLES

Program format:

dimension id **vint1**
dimension id **vint8**
dimension id **vint16**
dimension id **vint32**
dimension id **vint64**
dimension id **vfloat**

Binary format:

(0x21) for vint1
(0x22) for vint8
(0x23) for vint16
(0x24) for vint32
(0x25) for vint64
(0x26) for vfloat

Create a new variable with parameters from the data stack. Every variable has a unique identification number "id" which ranges from 0 to 255 (or another limit, based on the concrete system implementation). This means up to 256 variables can be used simultaneously at any moment in a program.

An attempt to create a variable with id which is currently being used by another variable has two possible behaviours, based on the specific system configuration:

a) (default) A system exception will be triggered.
b) The new variable will silently overwrite the old one without any indication of a collision.

Creating a variable also requires specifying its dimension. Single variables have dimension 0 which means there is no additional memory associated with it. Dimensions from 1 through the maximum allowed (2^{29}-1), indicate that the variable is an array of elements in the memory, all of the same data type, and the variable itself is a pointer to the first element in that array.

When an array variable is created, in practice it doesn't allocate any physical memory. It only defines a region in the system memory which can be accessed through indexing. The base of that region is stored in the core variable itself (otherwise said, at index 0). Thus, an array with one million elements can be legally defined in a system with actual 64KB of memory. It is also required storing the base array address in the core variable at index 0, before any further work with the array.

Accessing elements outside of the defined array has three possible outcomes:

a) (default) A system exception will be triggered.
b) The error will be ignored, and the indexed memory will be accesses at its calculated address.
c) The index will wrap around within the defined array range.

Attempting to create a variable with an invalid id (outside the allowed range of numbers) or an invalid dimension parameter, will always result in a system exception being triggered for the case.

Supported data types are 8-bit, 16-bit, 32-bit, 64-bit signed integer values, and 64-bit floating point numbers.

Program format:

id **free**

Binary format:

(0x20)

Free up a variable id. After execution of the instruction the variable id can be used for the creation of a new variable. If the id is already free, the instruction has no effect. An invalid id parameter will trigger a system exception.

Program format:

value [index] id **=**
[index] id **?**

Binary format:

(0x36) for =
(0x37) for ?

'=' gets value from the data stack and stores it into a variable. '?' gets value from a variable and puts it into the data stack.

Single variables only require id supplied in the data stack for the instruction. Array variables also take a mandatory preceding index parameter.

For the array variables only, index range from 0 to the maximum value specified during the creation of the variable. Index 0 is the core variable value itself. The core value contains a memory address pointing to the first element in the array. Index values from 1 onwards specify the actual index in the array.

Attempting to execute the instruction with an invalid id, invalid index (outside of the specified range for the particular variable) or accessing a variable which is supposed to be free, will result in a system exception.

Program format:

[index] id **v--**

Binary format:

(0x4A)

Decreases the value stored in variable by 1.

Single variables only require id supplied in the data stack for the instruction. Array variables also take a mandatory preceding index.

Program format:

[index] id **v++**

Binary format:

(0x4B)

Increases the value stored in variable by 1.

Single variables only require id supplied in the data stack for the instruction. Array variables also take a mandatory preceding index.

5 CONTROL

Program format:

reset

Binary format:

(0x00)

Reset the processor and restart the execution of the program.

This instruction affects only the current execution core (unless it is core 0); any others in the system will continue their work unaffected.

When executed by core 0, the instruction first stops all other cores and then restarts the core.

Program format:

where

Binary format:

(0x18)

Put the current value of the instruction pointer into the data stack.

Program format:

END

Binary format:

(0xFF)

Terminate execution of all cores and enter the built-in Monitor. The purpose of this instruction is mainly to ensure that a program is properly terminated. The assembler places it automatically at the very end of a program code. It is also used by the disassembler to determine when to stop the listing process. *END* instructions should not be used within a program unless the developer intends to invoke the Monitor.

Program format:

Dbase

Binary format:

(0x80)

Put in the stack an unsigned 32-bit value which represents the physical start address of the data memory (RAM) on the specific hardware platform.

Program format:

Dsize

Binary format:

(0x82)

Put in the stack the size of the data memory (RAM) in bytes on the specific hardware platform. The Centipede architectures assumes that there is a single contiguous chunk of RAM in the system.

Program format:

Pbase

Binary format:

(0x81)

Put in the stack an unsigned 32-bit value which represents the physical start address of the program memory (ROM) on the specific hardware platform.

Program format:

Psize

Binary format:

(0x83)

Put in the stack the size of the program memory (ROM) in bytes on the specific hardware platform. The Centipede architectures assumes that there is a single contiguous chunk of ROM in the system.

Program format:

nop

Binary format:

(0xF0) normal nop instruction (creates a new line in code listing)
(0xF1) invisible padding nop instruction (not shown in code listing)
(0xF2) nop code used to tell the lister that the following constant is binary
(0xF3) nop code used to tell the lister that the following constant is octal
(0xF4) nop code used to tell the lister that the following constant is hex
(0xF5) nop code used to tell the lister that the following constant is 'char'
(0xF6) nop code used to tell the lister that the following constant is "string"

No operation. The execution continues with the instruction that follows.

There are several *"nop"* codes, all performing the same function during execution – no operation. However, they are treated in different ways when the code is being disassembled.

The normal *"nop"* instruction is not displayed in a program listing. Instead, it causes the disassembler to start a new line. This behaviour is utilised by the assembler service to store the original new line codes from the source, so they can be restored later in a disassembled source output.

When any of the "lister nop" instructions are reached during disassembly, the lister sets an internal flag specifying the format in which the next constant needs to be displayed. By default, it is decimal unless set otherwise by one of those nop codes.

The change of format applies only for one constant, after which the display mode for numeric constants reverts to decimal. The mode, however, applies for an entire fragment when it is followed by a *datai8*, *datai16*, *datai32*, or *datai64* instruction. In such case the mode reverts to decimal after the entire data set is listed.

Program format:

*text commentaries stored in the source*

Binary format:

(0xF7) (count-L) (count-H) ((byte) (byte) ...)

The instruction skips number of bytes set in the two bytes following immediately the instruction code.

Up to 65535-byte long commentaries are supported for a single instruction. If the text is longer, it will have to be split into the necessary number of parts to fit the criteria.

The main purpose of the commentary instruction is to enable text commentaries stored directly within the source, so they can be displayed when source listing is performed. It can, however, be repurposed as per a program specific need.

Program format:

@*Label*

Binary format:

(0xFE) (count) ((byte) (byte) ...)

During execution, the instruction skips number of bytes set in the byte which follows immediately the instruction code.

During assembly, every time when the assembler finds the label being used somewhere, it inserts in the code a CINT32 constant with the memory address of the first byte immediately following the label.

Program format:

!*name* ...

Binary format:

(0xFD) ((nonce-LLL) (nonce-LLH) (nonce-LHL) (nonce-LHH)
(nonce-HLL) (nonce-HLH) (nonce-HHL) (nonce-HHH))
(count-L) (count-H) ((byte) (byte) ...) ...

The eight bytes following the count must be always present, but they are always ignored during execution. Also, they are not included in the following "*count*" field.

During execution, the instruction skips number of bytes set in the two bytes "*count*" which the instruction code and the 8-byte "*nonce*" field.

During assembly, the code which follows the name definition and until the end of the text line, gets inserted every time when the assembler finds the name being used somewhere. This code could be typically a numeric constant, but could be also a data statement, or a sequence of instructions.

Up to 65535 effective characters can be included into a definition of name.

Program format:

reladdr **goto**

Binary format:

(0x28)

Jump unconditionally to an address relative to the current.

The address parameter is a signed integer up to 32-bit length. Thus, the jump can be performed with the range of -2,147,483,648 to 2,147,483,647 bytes from the start of the instruction.

Program format:

reladdr **call**

Binary format:

(0x29)

Call a subroutine from an address relative to the current.

The address parameter is a signed integer up to 32-bit length. Thus, the jump can be performed with the range of -2,147,483,648 to 2,147,483,647 bytes from the start of the instruction.

If the return stack if full, a system exception will be triggered.

A “return” instruction in the called subroutine will return the execution back to the instruction immediately following this one.

Program format:

absaddr **agoto**

Binary format:

(0x2A)

Jump unconditionally to an absolute address in the memory.

The address parameter is an unsigned integer, up to 32-bit length. Thus, the entire memory can be up to 4,294,967,296 bytes.

Program format:

absaddr **acall**

Binary format:

(0x2B)

Call a subroutine at an absolute address in the memory.

The address parameter is an unsigned integer, up to 32-bit length. Thus, the entire memory can be up to 4,294,967,296 bytes.

If the return stack if full, a system exception will be triggered.

A “return” instruction in the called subroutine will return the execution back to the instruction immediately following this one.

Program format:

value reladdr **gotoif**

Binary format:

(0x2C)

Jump conditionally (if the value of the parameter “value” is not 0) to an address relative to the current.

The address parameter is a signed integer up to 32-bit length. Thus, the jump can be performed with the range of -2,147,483,648 to 2,147,483,647 bytes from the start of the instruction.

If the parameter “value” is 0, the execution continues with the next instruction.

Program format:

value reladdr **callif**

Binary format:

(0x2D)

Call conditionally (if the value of the parameter “value” is not 0) a subroutine from an address relative to the current.

The address parameter is a signed integer up to 32-bit length. Thus, the jump can be performed with the range of -2,147,483,648 to 2,147,483,647 bytes from the start of the instruction.

If the parameter “value” is 0, the execution continues with the next instruction.

If the return stack if full, a system exception will be triggered.

A “return” instruction in the called subroutine will return the execution back to the instruction immediately following this one.

Program format:

value absaddr **agotoif**

Binary format:

(0x2E)

Jump conditionally (if the value of the parameter “value” is not 0) to an absolute address in the memory.

The address parameter is an unsigned integer, up to 32-bit length. Thus, the entire memory can be up to 4,294,967,296 bytes.

If the parameter “value” is 0, the execution continues with the next instruction.

Program format:

value absaddr **acallif**

Binary format:

(0x2F)

Call conditionally (if the value of the parameter “value” is not 0) a subroutine at an absolute address in the memory.

The address parameter is an unsigned integer, up to 32-bit length. Thus, the entire memory can be up to 4,294,967,296 bytes.

If the parameter “value” is 0, the execution continues with the next instruction.

If the return stack if full, a system exception will be triggered.

A “return” instruction in the called subroutine will return the execution back to the instruction immediately following this one.

Program format:

return

Binary format:

(0x04)

Return from a subroutine call. An empty return stack will trigger a system exception.

6 MULTI-CORE CONTROL

Program format:

cores

Binary format:

(0x01)

Put in the data stack the total number of execution cores in the system.

Program format:

active

Binary format:

(0x02)

Put in the data stack the number of currently running execution cores.

Program format:

current

Binary format:

(0x03)

Put in the data stack the id of the current execution core. The instruction returns the core number as it is at the moment. Core numbers get reordered when a ***stop*** command is executed.

Program format:

stop

Binary format:

(0x06)

Stop the execution of the current core. If the current is the only running core in the system, then the entire execution is halted as if an ***end*** instruction is executed.

Program format:

addr **run**

Binary format:

(0x05)

Run a new core and jump to memory address *addr* for its execution.

The new core is cloned from the current with its entire data stack and return stack. In case there is no more available cores, an exception is triggered.

The address parameter is a signed integer up to 32-bit length. Thus, the jump can be performed with the range of -2,147,483,648 to 2,147,483,647 bytes from the start of the instruction.

The number of the new core is determined automatically and may change during execution.

Program format:

absaddr **arun**

Binary format:

(0x07)

Run a new core and jump to memory address *absaddr* for its execution.

The new core is cloned from the current with its entire data stack and return stack. In case there is no more available cores, an exception is triggered.

The address parameter is an unsigned integer, up to 32-bit length. Thus, the entire memory can be up to 4,294,967,296 bytes.

The number of the new core is determined automatically and may change during execution.

7 COMPARISONS

Program format:

value1 value2 **==**

Binary format:

(0x30)

Compare “value1” with “value2” and return 1 in the stack, if they are equal, or 0 otherwise.

Program format:

value1 value2 **<>**

Binary format:

(0x31)

Compare “value1” with “value2” and return 1 in the stack, if they are not equal, or 0 otherwise.

Program format:

value1 value2 **<**

Binary format:

(0x32)

Compare “value1” with “value2” and return 1 in the stack, if “value1” is smaller than “value2”, or 0 otherwise.

Program format:

value1 value2 **<=**

Binary format:

(0x33)

Compare “value1” with “value2” and return 1 in the stack, if “value1” is smaller or equal than “value2”, or 0 otherwise.

Program format:

value1 value2 **>**

Binary format:

(0x34)

Compare “value1” with “value2” and return 1 in the stack, if “value1” is greater than “value2”, or 0 otherwise.

Program format:

value1 value2 **>=**

Binary format:

(0x35)

Compare “value1” with “value2” and return 1 in the stack, if “value1” is greater or equal than “value2”, or 0 otherwise.

8 BIT LOGIC

Program format:

value bits **shiftl**

Binary format:

(0x38)

Shift the value by given number of bits left. Number of bits given as 0 or negative will return the initial value unchanged and shifting by 64 bits or more will always produce result 0.

The instruction disregards the data type in the value, so when applied on a floating-point value the result will be inaccurate.

Program format:

value bits **shiftr**

Binary format:

(0x39)

Shift the value by given number of bits right. Number of bits given as 0 or negative will return the initial value unchanged and shifting by 64 bits or more will always produce result 0.

The instruction disregards the data type in the value, so when applied on a floating-point value the result will be inaccurate.

Program format:

value1 value2 **or**

Binary format:

(0x3A)

Bitwise OR of two operands.

The instruction disregards the data type in the value, so when applied on a floating-point value the result will be inaccurate.

Program format:

value1 value2 **exor**

Binary format:

(0x3B)

Bitwise Exclusive OR of two operands.

The instruction disregards the data type in the value, so when applied on a floating-point value the result will be inaccurate.

Program format:

value1 value2 **and**

Binary format:

(0x3C)

Bitwise AND of two operands.

The instruction disregards the data type in the value, so when applied on a floating-point value the result will be inaccurate.

Program format:

value **invert**

Binary format:

(0x3D)

Bitwise inversion of the operand.

The instruction disregards the data type in the value, so when applied on a floating-point value the result will be inaccurate.

Program format:

value **not**

Binary format:

(0x3F)

Logical negation of the operand.

The instruction will return 1, if the operand has value 0, and 0 in any other case.

9 ARITHMETIC OPERATIONS

Program format:

value1 value2 **-**

Binary format:

(0x40)

Subtract the “value2” operand from “value1” and return the result.

Program format:

value1 value2 **+**

Binary format:

(0x41)

Arithmetic addition of two operands and return the result.

Program format:

value **--**

Binary format:

(0x48)

Decrease the value on the top of the data stack by 1.

Program format:

value **++**

Binary format:

(0x49)

Increase the value on the top of the data stack by 1.

Program format:

value1 value2 *****

Binary format:

(0x42)

Multiply the two operands and return the result.

Program format:

value1 value2 **/**

Binary format:

(0x43)

Divide the “value1” operand by “value2” and return the result.

In case when “value2” has value 0, there are three possible configurations of the outcome:

a) A floating-point NAN value is always returned.
b) (default) NAN is returned when the operands are of floating-point type, and a system exception in case of integer.
c) A system exception is always triggered.

Program format:

value1 value2 **idiv**

Binary format:

(0x44)

Integer division of the “value1” operand by “value2” and return the result.

In case when “value2” has value 0, there are two possible configurations of the outcome:

a) (default) A system exception is triggered.
b) A floating-point NAN value is returned.

Program format:

value1 value2 **mod**

Binary format:

(0x45)

Integer “modulo” operation of the “value1” operand by “value2” and return the result.

In case when “value2” has value 0, there are two possible configurations of the outcome:

a) (default) A system exception is triggered.
b) A floating-point NAN value is returned.

10 FLOATING-POINT OPERATIONS

Program format:

-INF

Binary format:

(0x50)

Puts in the data stack a ***-Infinity*** value.

Program format:

+INF

Binary format:

(0x51)

Puts in the data stack a ***+Infinity*** value.

Program format:

NaN

Binary format:

(0x53)

Puts in the data stack a ***NaN*** (Not-a-Number) value.

Program format:

_pi

Binary format:

(0x58)

Puts in the data stack a ***'pi'*** value. The value of 'pi' is stored as 3.14159265358979323846264338327950288.

Program format:

_e

Binary format:

(0x59)

Puts in the data stack an ***'e'*** value. The value of 'e' is stored as 2.71828182845904523536028747135266249.

Program format:

rand

Binary format:

(0x60)

Puts in the data stack a randomly generated floating-point number with value between 0.0 and 1.0. While the value may be equal to 0.0, it is always lower than 1.0.

Program format:

value **abs**

Binary format:

(0x63)

Returns the absolute value of the parameter.

Program format:

value **round**

Binary format:

(0x64)

Round the parameter to the nearest integer number.

Program format:

value **trunc**

Binary format:

(0x65)

Truncate the parameter to the nearest integer number.

Program format:

value **fract**

Binary format:

(0x66)

Return only the fraction of the parameter.

Program format:

value **logd**

Binary format:

(0x68)

Decimal logarithm with base 10.

Program format:

value **logn**

Binary format:

(0x69)

Natural logarithm with base ***'e'***.

Program format:

value **exp**

Binary format:

(0x6A)

Exponent with base ***e***. The function returns result e^{value}.

Program format:

value1 value2 **power**

Binary format:

(0x6B)

Raise the parameter *value1* to the power of *value2*. The function returns result $value1^{value2}$.

Program format:

value **root2**

Binary format:

(0x6C)

Square root of the input parameter. The function returns result $\sqrt[2]{value}$

Program format:

value **root3**

Binary format:

(0x6D)

Square root of the input parameter. The function returns result $\sqrt[3]{value}$

Program format:

value **sin**

Binary format:

(0x70)

Sine function.

Program format:

value **asin**

Binary format:

(0x72)

Arc sine function.

Program format:

value **hsin**

Binary format:

(0x74)

Hyperbolic sine function.

Program format:

value **cos**

Binary format:

(0x71)

Cosine function.

Program format:

value **acos**

Binary format:

(0x73)

Arc cosine function.

Program format:

value **hcos**

Binary format:

(0x75)

Hyperbolic cosine function.

Program format:

value **tan**

Binary format:

(0x78)

Tangent function.

Program format:

value **atan**

Binary format:

(0x7A)

Arc tangent function.

Program format:

value **htan**

Binary format:

(0x7C)

Hyperbolic tangent function.

Program format:

value **cotan**

Binary format:

(0x79)

Cotangent function.

11 MEMORY AND STRING OPERATIONS

Program format:

start size value **meminit**

Binary format:

(0x88)

Initialise memory area with a specified byte value.

Program format:

dest src size **memcopy**

Binary format:

(0x8A)

Copy memory area with specified size, starting from the source address and into an area starting from the destination address. The two areas can be overlapping.

Program format:

str **strlen**

Binary format:

(0x89)

Return the length of a zero-terminated string in bytes. The closing 0 byte in the string is not included in the result.

Program format:

dest src **strcopy**

Binary format:

(0x8B)

Copy zero-terminated string, starting from the source address and into an area starting from the destination address. The two strings can be overlapping, and the closing byte 0 is also copied.

Since the length of zero-terminated strings is unknown, in situations where the destination address is higher than the source address, an additional operation of determining the length of the source string is performed. This is needed because in such case the copying starts from the end and moves backwards in order to ensure the integrity of the content in overlapping areas.

Program format:

area1 area2 size **memcomp**

Binary format:

(0x8C)

Compare two memory areas and return a result which is 0 if they are a match, or the index (indexes start from 1, like in all arrays) of the first difference, which is a non-zero result of subtracting a byte in *area2* from a byte in *area1*, both with the same index.

Program format:

str1 str2 **strcomp**

Binary format:

(0x8D)

Compare two zero-terminated strings and return a result which is 0 if they are a match, or the index (indexes start from 1, like in all arrays) of the first difference, which is a non-zero result of subtracting a byte in *str2* from a byte in *str1*, both with the same index.

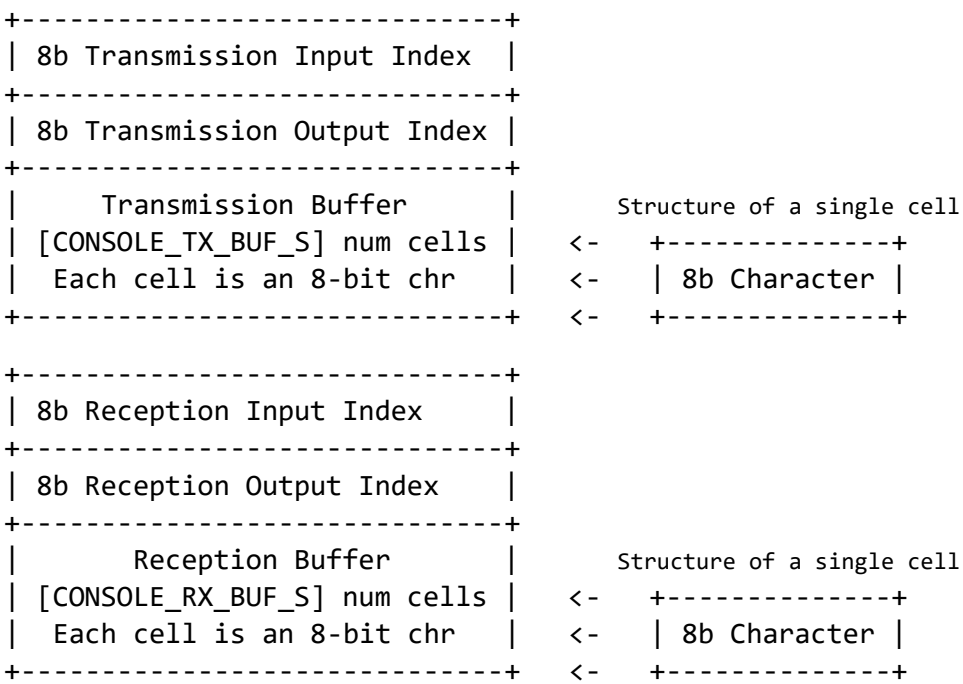
12 SYSTEM CONSOLE

The system console has the purpose to provide an always-on channel for communication with the Centipede processor.

From a hardware perspective, it is a fixed serial channel which conforms to the typical RS232-TTL specifications with signal levels ranging from 0V (V_L) to the V_{dd} (V_H typically 3.3V or 5V) level of the Centipede microprocessor.

In this protocol, the idle state of a line is encoded with V_H level. Logical 0 is encoded with V_L level and logical 1 - with V_H level.

The system console occupies a separate block, unrelated to the core resources. Internally organised as a 64-byte circular input buffer for the incoming data and a 64-byte circular output buffer for the outgoing data. When the input buffer is full, no further data from the CRX line is accepted until an available position in the input buffer becomes open. Opposite to that, when outputting data the Centipede processor delays further execution of the thread (only!) and waits until an available position in the output buffer becomes available.



Two data lines are needed: ***CTX*** for output data from the Centipede processor to another device, and ***CRX*** for input data from another device into the Centipede processor.

The serial console in the Centipede processor has fixed parameters 115200 baud, 8 data bits, no parity, one stop bit.

getchr

Binary format:

(0xF8)

Returns a 16-bit integer, which could be an 8-bit value (0 … 255) of a character who is waiting read from the console input buffer. The character is removed from the buffer. In case the buffer is empty, a value of -1 is returned instead.

peekchr

Binary format:

(0xF9)

Returns a 16-bit integer, which could be an 8-bit value (0 … 255) of a character who is waiting read from the console input buffer. The character remains in the buffer. In case the buffer is empty, a value of -1 is returned instead.

value **putchr**

Binary format:

(0xFA)

An 8-bit input value is output on the serial console in the form of a single character.

addr **putstr**

Binary format:

(0xFB)

The input is a 32-bit unsigned memory address, pointing to a zero-terminated string of characters, which is output on the console.

The output of strings is performed in parallel with other executed system cores. In case strings are output simultaneously by more than one core, the result is an unpredictable mixture of all strings.

value **putval**

Binary format:

(0xFC)

The *putval* instruction considers the data type of the input value. All values are output in decimal base.

With floating-point numbers the output is determined by the most suitable representation of the number. This means, whole numbers are output in the same way as integer and without a decimal point. Large numbers are output in an exponential form.

Floating point values NAN and INF are output as text.

All values are first converted into their text representation and stored into an internal buffer, then output as normal strings.

13 INSTRUCTION CODE MAP

A full map of the instruction codes is shown in the table below. Among other features, the Centipede instructions are designed in such way, so a program source can be retrieved and listed in its exact form directly from the binary code. This is intended to serve in development of self-contained systems where an external compiler or debugger may not be available.

H\L	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	reset	cores	active	current	return	run	stop	arun	clear	depth	dup	drop	over	swap		
10	const0	const1	cint8	cint16	cint32	cint64	cfloat		where		datai8	datai16	datai32	datai64	dataf	
20	free	vint1	vint8	vint16	vint32	vint64	vfloat		goto	call	agoto	acall	gotoif	callif	agotoif	acallif
30	==	<>	<	<=	>	>=	=	?	shiftl	shiftr	or	exor	and	invert		not
40	-	+	*	/	idiv	mod			--	++	v--	v++			usec	
50	-INF	+INF		NaN					_pi	_e						
60	rand			abs	round	trunc	fract		logd	logn	exp	power	root2	root3		
70	sin	cos	asin	acos	hsin	hcos			tan	cotan	atan		htan			
80	Dbase	Pbase	Dsize	Psize					meminit	strlen	memcopy	strcopy	memcomp	strcomp		
90																
A0																
B0																
C0																
D0																
E0																
F0	nop	.	setbin	setoct	sethex	setchr	setstr	#	getch	peekch	putch	putstr	putval	!	@	END

Note1: The light-greyed codes are reserved for use by platform-specific instructions.

Note2: The dark-greyed codes are used by the source disassembler.

14 THE MONITOR

The Monitor is a built-in resident program in ROM which takes care of providing basic interface between the Centipede processor and the user. With the monitor, the user can inspect the content of any memory in the system, erase the program memory, upload new program, and perform simple disassembly operations on the code.

The monitor is activated when the program memory is blank (by checking for 0xFF in the first four bytes in the ROM), or by sending a 27 (0x1B, ASCII 'ESC') code via the serial console during the first 1000 milliseconds after reset. When activated, the monitor takes over the system and any loaded user program is stopped from execution.

The Monitor is interactively controlled through commands send by the user via the console port. All commands are in text format and executed upon the receival of a 13 (0x0D, ASCII 'CR') character. Only one command per line is allowed.

The serial console in the Centipede processor has fixed parameters 115200 baud, 8 data bits, no parity, one stop bit.

Parameters for any command in the Monitor are always given in hexadecimal format and without a prefix, i.e., an example parameter with value 0x7A8C is given just as 7A8C.

Parameters are separated from commands and from each other by space characters.

When activated, the monitor announces that by sending a message “Centipede Monitor” to the console. From that point, the user is able to type in commands interactively.

14.1 Help

Syntax:

H

The command has no parameters and outputs information about system memories and available commands in the Monitor.

Executing the Help command produces output like this:

```
ROM start address 9D000000, length 1C800
RAM start address A0004000, length C000

[address]    Z  erase the entire program memory
                the parameter must match the ROM start address
[[address]]  U  upload program code
                no parameter defaults to the ROM start address
[[address]] [count]  V  view memory
[[address]] [count]  L  list code
H or ?  this help    X  exit and reset

Predefined constants: '.' last value, 'RAM', 'ROM'
```

Some pre-defined constants are included for easier operation. Those include one for the start address of the RAM and ROM in the system. Thus, for example, assuming the system has the ROM starting from physical address 0x4A000000, instead of typing '*4A000000*' as parameter, the user can replace it with the easier '*ROM*'.

Another interface aid is the '.' (single dot), which is equivalent to the last used value for the parameter. As an example, if the user had had a command:

```
5600C020 40 V
```

This time round, typing:

```
. 80 V
```

will be equivalent to typing '*5600C020 80 V*'.

14.2 Exit and reset

Syntax:

X

The command has no parameters and performs exit from the Monitor and system reset.

14.3 Full Memory Erase

Syntax:

[addr] Z

This command erases the entire non-volatile memory in the system. It requires one parameter, which needs to match the start address of the ROM in the system.

14.4 View Memory

Syntax:

[address] [count] V

In its full format, the command dumps the memory content on to the console, starting from a given address and output a specified number of bytes. Output is formatted in lines of up to 16 bytes each.

Syntax:

[address] V

Only a single of up to 16 bytes starting from a given address, is displayed

Syntax:

V

The command with no parameters displays a single line of up to 16 bytes, starting from the address which follows a previously displayed line.

14.5 List Centipede Source Code

Syntax:

[address] [count] L

In its full format, the command lists a specified number of Centipede Assembler source lines starting from a given address. The listing may finish preliminary before the specified count if a command END is reached.

Syntax:

[address] L

Only a single source line is listed.

Syntax:

L

The command with no parameters displays a single source line, starting from the address which follows a previously listed line.

14.6 Upload Program Code

Syntax:

[[address]] U

The command is used to upload program code into the system's non-volatile memory. If the address is not specified, it is assumed to be the start of the system ROM. The code to upload is transferred through the same console port via XMODEM protocol. It stops with an error, if a non-existing address is reached, or a certain amount of time has passed with no incoming data. The destination memory is checked first from the supplied start address or the default one, and until the end of the non-volatile memory. Uploading can be performed only if the destination area is blank (see the Z command).

Multiple chunks of code can be uploaded at different addresses, however, since the program memory can be erased only in full, following uploads must be from sequentially increasing addresses in order to satisfy the blank check of the target area.

As an example, if two pieces of code are to be stored in the program memory, the first one is always from its start, and the second follows from a higher memory address.

15 MACRO-ASSEMBLER

NOTE: *The macro-assembler may not be included in all embedded implementations*

The macro-assembler is a simple program taking on the role of translating instructions from program source format into binary executable code. There are a few extensions which are supported in the source to simplify writing program code.

15.1 Numeric Formats

The source code allows use of numeric constants expressed in several different base formats.

Binary numbers are preceded by a ‘0b’ or ‘0B’ sequence and the following number has base 2 and can contain only digits ‘0’ and ‘1’.

Examples for binary number: 0b10110, 0b00010001

Octal numbers are preceded by a ‘0’ character and the following number has base 8 and can contain digits between ‘0’ and ‘7’.

Examples for octal number: 0774, 02567

Decimal numbers are optionally preceded by a ‘0d’ or ‘0D’ sequence, and the following number has base 10 and can contain digits between ‘0’ and ‘9’. When a leading ‘0’ is required, the format needs to be explicitly defined with ‘0d’ or ‘0D’ preceding the number to avoid confusion with octal numbers.

Decimal numbers may be integer or floating-point type. The format of the latter is as follows:

[sign] [iii [.fff [E *or* **e [sign] [eee [.xxx]]]]]**

> *Where:*
> ***sign*** *can be the symbol ‘-’ or ‘+’.*
> ***iii.fff*** *are the integer part and the fraction of the number, accordingly.*
> *The* ***‘E’*** *or* ***‘e’*** *symbol indicates that an exponent is being supplied to the number.*
> ***eee.xxx*** *are the integer part and the fraction of the exponent, accordingly.*

Examples for decimal number: 123, 0d008918, 3.14159, -0.77E-8

Hexadecimal numbers are preceded by a ‘0x’ or ‘0X’ sequence and the following number has base 16 and can contain digits between ‘0’ and ‘9’ as well as the characters ‘A’ to ‘F’ (or ‘a’ to ‘f’).

Examples for hexadecimal number: 0xA34C11, 0x011E

15.2 Individual Characters

It is possible to use individual ASCII characters enclosed in single quotes in format '*character*'. The result is translated into an 8-bit integer constant.

Some important characters, mostly with ASCII code smaller than 32, have predefined constants:

```
'\0'   - ASCII code 0  (character NUL)
'\a'   - ASCII code 7  (alarm)
'\b'   - ASCII code 8  (backspace)
'\t'   - ASCII code 9  (horizontal tabulation)
'\n'   - ASCII code 10 (new line)
'\f'   - ASCII code 12 (form feed)
'\r'   - ASCII code 13 (carriage return)
'\e'   - ASCII code 27 (escape)
'\"'   - ASCII code 34 (the character " 'double quote')
'\''   - ASCII code 39 (the character ' 'single quote')
'\?'   - ASCII code 63 (the character ? 'question mark')
'\\'   - ASCII code 92 (the character \ 'backslash')
```

In addition to the predefined constants, any 8-bit ASCII code could be also entered in its digital form:

'**\x***nn*' with '*nn*' as a two-digit hexadecimal code in the range 00…FF,

or as

'*nnn*' with '*nnn*' as a three-digit decimal code in the range 000...255.

Examples for individual character: 'j', '\n', '\x7A', '\028', '\229'

15.3 Character String Constants

The character strings consist of one or more non-zero 8-bit ASCII characters, and are terminated by a character with code zero, which is added automatically by the macro-assembler. Strings are translated into executable code as **datai8** sequences.

The character strings are enclosed by double quote characters.

Examples for character string: "Hello", "Enter your name below:\r\n"

15.4 Commentaries

The source commentaries help with better understanding what a certain part of the code does. They also could carry important information about input and output parameters.

The source commentaries always start with the **#** character, and span until the end of the current source line. Multi-line commentaries are not

supported. Therefore, each individual commentary line should start with its own ‘#’ character.

Examples for commentary:

```
# This code calculates the average value in an array of FP numbers
# Expected parameters come from the data stack: <address> <count>
# Calculated result is returned in the data stack
```

15.5 Predefined Variables

To help facilitate more readable program code, the macro-assembler includes pre-defined variable names such as v12 or V47 which can be used to refer to the internal variables.

The following is supported (the character 'V' is not case-sensitive):

From **V0** to $\mathbf{V_{(VARIABLES-1)}}$ where the maximum number is determined by the definition of **[VARIABLES].**

15.6 Address Labels

The address labels replace addresses in code with human-friendly names. Once defined, a label can be used to provide the branch address for goto-type and call-type instructions.

The macro-assembler always calculates branches as relative offset to produce a relocatable code.

Labels are defined as a ‘**@**’ character immediately followed by the label as text. However, when used in the code, the leading ‘@’ is omitted.

The labels must start with an alphabet letter or a ‘_’ character, and can contain alphabet letters, the digits ‘0’ to ‘9’, or ‘_’ characters. The length of a single label can be theoretically up to 255 characters, although that length is likely further restricted by the specific macro-assembler implementation due to memory considerations.

Example of label definition and use:

```
@Entry       # entry point
... some program code ...
Entry goto   # jump back to the entry point
```

15.7 Named Definitions

Another useful extension of the macro-assembler is the possibility to replace a text word with a pre-defined constant or even a small piece of code until the end of the current source line. This enables more readability in the source code and reduces the chance of errors.

Constants are named as per the following model:

!name value

Definitions always start with a ‘**!**’ character, followed immediately by the name. The name of a constant must begin with an alphabet letter or a ‘_’ character, and can contain alphabet letters, the digits ‘0’ to ‘9’, or ‘_’ characters. It can be theoretically up to 255 characters long, although that length is likely further restricted by the specific macro-assembler implementation due to memory considerations.

What follows the name could be (typically) a numeric constant, or a sequence of instructions.

The definition completes at the end of the source line.

When used in the code, the defining ‘!’ character is omitted.

Previously named constants can also be used in a definition.

Examples of named definition:

```
!c       299792.458  # defines the value of constant ‘c’
!myC     c 1 +       # this defines a macro with value (c+1)
!flags   0b11010001  # integer constant defined as binary number
!CR      '\r'        # gives name to the ASCII code 13
!hello_s "Hello"     # gives name to the a text string
```

16 EXAMPLES

Several simple examples are provided in this chapter to demonstrate how the Centipede processor assembly code looks in real-life applications.

16.1 Counter with output to the console

```
0 V1 vint8          # reserve v1
0 V1 =              # initialise v1 with 0
@again
  V1 ? putval       # print the value from v1
  "\r\n" putstr     # go to a new line
  V1 v++            # increase v1 by 1
  V1 ? 20 <         # check if v1 is less than 20
again gotoif        # repeat while true
V1 free             # free up v1
END                 # automatically added END
```

16.2 Bubble sort of a random array and output the result

```
0 V1 vint8          # reserve v1
100 V2 vint32       # reserve array v2[100]
Dbase 0 V2 =        # set start address for the array (v2[0] = start of RAM)

# initialise the array with random values
"Initial array: " putstr
1 V1 =              # initialise v1 with 1
@loop_init
  rand 1000 * V1 ? V2 =       # v2[v1] = random number 0 ... 999
  V1 ? V2 ? putval "  " putstr
  V1 v++            # increase the index in v1 by 1
  V1 ? 100 <=       # check if v1 is less than or equal to 100
loop_init gotoif    # repeat so v2[1] … v2[100] all get values
"\r\n" putstr

# bubble sort
1 V1 =              # initialise v1 with 1
0 V3 vint32         # reserve v3
@loop_sort
  V1 ? V2 ?         # get V2[V1] in the stack
  V1 ? 1 + V2 ?     # get V2[V1+1] in the stack
  <= skip gotoif    # no swapping if the two values are in right order
    V1 ? V2 ? V3 =              # v3 = v2[v1]
    V1 ? 1 + V2 ? V1 ? V2 =     # v2[v1] = v2[v1+1]
    V3 ? V1 ? 1 + V2 =          # v2[v1+1] = v3
    1 V1 =                      # v1 = 1
    loop_sort goto              # start over
  @skip
```

```
  V1 v++          # increase the index by 1
  V1 ? 100 <      # check if the index is less than 100
loop_sort gotoif  # repeat if true

# print the sorted array
"Sorted array: " putstr
1 V1 =            # initialise v1 with 1
@loop_print
  V1 ? V2 ? putval "  " putstr
  V1 v++          # increase the index in v1 by 1
  V1 ? 100 <=     # check if v1 is less than or equal to 100
loop_print gotoif # repeat so v2[1] … v2[100] all get values
"\r\n" putstr

V3 free           # release v3
V2 free           # release v2 (note: the array remains unaffected)
V1 free           # release v1
END               # automatically added END
```