The ZTH1 Forth cross-compiler, v. 2.0

S. Morel, Zthorus-Labs, 2022-04-08

(Last document revision: 2023-01-25)

1 Introduction

This document describes the Forth language cross-compiler for the ZTH1 CPU. It runs on a Linux/Unix platform. It can be used to develop code for ZTH1-based machines, like the Zythium-1 microcomputer or the VectorUGo-2 game console, by generating `.mif` files (memory contents) that can be uploaded using Intel Quartus-Prime.
The fact that the ZTH1 CPU uses a stack of registers makes the Forth language optimal for software development, since Forth is based on the handling of a stack of numerical values.

2 Installation

The ZTH1 Forth cross-compiler consists of a C source file: `zth1f.c`. Once copied into a directory of a Linux/Unix computer, it can be compiled by:

```
gcc zth1f.c -o zth1f
```

The ZTH1 Forth cross-compiler also requires the OS files to be present in the same directory as `zth1f`. These files contain the OS of the Zythium-1 and VectorUGo-2 target machines and will be merged into the generated `mif` file. The names of the OS files are:

- For Zythium-1: `zth1_os_rom.mif`, `zth1_os_ram_h.mif`, `zth1_os_ram_l.mif`.
- For VectorUGo-2: `vug_os_rom.mif`, `vug_os_ram_h.mif`, `vug_os_ram_l.mif`.

3 Usage

Source code in Forth for the ZTH1 can be written with any editor, like `vi` or `emacs`. The suffix `.fth` can be used for these source files. To compile source code for Zythium-1, the compiller is launched by typing:

```
./zth1f <source file>
```

For example (if the source file is located in the same directory as the compiler):

```
./zth1f my_code.fth
```

To compile code for VectorUGo-2, the `-v` option must be added to the command line:

```
./zth1f -v <source file>
```

If the processing of the source file has been successful, the object files `rom.mif`, `ram_h.mif` and `ram_l.mif` are generated. These files correspond respectively to the memory contents of the instruction ROM, the high-byte data RAM bank and the low-byte data RAM bank of the ZTH1-based machine. They can be copied into the Intel Quartus-Prime project of the machine (and then be merged into the `.sof` file that will be uploaded into the FPGA implementing this machine).

An example of ZTH1 Forth program is the implementation for Zythium-1 of the Donkey-Kong arcade game which can be found at: `github.com/zthorus/Zythium1-Donkey-Kong`

Likewise, an example of ZTH1 Forth program for VectorUGo-2 is the adaption of the Asteroids arcade game which can be found at: `github.com/zthorus/VUG2-Asteroids`

4 Caveat and limitations

In its current version, this Forth cross-compiler does not strictly follow the standards of other Forth compilers. It has been designed for the ZTH1 architecture as an efficient alternative to the assembly language and has the following limitations and differences, compared to other Forth compilers and interpreters:

1. Variables are defined with a non-standard syntax and it is not possible to define the initial value of a variable from the top of the stack.
2. Variables can only be signed or unsigned 16-bit integers. No 8-bit-integer or floating-point variables are supported.
3. Data are defined with a non-standard syntax.
4. Some new structuring words for branching (i.e., variations of `if` and `until`) have been introduced to optimize the ZTH1 object code.
5. Inclusion of source files from within a source-file is possible.
6. The source code must include the definition of a word called `main` (like in C language) which is executed when the machine is booted.
7. Special care must be taken when using the stack for arithmetic operations: as the Forth stack is actually the ZTH1 CPU register stacks, there are only 8 levels available.
8. Likewise, nested calls to sub-routines are possible only to the 8th level. Considering the use by the routines of the Zythium-1 or VectorUGo-2 OS, user-defined words shall not exceed nested calls to the 4th level. Recursive algorithms are not allowed by the compiler (i.e., the definition of a word cannot contain a call to the word itself).

5 ZTH1 Forth syntax

5.1 General structure

The code written in a `.fth` source file consists of "atoms". Atoms are groups of characters having ASCII codes (in decimal) between 33 and 126 (included). Atoms can be integer values (written in decimal or hexadecimal), Forth words (base or user-defined), meta-characters, parts of character strings, file names. Atoms can be separated by any number of formatting characters like spaces, tabs, carriage-returns.

Comments in the source file must be framed by the atoms '`(`' and '`)`' (standard Forth syntax). In the following code excerpts, comments have to be replaced by the actual code.

The minimal content of a `.fth` file would look like:

```
: main
  ( definition of the main word )
;
```

Likewise, the other words (that are called by the `main` word) are defined by the atom '`:`' followed by the name of the new word, then by the code (i.e., a list of operation on the stack) and terminated by the atom '`;`'. In the same source file, the `main` word can be defined after or before the other words. The ZTH1 Forth compiler is case-insensitive for the words, the variables, the constants and the hexadecimal values.

It is possible to include from within a `.fth` file the content of other `.fth` files with the sequence of atoms:

```
# ( list of file names separated by space characters ) ;
```

For example, specific user-defined libraries of Forth words can be included by:

```
# math_lib.fth graph_lib.fth string_lib.fth ;
```

5.2 Numerical values

By default, numerical values are decimal on 16 bits. They can be either positive or negative. It is possible to use a numerical value in hexadecimal by using the prefix '`$`'.

Within the definition of a word, and unless it is within the definition of a local constant, any atom consisting of a numerical value means that this value has to be pushed into the top of the stack.

5.3 Variables

Variables are defined by:

```
variable ( name of the variable )
```

The following is also valid:

```
var ( name of the variable )
```

Unlike standard Forth, and because the ZTH1 Forth is compiled. it is not possible to give to a variable, when it is declared, the current value of the top of the stack as default value.

All variables are 16-bit words. They can represent a numerical value or an address in data RAM of a value (i.e., a pointer). It is therefore possible to create arrays of any dimensions (by combinations of the '`@`' and '`+`' Forth words.

Variables can be either global are local. A global variable is defined outside the definition of a word and can be used by any word within its definition. A local variable is defined inside the definition of a word (normally, at the beginning) and can only be used by this word.

In the following excerpt of code,

```
: word1
  var foo
```

```
  ( list of atoms defining word1 )
;

: word2
  var foo
  ( list of atoms defining word2 )
  word1
  ( rest of list of atoms defining word 2 )
;
```

it is not a problem for the compiler that `word2` calls `word1` if both words use local variables with the same name. Any action on `foo` during the execution of `word1` will not affect the value of `foo` in `word2`.

5.4 Constants

Constants are defined by:

```
constant ( name of the constant ) ( value of the constant )
```

The following is also valid:

```
cst ( name of the constant ) ( value of the constant )
```

Unlike standard Forth, and because the ZTH1 Forth is compiled. it is not possible to give to a constant, when it is declared, the current value of the top of the stack.

All constants are 16-bit words. They can represent a numerical value or an address in data RAM of a value (i.e., they can represent a variable for which memory space has already been allocated by the system. Hence a constant can be used as a system variable).

Constants can be either global or local. A global constant is defined outside the definition of a word and can be used by any word within its definition. A local constant is defined inside the definition of a word (normally, at the beginning) and can only be used by this word.

In the following excerpt of code,

```
: word1
  cst foo 1000
  ( list of atoms defining word1 )
;

: word2
  cst foo $abcd
  ( list of atoms defining word2 )
  word1
  ( rest of list of atoms defining word 2 )
;
```

it is not a problem for the compiler that `word2` calls `word1` if both words use local constants with the same name. Any action on `foo` during the execution of `word1` will not affect the value of `foo` in `word2`.

5.5 Data

With the ZTH1 CPU, data are stored in a different memory space than the one used for the instructions and are addressed separately (Harvard architecture). In the ZTH1 Forth, data are defined, outside any definition of a word, by the sequence of atoms:

```
data ( address ) ( set of values )
```

The address can be either a 16-bit word, in decimal or in hexadecimal (using the '$' prefix), or any atom considered as a "label". In the later case, the compiler will automatically give an address to the declared label (which can be used as a global variable in the word definitions) and will allocate the memory space needed for the set of values.
The set of values can be either a 16-bit word in decimal or a sequence of 16-bit words in hexadecimal that are not separated by space characters and which uses the '$' prefix. For example:

```
data array1 $ffffc0001846
```

The set of values can also be a character string (characters are alternatively stored in high-bytes and low-bytes of the 16-bit memory words) by using the '"' prefix. For example:

```
data string1 "ABCCDEF
```

No ending '"' is needed. Space characters are not allowed in strings defined this way.
It is also possible, instead of defining a set of values, to just allocate a block of memory, by specifying its size (in number of 16-bit words) written in decimal with the prefix '*'. For example:

```
data array2 *25
```

6 Base ZTH1 Forth dictionary

In the following table, the Forth stack is described by the stack of ZTH1 registers ($A, B, C...$), where A is the top of the stack. The programmer shall consider that all the register values used as arguments by a Forth word are dropped after the execution of this word.

6.1 Standard Forth words and meta-characters

`!`	Write value of B at the address specified by A. Do an implicit double `drop` afterwards.
`$`$nnnn$	$(A, B, C...) \Rightarrow (nnnn, A, B, C...)$. $nnnn$ is hexadecimal.
`(`	Start a comment. Next atoms will be ignored by the compiler until '`)`' is met.
`)`	Terminate a comment.
`*`	$(A, B, C, ...) \Rightarrow (A * B, C, ...)$. Signed multiplication. A and B shall be between -128 and +127.
`+`	$(A, B, C, ...) \Rightarrow (A + B, C, ...)$.
`-`	$(A, B, C, ...) \Rightarrow (A - B, C, ...)$.
`.`	Display (in decimal) value of A. Do an implicit `drop` afterwards (syntax of this word on VectorUGo-2 is different and non-standard).
`."`	Display a character string defined by the next atoms (until the character '`"`' is met at the end of an atom). Space characters right after '`."`' will be ignored until a non-space character is met and then all space characters are considered until the end of the string (syntax of this word on VectorUGo-2 is different and non-standard).
`/`	$(A, B, C, ...) \Rightarrow (A/B, C, ...)$. Quotient of Euclidian division (A and B are considered as unsigned integers).
`/mod`	$(A, B, C, ...) \Rightarrow (A \ mod \ B, C, ...)$. Remainder of Euclidian division (A and B are considered as unsigned integers).
`1+`	$(A, B, ...) \Rightarrow (A + 1, B, ...)$.
`1-`	$(A, B, ...) \Rightarrow (A - 1, B, ...)$.
`:`	Start the definition of a word (the next atom is the name of the word).
`;`	Terminate the definition of a word.
`<`	$(A, B, C, ...) \Rightarrow (1, C, ...)$ if $A < B$; $(0, C, ...)$ otherwise.
`=`	$(A, B, C, ...) \Rightarrow (1, C, ...)$ if $A = B$; $(0, C, ...)$ otherwise.
`>`	$(A, B, C, ...) \Rightarrow (1, C, ...)$ if $A > B$; $(0, C, ...)$ otherwise.
`@`	$A \leftarrow$ content of address specified by A.

`and`	$(A, B, C, ...) \Rightarrow (A\&B, C, ...)$.
`at`	Set screen location to display text at $(x = A, y = B)$. $(0, 0)$ is the top-left corner on Zythium-1 and bottom-left corner on VectorUGo-2. Do an implicit double `drop` afterwards.
`begin`	Start of `begin`/`until` loop.
`cls`	Erase screen (implemented on Zythium-1 only).
`cr`	Move to next line to display text (implemented on Zythium-1 only).
`do`	Start of `do`/`loop` with first value of iteration = A and last value = B. Do an implicit double `drop` afterwards.
`drop`	$(A, B, C, ...) \Rightarrow (B, C, ...)$.
`dup`	$(A, B, ...) \Rightarrow (A, A, B, ...)$.
`else`	Execute next words (until `then` is met) when the condition of the previous `if` is not met.
`emit`	Display character for which A = ASCII code. Do an implicit `drop` afterwards (syntax of this word on VectorUGo-2 is different and non-standard).
`i`	$(A, B, C, ...) \Rightarrow (i, A, B, C, ...)$. i= current value of iteration inside a `do`/`loop`.
`if`	Execute next words (until `then` or `else` is met) if $A = 1$.
`loop`	Repeat execution of words after the precious `do` if the maximum value of iteration is not reached.
`neg`	$A \leftarrow -A$.
`not`	Toggle all the bits of A.
`or`	$(A, B, C, ...) \Rightarrow (A\|B, C, ...)$.
`over`	$(A, B, C, ...) \Rightarrow (B, A, B, C, ...)$.
`swap`	$(A, B, C, ...) \Rightarrow (B, A, C, ...)$.
`then`	Terminate an `if` or an `if`/`else` sequence.
`u*`	$(A, B, C, ...) \Rightarrow (A * B, C, ...)$. Unsigned multiplication. A and B shall be less than 256.
`until`	Repeat execution of words after the previous `begin` if $A \neq 1$. Do an implicit `drop` afterwards (whatever the value of A).
`xor`	$(A, B, C, ...) \Rightarrow (A\hat{}B, C, ...)$.

6.2 Non-standard Forth words and meta-characters

`#`	Compile source files specified by the next atoms until `;` is met. Source file names cannot include space characters.
`2*`	$A \leftarrow 2 * A$. A is considered as an unsigned integer.
`2/`	$A \leftarrow A/2$. A is considered as an unsigned integer and is rounded down if odd.
`constant`	Define a global or local constant (the name of this constant will be the next atom, and its value will be the atom after).
`cst`	Same as `constant`.
`if!=`	Execute next words (until `then` or `else` is met) if $A \neq B$.
`if<`	Execute next words (until `then` or `else` is met) if $A < B$.
`if<=`	Execute next words (until `then` or `else` is met) if $A \leq B$.
`if=`	Execute next words (until `then` or `else` is met) if $A = B$.
`if>`	Execute next words (until `then` or `else` is met) if $A > B$.
`if>=`	Execute next words (until `then` or `else` is met) if $A \geq B$.
`rolldn3`	$(A, B, C, ...) \Rightarrow (C, A, B, ...)$.
`rolldn4`	$(A, B, C, D, ...) \Rightarrow (D, A, B, C, ...)$.
`rollup3`	$(A, B, C, ...) \Rightarrow (B, C, A, ...)$.
`rollup4`	$(A, B, C, D, ...) \Rightarrow (B, C, D, A, ...)$.
`until!=`	Repeat execution of words after the previous `begin` if the condition $A \neq B$ is not met.
`until<`	Repeat execution of words after the previous `begin` if the condition $A < B$ is not met.
`until<=`	Repeat execution of words after the previous `begin` if the condition $A \leq B$ is not met.
`until=`	Repeat execution of words after the previous `begin` if the condition $A = B$ is not met.
`until>`	Repeat execution of words after the previous `begin` if the condition $A > B$ is not met.
`until>=`	Repeat execution of words after the previous `begin` if the condition $A \geq B$ is not met.
`var`	Declare a global or local variable (the name of this variable will be the next atom).
`variable`	Same as `var`.

6.3 Zythium-1 OS words

Please refer to Section 7 and to the Zythium-1 manual for details on the use of these words. The Zythium-1 manual is available at: `github.com/zthorus/zythium-1/zythium1.pdf`

`colorsprite`	Set color of sprite A to B (according to the OS color look-up table).
`defsprite`	Define for sprite A the bitmap (made of four 16-bit words) defined at address B. Do an implicit double `drop` afterwards.
`hidesprite`	Hide sprite A. Do an impicit `drop` afterwards.
`joystick`	$(A, B, C, ...) \Rightarrow (j, A, B, C, ...)$ j=state of joystick=$8 \times down + 4 \times up + 2 \times left + right$. Direction bits are active at 0.
`putsprite`	Display sprite A at $(x = B, y = C)$. $(0, 0)$ is the top-left corner. Do an implicit triple `drop` afterwards.

6.4 VectorUGo-2 OS words

Please refer to Section 8 for details on the use of these words.

`.`	Display (in decimal) value of B using the text handle A. Do an implicit `drop` afterwards.
`."`	Display a character string defined by the next atoms (until the character '`"`' is met at the end of an atom). Space characters right after '`."`' will be ignored until a non-space character is met and then all space characters are considered until the end of the string. After routine call, a text handle is returned in A, which must be stored into a variable to later manipulate or delete the display of the character string.
`closesprite`	Make sure a sprite (identified by the handle in A) consisting of a polygon appears closed by calculating its last vector from the sum of its other vectors. Do an implicit double `drop` afterwards.
`collision`	Detect if any collision of the sprite identified by the handle in A and the sprite identified by the handle in B has occurred. After routine call, 0 is returned in A if no collision occurred. Otherwise, A (resp. B) contains the index of the vector of sprite A (resp. B) that is implied in the collision.
`copysprite`	Copy the vectors of the sprite or of the text item, as currently displayed and identified by the handle stored in A, into the memory block (previously allocated with `data` and the '`*`' prefix) starting at address stored in B. This word can be used to transform a text displayed with '`."`' into a full sprite. Do an implicit double `drop` afterwards.
`defsprite`	Define a sprite made of A vectors (including the radial vector) with the sequence of vectors as defined at address B. After routine call, a sprite handle is returned in A must be stored into a variable to later manipulate the sprite.
`delsprite`	Delete the sprite identified by the handle stored in A. The sprite becomes immediately invisible and its handle cannot be used anymore. Do an implicit `drop` afterwards.
`emit`	Modify a text-item directly in the vector-table, from the address stored in B, to display the character having the ASCII code indicated by A. Do an implicit double drop afterwards.
`hidesprite`	Make a sprite invisible on the display device (however the sprite is not deleted and can be made visible again). This sprite is identified by the handle stored in A. Do an implicit `drop` afterwards.
`joystick`	$(A, B, C, ...) \Rightarrow (j, A, B, C, ...)$ j=state of joystick=$16 \times right + 8 \times left + 4 \times down + 2 \times up + fire$. Direction bits are active at 0.
`masksprite`	Make visible some of the vectors of the sprite identified by handle in A and make invisible other vectors. B must contain the address of the "mask" (indicating the state for each vector) to be used. Do an implicit double `drop` afterwards.
`putsprite`	Display the sprite (if it has previously been made visible) identified by the handle stored in A at $(x = C, y = B)$. $(0, 0)$ is the top-left corner. Do an implicit triple `drop` afterwards.
`rcos`	$(A, B, C...) \Rightarrow (B \times \cos(A), C, ...)$. A is expressed in units of $\pi/128$ (e.g., $A = 64$ means $\pi/2$ rad or $90°$). Input values of A and B must be between 0 and 255.

`rotscalsprite`	Re-display the sprite defined by the handle A tilted with an angle B and with a scaling factor C. Do an implicit triple `drop` afterwards.
`rsin`	$(A, B, C...) \Rightarrow (B \times \sin(A), C, ...)$. A is expressed in units of $\pi/128$ (e.g., $A = 64$ means $\pi/2$ rad or $90°$). Input values of A and B must be between 0 and 255.
`showsprite`	Make a sprite visible (all its vectors, except the position and radial vectors, become visible) on the display device. This sprite is identified by the handle stored in A. Do an implicit `drop` afterwards.
`switch`	Swap the low-byte and high-byte parts of A (similar to the `SWA` ZTH1 op-code). This word is useful to directly manipulate 16-bit vector coordinates (where x is stored in the high-byte and y is stored in the low-byte of a 16-bit word).

7 Notes on the Zythium-1 OS

The OS for the Zythium-1 computer consists of routines (coded in ZTH1 assembly language) that are called by the following ZTH1 base Forth words: '`*`', '`/`', `/mod`, '`.`' '`."`', `at`, `cls`, `cr`, `emit`, `u*`, `colorsprite`, `defsprite`, `putsprite`, `hidesprite`, `joystick`. It also contains the bitmap of the character set to display text.

7.1 Colors

The color look-up table of the Zythium-1 OS is the following:

Value	Color
0	Black
1	Dark blue
2	Yellow
3	Red
4	Pink
5	Cyan
6	Orange
7	Bright blue
8	Grey
9	White
10	Purple
11	Green

Setting the color of a sprite is done by using the Forth word `colorsprite` with the top of the stack containing the sprite number and the second level of the stack containing the value of the color (as indicated by the table above).

To set the color of text (displayed by the '`.`', '`."`' and `emit` Forth words), the color value has to be written at the address $1873. The high-byte is the color of the characters and the low-byte is the color of the background.

7.2 Redefining the character set

Any character can be redefined (to display graphic elements) by:

```
data ( address ) ( bitmap )
```

The address shall be: $1E80 + 4×(ASCII code of character - 32)

The bitmap consists of a lump of eight 8-bit hexadecimal values, defining the new bitmap of the character from the top to the bottom.

8 Notes on the VectorUGo-2 OS

The OS for the VectorUGo-2 console consists of routines (coded in ZTH1 assembly language) that are called by the following ZTH1 base Forth words: '`*`', '`/`', `/mod`, '`.`' '`."`', `at`, `emit`, `u*`, `closesprite`, `collision`, `copysprite`, `defsprite`, `delsprite`, `hidesprite`, `joystick`. `masksprite`, `putsprite`, `rcos`, `rotscalsprite`, `rsin`, `showsprite`. It also contains the vector sequences of the character set to display text (currently, only digits and uppercase letters can be displayed).

8.1 Sprites

In the VectorUGo-2 OS, a sprite consists of a sequence of vectors displayed one after the other. Each vector has an x component (signed integer from -128 to +127), a y component (signed integer from -128 to +127) and a visibility state (either 0 or 1). For each sprite, the sequence of vectors is:

- A "position" vector, with $x > 0$ and $y > 0$. This vector is invisible.
- A "radial" vector, going from the end of the position vector to the origin of the first "body" vector. This vector is by default invisible. Its origin can be considered as the "center of gravity" of the sprite and can be used to rotate the sprite.
- A set of "body" vectors that are visible and define the shape of the sprite.
- Three "beam-homing" vectors that are used to return the electron beam of the display device (i.e., an oscilloscope featuring an XY mode and a Z input) to a home position, avoid accumulation of errors that would lead to bas positioning of the sprites displayed after.

A part of the RAM of VectorUGo-2 is reserved by the OS to contain a "vector table" containing all the active vectors (whether they are visible or not) on the display. Another part of the RAM is reserved for a "sprite table" in which important parameters regarding each sprite are stored. The address of each sprite in this table is called a "sprite handle" and is the way to control sprites from the program written in Forth. Sprite handles must be stored into declared Forth variables. A sprite has first to be defined by, for example:

```
data squareshape $0808f00000ff10000010
var mysprite

sqauareshape 5 defsprite mysprite !
```

The shape of the sprite defined by `data` consists of a sequence of 8-bit hexadecimal values which represent alternatively the x and y vector components. This data block, as well as the number of body+radial vectors, are the input of `defsprite`. The handle returned in A (top of the stack) is stored into a variable that will be used for all the operations on the sprite, until it is deleted by `delsprite`.
Once the sprite is defined, it is not displayed until `showsprite` is called:

```
mysprite @ showsprite
```

It can be then positioned on the screen of the display device at a location (x, y) by `putsprite`. x and y must be positive and ranging from 0 to +127. Assuming x and y are variable:

```
x @ y @ mysprite @ putsprite
```

A sprite can be made invisible but still defined and active by using `hidesprite`:

```
mysprite @ hidesprite
```

Some of the body vectors of a sprite that is invisible can be made visible again by defining a binary mask and using `masksprite`:

```
data mymask $a800

mymask mysprite @ masksprite
```

The mask is an N-bit number (N being the number of vectors, body+radial, of the sprite) in which each bit represents the state of a vector (0 for invisible, 1 for visible). The MSB of this number is the state of the radial vector (which can therefore be set visible) and the LSB is the state of the last body vector. This number has to be padded with bits (set to zero) at the end to form one or several 16-bit words. In the above example, the mask is (in binary) 10101, which indicates that the radial vector and the second and fourth body vectors shall be visible. When padded with zeros to form a 16-bit word, this mask becomes 1010100000000000 = $A800.

A sprite can be rotated (tilted) and scaled (to appear larger or smaller) by using `rotscalsprite`. The rotation pivot and the neutral scaling point are the end of the position vector (i.e., the origin of the radial vector) of the sprite. The given angle is expressed in units equal to $\pi/128$ rad. The actual scaling factor has to be multiplied by 16 (for example, a given factor of 32 means "double size", 8 means "half size" and 16 means "no size change"). For example:

```
24 32 mysprite @ rotscalsprite
```

will display the sprite `mysprite` magnified by 50% and tilted counterclockwise with an angle of 45°, compared to its original definition.

Finally, a sprite can be removed from the display and from the sprite-table by `delsprite`:

```
mysprite @ delsprite
```

8.2 Text display

VectorUGo-2 has limited possibilities to display text, partly because only digits (from 0 to 9) and uppercase letters (from A to Z) can be displayed. However, short sequences of characters can be displayed by using the standard '`."`' word. The use of this word returns a "text-item handle" which is actually a sprite handle (because '`."`' calls internally `defsprite`). The difference between a text-item and a user-defined sprite is the absence of user-provided data to define a radial vector and a shape. The OS uses built-in vector sequences to draw the text. However, it is possible to manipulate a text-item as a full sprite by using the `copysprite` word. This allows, for example. the use of `rotscalsprite` on text-items. Here is an example:

```
data mytextvecs *41
var mytext

64 64 at ." Hello" mytext !
mytextvecs mytext @ copysprite
24 32 mytext @ rotscalsprite
```

The size of the allocated block (`mytextvecs` in the example above) must be at least $N \times 8 + 1$, where N is the number of characters of the text-item. Note that the input of the `at` word are in vector units and shall range from 0 to +127 for x and y.

Displaying the value of the top of the stack (in the A register) with the standard '`.`' word is possible if a 5-character text-item has been previously created and its handle stored. The value is always considered as an unsigned 16-bit integer and is displayed with padding zeros. The following code is an example that displays the value of a variable:

```
var mytext
var myvar

64 64 at ." 00000" mytext !
12345 myvar !
myvar @ mytext @ .
```

It is possible to display a single character from its ASCII code with the standard `emit` word, but the procedure is a little complex. First, a text-item must have already been created and `emit` will modify one of the characters of this text-item. Then, it is necessary to find out the address of this character in the vector-table of the display. For example, the following code modifies the second character of a text-item:

```
var mytext

64 64 at ." 00" mytext !

mytext @ 1+ @ 20 + ( modify character N => use [N-1]*16+4 )
49 emit ( display 1 )
```

___oOo___