Notes on the VectorUGo-2 OS

S. Morel, Zthorus-Labs, 2023-02-14

1 Introduction

This document describes the operating system (OS) of the VectorUGo-2 console. This OS has been written in ZTH1 assembly language and provides a set of routines to help with the creation of games. These routines can be called by using Forth words in the ZTH1-Forth compiler. In this document, we will refer to these routines by their label names (starting by '`@`') in the OS source code `vug_os.asm`. The arguments of these routines are passed in the ZTH1 CPU register stack ($A, B, C...$). Some of these routines return results in this stack (the input arguments are deleted from the stack after a call to a routine).

For more information about the hardware and firmware of the VectorUGo-2 console, please read the document that can be found at:

`github.com/zthorus/VectorUGo-2/blob/main/VectorUGo2.pdf`

2 Vectors

A peculiarity of the VectorUGo-2 console is the use of vector display. In this context, a vector is a set of two coordinates x and y, where x and y are 8-bit signed integers (ranging from -128 to +127) that defines the coordinates of the end of a drawn line from a given origin. The RAM of VectorUGo-2 contains a "vector-table" that is continuously read by the vector-display interface. This table contains a sequence of vectors that are drawn one after the other (the end of a vector being the origin of the next one) by applying ramp voltages to the x and y deflection of the display device (i.e., an oscilloscope). A vector can be set either visible or invisible. In the case of invisible vectors, voltages are applied but the spot is turned off (by controlling the z-axis input of the oscilloscope). This allows to have vectors appearing "separated" on the display.

Besides x and y, each vector has a byte called "LFZ" (length factor and z) containing the length-factor of the vector (i.e., the time spent by the vector-display interface to draw the vector) and the z flag (0 if the vector is invisible, 1 if the vector is visible). Most of the VectorUGo-2 OS routines consider that the length-factor is the same for all the vectors and do not use it in their algorithms.

3 Sprites

A sprite is a mobile graphic element in a game. In the case of VectorUGo-2, a sprite is a sequence of vectors, defined as follows:

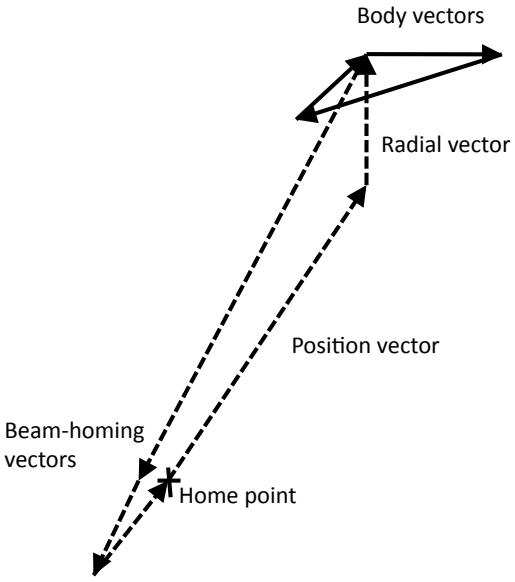
- A "position" vector, with $x > 0$ and $y > 0$. This vector is invisible.
- A "radial" vector, going from the end of the position vector to the origin of the first "body" vector (see below). This vector is by default invisible. Its origin can be considered as the "center of gravity" of the sprite and can be used to rotate and rescale the sprite (the origin of the radial vector is invariant by these transformations).
- 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 to a home position, in order to avoid the accumulation of errors that would lead to bad positioning of the sprites displayed after.

When a sprite is defined (by calling `@defsprite`), an entry is created by the OS in the "sprite-table" stored in RAM. Each entry consists of three 16-bit words:

- First word: the number of body vectors of the sprite + 1 (the radial vector is included).
- Second word: the address of the vectors of the sprite (starting from the position vector), as displayed, in the vector-table.
- Third word: the address of the vectors of the sprite (starting from the radial vector), as defined (before any sprite transformation by rotation or scaling for display). These vectors are defined by the user as a sequence of numbers (`data` ZTH1-Forth word).

The arguments passed in the ZTH1 CPU stack, when `@defsprite` is called, are the number of vectors (body + radial vectors) of the sprite and the address of these vectors that define the sprite. This address shall be the one of a sequence of 16-bit words, with x stored in the high-byte and y stored in the low-byte of each word (that defines a vector). `@defsprite` returns (at the top of the stack) a sprite "handle" which is actually the address of the defined sprite in the sprite-table.



Vectors forming a sprite (in this case, the sprite has a triangular shape). Visible vectors are represented by solid lines and invisible vectors by dashed lines.

Once a sprite is defined, its position vector can be set by calling the `@putsprite` routine with the sprite-handle in A, the y coordinate in B and the x coordinate in C.

The sprite is not visible on the display until the `@showsprite` routine is called (with A containing the sprite-handle). It is also possible to make a sprite invisible by calling the `@hidesprite` routine (with A containing the sprite-handle). Finally, the vectors of a sprite can be individually set visible or invisible by calling the `@masksprite` routine). In that case, the input arguments are the sprite-handle in A and the address of a "mask" in B. This 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.

A sprite can be deleted from the vector-table and from the sprite-table by calling the `@delsprite` routine (with A containing the sprite-handle) that will indeed erase the sprite from the display. The entry of the deleted sprite in the table becomes free for a new sprite to be defined.

4 Math routines

The VectorUGo-2 OS features the following set of math routines. Note that for trigonometric functions, the input angle is expressed in "binary degree" (bd), with 1 bd = $\pi/128$ rad. The input angle must be between 0 and 255.

Routine	Description
`@umult`	Unsigned multiplication of two integers (A and B). A and B must be between 0 and 255. The result is stored in A.
`@mult`	Signed multiplication of two integers (A and B). A and B must be between -128 and +128. The result is stored in A.
`@div`	Euclidian division of two unsigned integers. $(A, B, ...) \leftarrow (A/B, A \bmod B, ...)$.
`@quot`	Return only $A \leftarrow (A/B, ...)$.
`@modulo`	Return only $A \leftarrow (A \bmod B, ..)$.
`@sin`	$A \leftarrow 256 * \sin(A)$. Result is 16-bit signed, but $\sin(A) = 1 \Rightarrow A \leftarrow 1$ and $\sin(A) = -1 \Rightarrow A \leftarrow -1$.
`@cos`	$A \leftarrow 256 * \cos(A)$. Result is 16-bit signed, but $\cos(A) = 1 \Rightarrow A \leftarrow 1$ and $\cos(A) = -1 \Rightarrow A \leftarrow -1$.
`@rsin`	$A \leftarrow B * sin(A)$. Result is 16-bit signed and B must be between -128 and +127.
`@rcos`	$A \leftarrow B * cos(A)$. Result is 16-bit signed and B must be between -128 and +127.
`@rotscal`	Rotate and scale (see below) a vector. As input argument, A contains the address of the vector (with x in the high-byte and y in the low-byte of the addressed word), the internal variable `v1` (=`x0803`) contains the rotation angle (in bd) and the internal variable v11 (=`x080D`) contains the scaling factor multiplied by 16. After routine call, the internal variable `v4` (=`0x806`) contains x (signed 16-bit) of the transformation and the internal variable `v5` (=`x0807`) contains y (signed 16-bit) of the transformation.

The rotation and scaling of a vector $\vec{V} = (x, y)$ around its origin by an angle θ and a scaling factor ρ is defined by:

$$\vec{R}_{\theta,\rho}(\vec{V}) = \begin{pmatrix} \rho x \cos\theta - \rho y \sin\theta \\ \rho x \sin\theta + \rho y \cos\theta \end{pmatrix}$$

5 Sprite rotation and scaling

It is possible to scale a sprite (to make it appearing larger or smaller) and to rotate a sprite by using the `@rotscalsprite` routine. The input of this routine is the address (stored in A) of the sprite in

the sprite-table, the rotation angle expressed in bd (stored in B) and the scaling factor multiplied by 16 (stored in C). Considering a sprite has been defined by a sequence of vectors $(\vec{V_0}, \vec{V_1}, ..., \vec{V_n})$, where $\vec{V_0}$ is the radial vector, the following algorithm calculates the vectors $(\vec{V_0'}, \vec{V_1'}, ..., \vec{V_n'})$ that are copied into the vector-table for display:

$$
\begin{array}{l}
\vec{S} \leftarrow \vec{V_0} \\
\vec{T} \leftarrow \vec{R}_{\theta,\rho}(\vec{V_0}) \\
\text{For } i = 1 \text{ to } n, \text{do :} \\
\quad \vec{S} \leftarrow \vec{S} + \vec{V_i} \\
\quad \vec{V_i'} \leftarrow \vec{R}_{\theta,\rho}(\vec{S}) - \vec{T} \\
\quad \vec{T} \leftarrow \vec{R}_{\theta,\rho}(\vec{S})
\end{array}
$$

Note that if the shape of the sprite is a polygon, it is possible that after a rotation it does not appear closed on the display, due to small calculation errors. It is possible to correct it by calling the `@closesprite` routine (with A containing the address of the sprite in the sprite-table). This routine will recalculate $\vec{V_n'}$ to be equal to: $-\sum_{i=1}^{n-1} \vec{V_i'}$.

6 Sprite collision detection

Collision of two sprites consists in verifying, for every pair of body vectors (each one taken from a different sprite), if these two vectors are crossing each other when drawn on the display device. Here, the vectors are considered as "displacement vectors" (applied to an origin point), with the origin of each vector being defined by the sum of the position vector, the radial vector and the previous body vectors in the sequence that draws the sprite.

To test if a pair $(\vec{v}, \vec{w})$ of displacement vectors is a pair of vectors crossing each other, the following algorithm is used:

1. Define $\vec{a}$ as the vector joining the end of $\vec{v}$ to the origin of $\vec{w}$.
2. Define $\vec{b}$ as the vector joining the end of $\vec{v}$ to the end of $\vec{w}$.
3. Define $\vec{c}$ as the vector joining the end of $\vec{w}$ to the origin of $\vec{v}$.
4. Define $\vec{d}$ as the vector joining the end of $\vec{w}$ to the end of $\vec{v}$ ($\vec{d} = -\vec{b}$).
5. $\vec{v}$ and $\vec{w}$ are colliding (crossing each other) if the sign of the vector product $\vec{v} \times \vec{a}$ is different from from the sign of the vector product $\vec{v} \times \vec{b}$ AND the sign of the vector product $\vec{w} \times \vec{c}$ is different from from the sign of the vector product $\vec{w} \times \vec{d}$. The sign of a vector product is basically the "turn direction" that has to be made when "following" the first vector in the product then the second.

This algorithm is implemented by the `sprtcoll` routine. The input arguments are the addressed in the sprite-table of the two sprites to be tested, stored in A (sprite 1) and B (sprite 2). The routine returns 0 in A if no collision has been detected. Otherwise, the indices (starting at 1 and counting in the drawing order) of the colliding body vectors of the sprites are stored in A (sprite 1) and B (sprite 2).

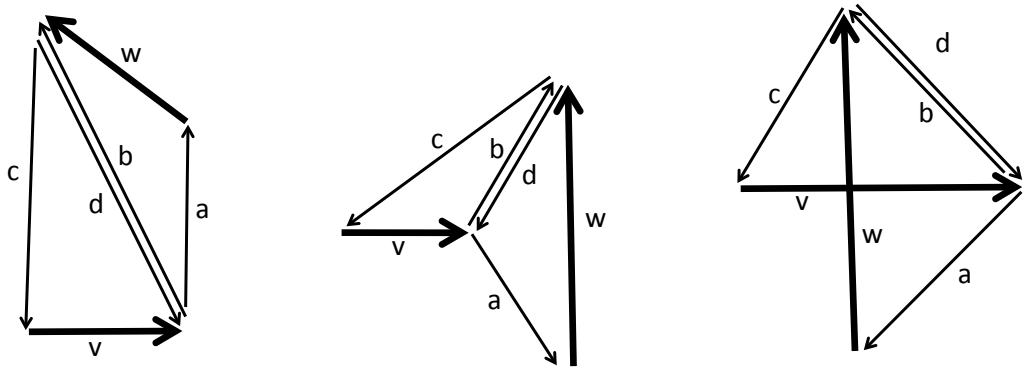
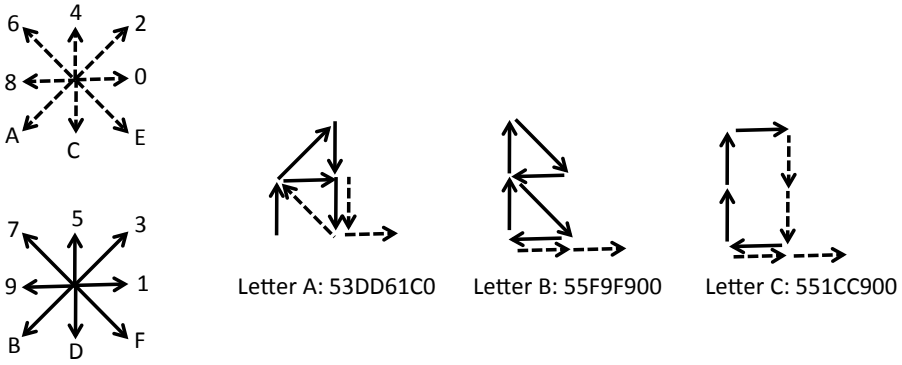


Illustration of the vector collision detection method. Left case: $\mathrm{sgn}(\vec{v} \times \vec{a}) = \mathrm{sgn}(\vec{v} \times \vec{b})$ *and* $\mathrm{sgn}(\vec{w} \times \vec{c}) = \mathrm{sgn}(\vec{w} \times \vec{d})$*, therefore* $\vec{v}$ *and* $\vec{w}$ *are not colliding. Center case:* $\mathrm{sgn}(\vec{v} \times \vec{a}) \neq \mathrm{sgn}(\vec{v} \times \vec{b})$ *but* $\mathrm{sgn}(\vec{w} \times \vec{c}) = \mathrm{sgn}(\vec{w} \times \vec{d})$*, therefore* $\vec{v}$ *and* $\vec{w}$ *are not colliding. Right case:* $\mathrm{sgn}(\vec{v} \times \vec{a}) \neq \mathrm{sgn}(\vec{v} \times \vec{b})$ *and* $\mathrm{sgn}(\vec{w} \times \vec{c}) \neq \mathrm{sgn}(\vec{w} \times \vec{d})$*, therefore* $\vec{v}$ *and* $\vec{w}$ *are colliding.*

7 Text display

Vector-display systems are usually not suitable for displaying long lines of text. However, the ability to display short text information (like a score or a number of lives) is required in a game. The VectorUGo-2 provides the possibility to display "text-items" that are a sequence of characters displayed by default horizontally from the left to the right. Text-items can be placed anywhere on the display screen by setting the system variables located at addresses `x081A` (for x) and `x081B` (for y). Currently, only digits and uppercase letters can be the characters in a text-item. Each character consists of a sequence 8 vectors (visible or invisible). Each vector is picked from a set of 8 vectors pointing in 8 different directions (right, up-right, up, up-left, left, down-left, down, down-right). With 2 vector states possible (either visible or invisible), each graphic representation of a character is coded by a sequence of eight 4-bit nibbles and is therefore coded on two 16-bit words. The last vector is always "right, invisible" and is displayed shorter than the other to mark the separation between this character and the next one in the text-item. These rules give a "runic" look to the character font of the VectorUGo-2 OS.



How characters have been designed from a set of 16 vectors (8 visible and 8 invisible, shown at the left) to form 32-bit (two 16-bit) words that are used by the OS to draw characters.

A text-item is similar to a sprite, except that its graphic definition (sequence of vectors) is not given by the user but automatically generated by the OS. A text-item is created and display by

the `@dispstring` routine. The input argument of this routine is the address (stored in A) of the character string. This string is a sequence of 8-bit ASCII values, stored at the given address with the first character in the high-byte (each 16-bit word of the RAM stores two consecutive characters). The character string must end with the null (=`x00`) character, placed either in the high-byte or the low-byte of the last 16-bit word. After calling `@dispstring`, A contains the address of the text-item in the sprite-table.

It is possible to transform a text-item into a full sprite that can be rotated and re-scaled. A memory block, large enough to store the vectors must be allocated. The number of 16-bit words of this block has to be at least 8 times the number of characters + 1 (for the radial vector). The routine `copysprite` has then to be called with, as input arguments, A containing the address of the text-item in the sprite-table and B containing the address of the memory block. This routine copies the text-item vectors from the vector-table to the memory block, and the data in the memory block can be used as input data of the rotation and scaling algorithm to update the displayed vectors. The entry of the text-item in the sprite-table is updated by `copysprite` and `rotscalsprite` can be used, as for a "normal" sprite.

Any 16-bit unsigned integer (which can be the value of a variable) can be displayed (in decimal with padding zeros) by first creating a text-item of 5 characters (for example "00000"). Then, the `@dispint` routine shall be called with, as input arguments, A containing the text-item address in the sprite-table and B containing the value to be displayed.

8 Joystick

The routine `@joystick` queries the state of the joystick (connected to the Atari joystick port of VectorUGo-2) and returns this state in A. The following bits of A (0=LSB) indicate an action on the joystick when set to 0:

Bit	**Action**
0	Fire
1	Up
2	Down
3	Left
4	Right

9 Organization of the RAM

The following table describes the part of the RAM of VectorUGo-2 that is used by the OS. Some of the addresses in the RAM are not hard-coded and instead a label has been created in the source code.

Address	**Content**
`x0000`	Vector table
`x0803`	Internal variables (`v1` to `v20`)
`x0817`	System variables
`x0820`	Sprite table
`x08E0`	Character vector set (to display text)
`@pow10`	Powers of 10 (used to display number in decimal)
`@sintab`	Trigonometric table (used by `cos` and `sin`)
`x0A00`	User RAM
`x0FFF`	End of RAM

The organization of the vector-table is the following:

Address	**High-byte**	**Low-byte**
x0000	x DAC offset	y DAC offset
x0001	x zero-point	y zero-point
x0002	Number of vectors	
x0003	x vector 1	y vector 1
x0004	LFZ vector 1	-
x0005	x vector 2	y vector 2
x0006	LFZ vector 2	-
...	...	...

The organization of the sprite-table (see Sect. 3 for details) is the following:

Address	**Content (on 16 bits)**
x0820	Number of body vectors +1 of sprite 1.
x0821	Address of first vector (position vector) of sprite 1, as displayed (= address in the vector-table).
x0822	Address of first vector (radial vector) of sprite 1, as defined by the user.
x0823	Number of body vectors +1 of sprite 2.
x0824	Address of first vector (position vector) of sprite 2, as displayed (= address in the vector-table).
x0825	Address of first vector (radial vector) of sprite 2, as defined by the user.
...	...

Hence, the handle of the first defined sprite is x0820, the handle of the second defined sprite is x0821, etc... Up to 64 sprites can be defined.

___oOo___