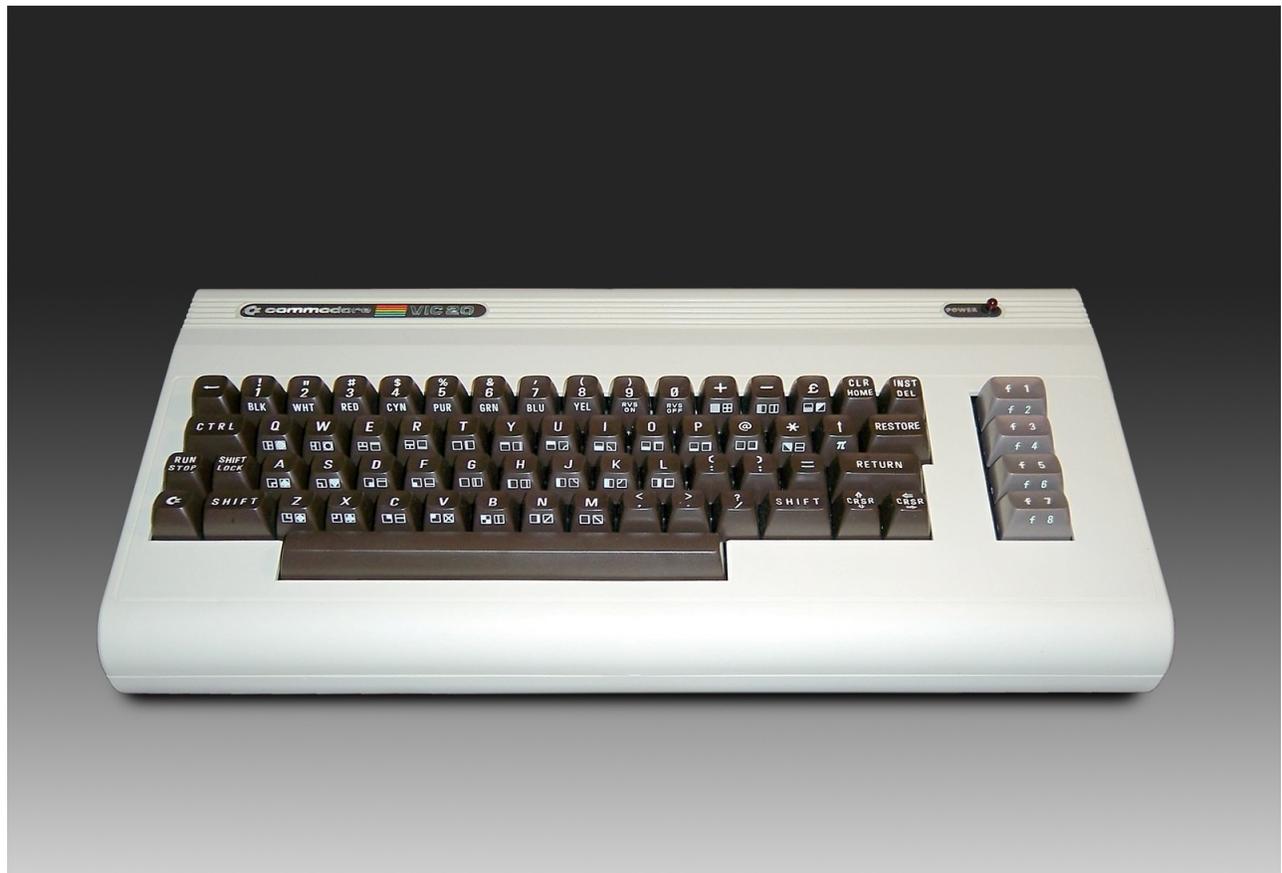


VIC SOFTWARE SPRITE STACK

MMX Edition

for COMMODORE VIC 20

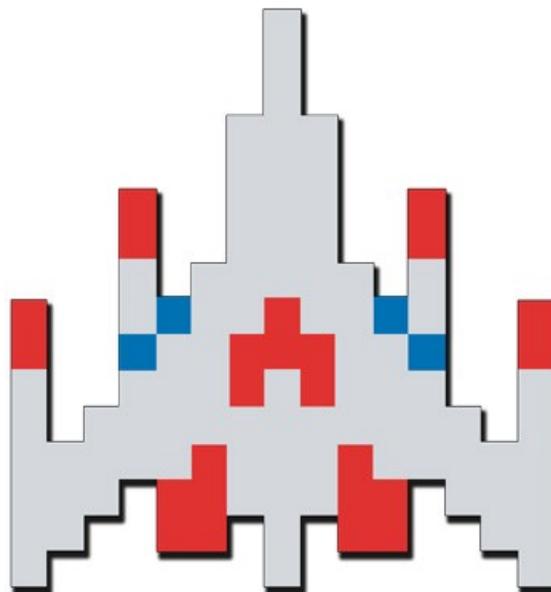


written by Robert Hurst

Rev. 20-Nov-2010

Table of Contents

Foreword.....	2	SSSPEEKXY.....	14
Objective.....	2	SSSPOKE.....	14
Disclaimer.....	2	SSSPRINT.....	14
Theory of Operation.....	3	SSSPRINTS.....	14
Prerequisites.....	4	SSSREAD.....	14
Primer.....	4	SSSWRITE.....	15
Compiling.....	5	Other Considerations.....	16
Linking.....	5	Fixed binaries.....	16
Sprite definitions.....	6	VIC screen geometry.....	16
Sprite pixel coordinate system.....	8	Optimization.....	16
API.....	9	Sprite Ordering.....	16
SSSINIT.....	9	VIC Memory Map.....	17
SSSIRQ.....	10	VIC zero page.....	17
SSSCLEAR.....	11	VIC addresses.....	17
SSSCREATE.....	11	VIC registers.....	17
SSSUSE.....	11	Software Sprite Stack Memory Map.....	18
SSSANIM.....	11	SSS zero page.....	18
SSSMOVEXY.....	11	SSS addresses.....	18
SSSTOUCH.....	12	SSS registers.....	18
SSSREFRESH.....	12	VIC-SSS Run-Time Memory Map.....	20
SSSFLIP / SSSFFLIP.....	12	Run-Time heap.....	20
SSSCELL.....	12	Run-Time free.....	20
SSSPLOT.....	13	Run-Time program code.....	20
SSSPLOTS.....	13	Resources.....	21
SSSPEEK.....	13	Author.....	21
SSSPEEKs.....	13	Online.....	21



Foreword

I have found that programming for an 8-bit computer is a very challenging, yet personally rewarding, experience. When showcasing your effort, you may hear from users (and even other programmers) phrases that start with, “why can't it do ...” and “if only it could ...”. When confronted by those demands, try to keep in mind that it is exactly those attitudes and expectations from the humble beginnings of this “Wonder Computer of the 1980s” that pushed and advanced video game programming into what it has become today.

So given the advancement in computing power, storage, and graphic processing, why even bother writing for an 8-bit machine today? The truth behind that answer lies only within its author. Discover, solve, and enjoy!

Objective



Nothing can replace a tightly-written machine language routine to implement a specific graphic animation technique. But all techniques involve a common set of requirements, and thus generalizing it into a solid working set of subroutines can be constructed well enough to free the game programmer of those bit-twiddling tasks. As a result of this generalization, some trade-offs may be accounted for its slightly larger memory footprint to balance between programmer-friendliness and performance.

Thus, VIC-SSS provides a programmer-friendly API to manage your game's playfield with software-rendered sprites and other animations for a flicker-free video experience. On-the-fly custom character manipulations with dual video buffers accomplish these goals, avoiding the alternative of dedicating all internal RAM for a smaller, but fully, bit-mapped screen. This API supports both NTSC and PAL VIC 20 computers, and allows for display modes that change VIC's 22x23 screen layout.



Disclaimer

The software sprite stack promotes a flicker-free video experience, with the option by the game programmer to govern frame buffer flips with screen raster timing. While the VIC 20 computer and its graphics are primitive to begin with, this API was created to strike a balance between machine and *programmer friendliness* – which is what the VIC is all about. The result of that friendliness makes the code size around 2 kilobytes and requires nearly all of the internal 4 kilobytes of RAM for graphics display and management. Thus, your game program will require some form of memory expansion – all examples provided will run on 8k expansion.

But if you find VIC to be too restricting for your creativity, it might be better to implement your game ideas on a more advanced 8-bit platform, such as Commodore 64's VIC-II – which was designed specifically for home video games. Or, use a modern API for cross-platform development on PCs, such as that found in the SDL project.

The point being is in order to have any success, keep your project fun and enjoy the challenge of making it happen within the scope and humble workings of the Commodore VIC 20 home computer... and you will gain better appreciation of what 1980s arcade gaming was all about.



Theory of Operation



First, welcome to the fascinating world of VIC graphics and animation. Before you can get creative and implement VIC-SSS within a game, you must understand how all this stuff works, and what you can expect from it.

VIC Software Sprite Stack is an application program interface, specifically to manage the play field of a video game or some graphic animation need (like a mouse pointer). The play field (see **PLAYFIELD** variable) is where the base display information lives, such as alpha-numeric characters, VIC graphic characters, and any custom characters you may define. The play field also has its own corresponding color space (see **PLAYCOLOR** variable) for each character cell. Reading and writing to this play field is done by simple subroutines, equivalent to BASIC **PEEK** and **POKE** statements. For both performance and flicker-free purposes, writes to the play field are tracked by an internal system of dirty bits, which queues up all pending changes into one batch operation later that commits them to a video frame buffer (see **PENDING** variable). This video buffer allows changes to character and color address space that will become visible to the player when the appropriate VIC control register is re-directed to use it.



 In addition to the play field are the software sprites. These objects have a structure that defines their image, size, color, and position to place them on top of the play field. Unlike regular VIC characters, a software sprite can be defined to “float” across character cells. For that to occur, an extended sprite pixel coordinate system is in place to target that finer degree of resolution – as opposed to the simpler cursor plots to a 23-row by 22-column character display. The remaining focus of this document will detail that use of managing software sprites.

After setting up your play field and updating sprite registers, your program must inform VIC-SSS to have all that new information activated. This is managed by a screen flip operation (see **SSSFLIP**), which makes the prior changes visible on the VIC display, by a series of write-commit operations to the frame buffers. And all this prep work is for just one screen update! After each flip operation, your program loop will continue by building out the next frame, make a video frame flip, and so on, and so on.



This MMX (2010) edition not only allows for sprites to float across character cells on the play field, but they can now also move behind them. For that to occur, this API adds another kind of write to the play field that sets a flag in that cell to make it *static* – these static cells forces the sprite rendering to not overwrite them. Additionally, these static cells can have a different character code, and even color, per frame buffer. The result of that allows for animations to occur simply from a screen flip.

 Having a lot of animated software sprites floating around can really eat up the microprocessor and slow the game's action down to an unbearable crawl. While a lot of effort has been put into speed optimizations, there is no escaping those mathematics. To assist the game programmer in a friendly way, **SSSFLIP** has an extended fast version, which detects if too many screen refreshes are occurring between game loops – thus allowing for an automatic “frame skip” to make up for the slowed action.

Lastly, a feature for allowing *repeating* sprites has been added. If your game allows for multiple sprite images that can be rendered only once, then this bit-flag is for you. **SPRITE INVADERS** is a fine example of this new feature.



Prerequisites



Access to and operational knowledge of a real VIC 20, or substituted with the use of machine emulation, such as that provided nicely by VICE Team or MESS project.

You should already know how to write and debug in 6502 assembly language, or even better, native machine code. It is even quite possible to invoke this API using BASIC SYS statements, although the overall speed will be hampered by your BASIC program's main loop.

This API was written in assembly language, with its source code and configuration files made specifically for use with the CC65 assembler and linker utilities. If a different assembly process is sought, these source files can be modified to meet your choice of tools.

You should also possess the fundamentals of VIC graphics – its control registers and what it means to make custom graphic characters. There is already plenty of written material, tutorials, and software tools with examples to explain all of that. And, there is still a modest user community following to assist in any particular detail that may escape you. Like the friendly computer VIC 20, its users are just as friendly!

Please see the resources section for online links.

Primer

To assist in getting you started more quickly, there is a **primer** folder with assembler sources and a linker configuration file. There is a **README.TXT** file in there with quick instructions on how to get started. The folder comes with a copy of Windows 32-bit binaries of the cc65 project's assembler, **ca65.exe**, and linker, **ld65.exe**, as well as the pertinent documentation for those two tools. There is also a Windows batch file and Linux script to compile and link the assembler source files into a VIC binary, which can be loaded as any ordinary BASIC program. You only need copy this folder to get started.



The primer folder was designed to get the VIC game programmer started quickly in a successful direction. You may decide later in your project that more memory is required. Or you want the game to reside in the VIC's ROM cartridge slot (\$A000). Or you want to use multi-loading and place the SPRITE routine and data into the 3k address space to free up more contiguous space for your game. Whatever your requirements are, it is achievable by making changes to the segment directives in your assembler source files as well as your project's linker configuration file.

Complete details of the assembler and linker are included in the **docs** folder.

Compiling

Since VIC-SSS is composed of tiny source, header, and object files, it is acceptable to simply copy them into your game's project folder. If you are a purist and want to reference VIC-SSS as a single copy, there are equivalent command-line options to include that path as well. I have found the pre-compiled Microsoft Windows binaries to work just fine using WINE under Linux, so these instructions work for both host environments:

```
$ ca65.exe --cpu 6502 --listing --include-dir . yourgame.s
```

When ca65 executes, it will find this required compiler directive (entered as part of yourgame.s source file), which instructs it to use the header file, **VIC-SSS-MMX.h**, inside the current directory:

```
.include "VIC-SSS-MMX.h"
```

If there are no errors, the assembler will produce both an object file (yourgame.o) and a source listing file (yourgame.lst).

Please refer to CC65 documentation for complete details on its assembler tool, ca65.



Linking

After running the assembler against your program's source code, it is time to link its object file with the VIC software sprite stack: **VIC-SSS-MMX.o**. There is a pre-made linker configuration file to use, with the beauty of using an assembler is that you are not limited to using just one. You are free to make your own startup and/or linker configuration, particularly if your program requires a different memory address layout, i.e., +3k, +16k, or +24k memory expansions.

There are sample assembler source files to make a VIC program file, which is loaded after starting a VIC 20 (with at least 8k memory expansion) in BASIC mode. Its **BOOT** and **STARTUP** segments are compiled in **basic.o**, and must be used with the corresponding linker configuration file, **basic-8k.cfg**.

```
$ ld65.exe -C basic+8k.cfg -Ln yourgame.sym -m yourgame.map \
  -o yourgame.prg basic.o yourgame.o VIC-SSS-MMX.o
$ mess vic20 -ramsize 16k -quik yourgame.prg
$ xvic -memory 8k -autostart yourgame.prg
```

When linking is successful, it not only creates a VIC binary file, but also useful symbol and map files. The symbol file can be loaded inside VICE's monitor to allow for setting program breakpoints, watching load/store to memory addresses, and inspection of values using the same global symbols from the assembler source. This makes debugging much easier to do.

The contents of this map file details many useful things, among which is the consumed address space that each segment uses. From the demos folder, here is **bigdude.map**:

Segment list:

Name	Start	End	Size
BOOT	0011FF	00120C	00000E
SSSBUF	001800	001B1F	000320
STARTUP	002000	002031	000032
CODE	002032	002166	000135
SPRITE	002167	002912	0007AC
RODATA	002913	00297E	00006C

With an 8k memory expander, we can see that there is still free space available from \$297F - \$3FFF. Free bytes for more program code, more graphics – oh joy!! What is not listed here is the **MYCHAR** segment, because no other custom characters were provided for this demo. If a fixed set of custom characters are desired, simply add the segment data in the section marked in **basic.s**, and it will be included in the binary.

Please refer to CC65 documentation for complete details on its linker tool, ld65.



Sprite definitions

The **SPRITES** register controls how many sprites are active. It gets incremented whenever you call **SSSCREATE**. The maximum number (in theory) is 64. It is likely not to exceed 16 discrete sprites, but repeating sprites allow for many more without the performance hit. You should modify the **VIC-SSS-MMX.h** file to reflect your game's need in **SPRITEMAX**.

While the included **basic.s** file makes use of **SPRITEMAX** to reserve the appropriate amount of bytes for its registers, you are also free to modify the bytes used to maintain each sprite image, before it gets merged into the custom character space. If your game requires less than the maximum $64 * 2$ (128) custom characters to display all its active sprites, go to the segment **SSSBUF** and shrink the reserved character count.

SPRITEIMGL/H contains the address pointer to each sprite's source bit-mapped image.

SPRITEBUFL/H contains the address pointer to each sprite's image, shifted X/Y as requested within a character cell matrix.

SPRITEC1L/H and **SPRITEC2L/H** contains the address pointer to each sprite's custom character cell matrix, per video buffer.

SPRITEBACK contains the character code each sprite has collided with, as long as collision detection is enabled AND the sprite overlaps any part of its background.

SPRITECX/CY are the sprite coordinates for each sprite that has collision detection enabled AND the sprite overlaps any part of its background. See also **SPRITEDEF** and **SPRITEZ**.

SPRITECOL contains the 4-bit VIC color code:

- bit 3 (\$08) claims whether this sprite is either 0 = hi-res color, or 1 = multi-color mode
- bits 0-2 (\$00 - \$07) is this sprite's character color

SPRITEDEF contains a bit field that defines the following characteristics for that sprite:

- bit 7 (\$80) enables or disables this sprite from rendering and displaying.



- bit 6 (\$40) enables or disables collision detection. See also **SPRITEZ**.
- bit 5 (\$20) enables (**XOR**) or disables (**OR**) sprite image operator with any background pixels. See also **SPRITEDEF5**.
- bit 4 (\$10) determines if this is a repeating sprite (another copy of the previous sprite rendered, but with its own X/Y and color attributes), or an independent sprite.
- bit 3 (\$08) claims whether this sprite is either 0 = horizontally fixed, or 1 = floats horizontally
- bit 2 (\$04) claims whether this sprite is either 0 = vertically fixed, or 1 = floats vertically
- bit 1 (\$02) claims whether this sprite is either 0 = 8-pixels wide, or 1 = 16-pixels wide
- bit 0 (\$01) claims whether this sprite is either 0 = 8-pixels high, or 1 = 16-pixels high

SPRITEH contains the value of the number of pixel rows used by this sprite image. It must contain a value >0 and <= the sprite's height (8 or 16) as defined in **SPRITEDEF**.

SPRITEX and **SPRITEY** contain their respective ordinate position relative to the sprite pixel coordinate system. This is the only register you might want to reference within your game. Use **SSSMOVEXY** to change their values.

SPRITEZ contains a bit field used by the sprite rendering process in **SSUPDATE**. Of particular interest is bit 3, which will get set if **SPRITEDEF** bit 6 is enabled AND the sprite image overlaps any part of its background. See also **SPRITEBACK**, **SPRITECX**, and **SPRITECY**.

NOTE: The index register, **SSNUM**, is maintained to point to the current set of sprite registers.



Sprite pixel coordinate system

A sprite is placed on a virtual playing field that is 32-pixels higher and wider than the VIC's current screen resolution. This extra 16-pixels above the top, below the bottom, and outside the left and right borders allow for a sprite to “enter” or “exit” the visible playing field, with appropriate image clipping. Thus, 16 (\$10) is the first visible pixel for either ordinate, and any value less than 16 would “clip” the sprite by the VIC screen's border. And there are two computed symbols, **SSCLIPX** and **SSCLIPY**, which contain the ordinate value of the first pixel that lies outside the opposite borders.

It is ultimately up to the programmer to manage these boundaries within the game, because the software sprite rendering process will not update the display for any sprite that has a value that would place it completely outside the virtual playing field. Below is a sample graph for your typical 23-row and 22-column VIC display. Sprite pixel coordinates are depicted as (X,Y) values:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22		
	(0,0)	(8,0)																							(192,0)
	(0,8)	(8,8)	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160	168	176	184	(192,8)
1		16																							SSCLIPX
2		24																							
3		32																							
4		40																							
5		48																							
6		56																							
7		64																							
8		72																							
9		80																							
10		88																							
11		96																							
12		104																							
13		112																							
14		120																							
15		128																							
16		136																							
17		144																							
18		152																							
19		160																							
20		168																							
21		176																							
22		184																							
23		192																							
	(0,200)	(8,200)	SSCLIPY																				(192,200)		



API

SSSINIT



This should be the first subroutine called to initialize both VIC and SSS registers for managing the **PLAYFIELD/PLAYCOLOR**, **VICFRAME1/VICCOLOR1**, and **VICFRAME2/VICCOLOR2** frame buffers. The VIC control registers are read to setup **PLAYROWS** and **PLAYCOLS**, which in turn compute the borders for sprite image clipping in **SSSCLIPX** and **SSSCLIPY**. The VIC control registers are modified to point to an **ACTIVE** frame buffer, with **PENDING** set to point to the other frame buffer on the next flip.

SSSNUL is the character code to fill the video frame buffers; a typical value would be a SPACE (\$A0), which points to VIC ROM character set. This value also has rendering optimization implications if a sprite occupies an “empty” background.

The VIC 4-bit value stored in **COLORCODE** is used to fill the frame buffers default color.

Use the following table to determine your program requirements. Multiply each cell requirement by the number of sprites that could potentially be active on-screen at the same time:

<u>SPRITEDEF</u>	<u>Floats</u> <u>X / Y</u>	<u>X x Y</u> <u>W x H</u>	<u>Row x Col</u> <u>Matrix</u>	<u>Cells</u>	
%0000	no / no	8 x 8	1 x 1	1	← great for missiles
%0001	no / no	8 x 16	2 x 1	2	
%0010	no / no	16 x 8	1 x 2	2	
%0011	no / no	16 x 16	2 x 2	4	
%0100	no / yes	8 x 8	2 x 1	2	
%0101	no / yes	8 x 16	3 x 1	3	← up/down tall sprite
%0110	no / yes	16 x 8	2 x 2	4	
%0111	no / yes	16 x 16	3 x 2	6	
%1000	yes / no	8 x 8	1 x 2	2	
%1001	yes / no	8 x 16	2 x 2	4	
%1010	yes / no	16 x 8	1 x 3	3	← left/right wide sprite
%1011	yes / no	16 x 16	2 x 3	6	
%1100	yes / yes	8 x 8	2 x 2	4	← most “common” format
%1101	yes / yes	8 x 16	3 x 2	6	
%1110	yes / yes	16 x 8	2 x 3	6	
%1111	yes / yes	16 x 16	3 x 3	9	← big dude

While **H** in the table represents the maximum sprite height, the actual *required* height value is governed by **SPRITEH**.

Note that the character cell matrix is organized by columns, then by rows, i.e., an 8x8 sprite image allowed to float in both **X** and **Y** directions is rendered in a 2 x 2 character cell matrix and might be displayed as (only for example):

```
@B
AC
```



Note that you can define a set of VIC custom characters in conjunction with sprite allocations. Start their definition using “@”, “A”, “B”, “C”, ... as many as your game requires, but as long as those custom characters are BEFORE the active sprite pool. The sprite pool is allocated starting from the TOP of that VIC custom character address space within \$1C00 – 1FFF, and works its way backward for each new sprite creation. In addition to these 128 custom characters, all 128 VIC ROM characters are available by using their reverse attribute (+\$80).

SSSIRQ

Do not call this routine directly and this IRQ handler must not be invoked before **SSSINIT**, because unpredictable results can occur from uninitialized variables. This routine may be setup to directly intercept the IRQ software event, such as:

```
SEI
LDX #<SSSIRQ
LDY #>SSSIRQ
STX $0314
STY $0315
CLI
```

If you have your own custom IRQ handler, simply put a **JMP SSSIRQ** at the end of your handler as **SSSIRQ** does the standard **JMP \$EABF** at its conclusion.

Only with **SSSIRQ** in place can the **SSSFLIP** routine be instructed to wait for the VIC to complete a screen refresh cycle (or more) – this is when it modifies its control registers to swap out the **ACTIVE** frame buffer with the **PENDING** one. Not only does double-buffering allow for flicker-free animation, but also timing it with the vertical sync allows for tear-free graphics. 

The trade-off for this pleasant video effect is that your program's main loop will be governed by this vertical sync. But this has the positive effect of maintaining a consistent pace, especially if a varying mix of action occurs within the game (multiple volleys, multiple explosions, large enemy ships, etc.) So, if the program and software sprite rendering takes more than 1/60th (NTSC) or 1/50th (PAL) of a second to complete an iteration, the video frame rate (or Frames-Per-Second) will reduce and game play will wait to synchronize with the next hardware video refresh:

<u>NTSC</u>	<u>FPS</u>	<u>PAL</u>	<u>FPS</u>
1	60	1	50
2	30	2	25
3	20	3	16.7
4	15	4	12.5
5	12	5	10
6	10	6	8.2



Invoking this feature is always at the option by the programmer. For example, a special animation scene may want to implement vertical sync pacing, but the main game loop requires a faster arcade-style of play. For arcade-style games, you might find this feature to be an easy way to implement two skill level options, such as a “teddy bear” level that enables vertical sync timing (slower), or a “normal” level that runs as fast as the machine/program allows.

If the vertical sync option is to be used anywhere within the game, it is first required to synchronize VIC's IRQ software timer with its vertical refresh (NTSC or PAL). This initial synchronization process is provided in the **STARTUP** segment within **basic.s**. The startup routine inspects the machine's kernal for the correct video timing, NTSC or PAL, that the VIC was made for.

SSSCLEAR

Clear the **PLAYFIELD** and **PLAYCOLOR** frame buffer.

Pass **Accumulator** with the character code to fill the video frame buffers; a typical value would be a SPACE (\$A0), which points to the ROM character set. The value stored in **COLORCODE** will be used to fill the color space.

SSSCREATE

Create a new sprite on the stack.

Pass **Accumulator** with the coded **SPRITEDEF** value. Even though **SPRITEDEF** contains the sprite's maximum height of 8 or 16, you must pass **Y** with the sprite's image height.

When successful, the **X** index register is returned with the sprite number in **SSSNUM**.

SSSUSE

Make this sprite's registers the current working set.

Pass **X** index register with the sprite number you want to manipulate.

A few working sprite registers are initialized, notably **SSSNUM**, which returns back in the **X** index register.

SSSANIM

Load the address pointer to a sprite image into the current sprite, **SSSNUM**.

Pass **X,Y** as the source to the sprite image.

Pass **Accumulator** with the sprite's color code to use.



SSSMOVEXY

Move the current sprite, **SSSNUM**, to these absolute coordinates.

Pass **X,Y** with new sprite pixel coordinates to use.

SSSTOUCH

This routine touches a sprite to force it to re-render on the next flip operation. It gets called as part of **SSSANIM** and **SSSMOVEXY**.

SSSREFRESH

This routine touches all sprites to force them to re-render on the next flip operation. While this is convenient in circumstances when there is a lot of direct manipulation of sprite registers, it bypasses the efficiency programmed into **SSUPDATE** and may degrade performance.

SSSFLIP / SSSFFLIP

Make the **PENDING** frame buffer **ACTIVE**.

Pass **Y** index register with the number of screen refreshes to wait up to. A typical value of zero means “no wait” for best performance, but that also means some visible tearing may be observed.

A value of 1 means to wait for VIC's next screen refresh (or an infinite loop if **SSSIRQ** is not enabled). This will eliminate visible tearing, as the routine will wait until VIC is done drawing the current frame. However, if this value is too low, you may observe varying speeds with your graphic animations. If that occurs, it is the result of your game loop – plus any software sprite updates – taking too long to consistently make it in time for the next frame redraw. If eliminating video-tearing is desired, then keep increasing this value until the animations can run within tolerable range of the video refresh timing.

Your game may have a variable amount of active sprites visible on the screen, which can lead to wider variances occurring during screen update timings. Those periods of slow performance can really detract from the arcade-style gaming experience.

To help keep the pace of the game at a higher rate, there is an extension to this video flip call that can be used: **SSSFFLIP** (fast flip). What this does is determine if the time for the last video flip took longer (+2 frames) than the passed **Y** value. So if too many screen refreshes passed before the frame buffers were swapped, then the next **SSSFFLIP** call is immediately returned – skipping all of the frame buffer updates for that iteration. This essentially makes your game loop execute TWICE per screen update, resulting in a frame “skip”, until the game's animation and action allows for rendering to occur within the threshold desired.



SSSCCELL

Write to a protected cell on the **PENDING** frame buffer. These cells are specially marked against a software sprite from being rendered on top of them. Any such part of a sprite's image is “clipped” when colliding with these protected cells. See also **SSSPLOTS**.

Pass **X** index register with the column number of the character space, counted from zero and up to, but not including, the value computed in **PLAYCOLS**.

Pass **Y** index register with the row number of the screen line, counted from zero and up to, but not including, the value computed in **PLAYROWS**.

Pass **Accumulator** with the character code to write to the **PENDING** video frame buffer.

The value stored in **COLORCODE** will be used to color that space.

SSSPLOT

Puts the “cursor” to the specified character cell coordinate on the **PLAYFIELD** and **PLAYCOLOR** frame buffer. This updates valuable pointers into the frame buffer for your program to use, and it sets up calls to **SSSPOKE** and **SSSPRINT**.

Pass **X** index register with the column number of the character space, counted from zero and up to, but not including, the value computed in **PLAYCOLS**.

Pass **Y** index register with the row number of the screen line, counted from zero and up to, but not including, the value computed in **PLAYROWS**.

This routine re-uses the same VIC BASIC variables that depicts its screen cursor. This has the potential of providing compatibility and future value as a BASIC wedge or integration with another customized video output technology, such as a 40-column display, etc.

Refer to **SCRNLINE**, **COLORLINE**, **DIRTMAP**, **CRSRROW**, and **CRSRCOL**.

SSSPLOTS

The same functionality as **SSSPLOT**, but puts the cursor on the **PENDING** frame buffer for reading and writing.

This is used for rendering sprites on the display without affecting the underlying **PLAYFIELD** frame buffer. It is also used by **SSSCCELL** for other simple animations, such as:

... placing a colored object on one video frame buffer, but changing its color on the other video buffer. This produces an interesting flickering effect by the screen flips. An example is to use a white & yellow diamond for a glowing effect, or a black & yellow diamond for a dimming effect.

... placing two different objects on each video frame buffer, making for a simple animation effect. An example is to use the plus (+) and asterisk (*) to make only the cross marks (X) flicker. Another is to use VIC's graphic characters, such as the circle and ball to make for a pronounced outline with a flickering colored center.

SSSPEEK

Makes the same call to **SSSPLOT**, but then reads the **PLAYFIELD** contents pointed to by **SCRNLINE** and **COLORLINE** and updates **CRSRCHAR** and **CRSRCOLOR**, with the former returned by the Accumulator.

Pass **X** index register with the column number of the character space, counted from zero and up to, but not including, the value computed in **PLAYCOLS**.

Pass **Y** index register with the row number of the screen line, counted from zero and up to, but not including, the value computed in **PLAYROWS**.



SSSPEEKS

The same call requirements and functionality as **SSSPEEK**, but then reads the cursor position from the **PENDING** frame buffer, instead of **PLAYFIELD**.

SSSPEEKXY

Similar to functionality as **SSSPEEK**, but it uses the software sprite pixel coordinate system instead of the cursor position.

Pass **X** index register with the pixel column number on the sprite playing field, counted from zero and up to, but not including, the value computed in **SSSCLIPX** plus 16.

Pass **Y** index register with the pixel row number of the sprite playing field line, counted from zero and up to, but not including, the value computed in **SSSCLIPY** plus 16.

SSSPOKE

This call writes to the **PLAYFIELD** contents pointed to by **SCRNLINE** and **COLORLINE**. The cell is flagged as dirty to correctly update the **PENDING** frame buffer during the video flip phase.

Pass **Accumulator** with the character code to write to **PLAYFIELD** immediate, and to the video frame buffers deferred.

The value stored in **COLORCODE** will be used to color that space.

SSSPRINT

The same call and functionality as **SSSPOKE**, but the cursor is advanced to the right after writing its cell – with line wrap taken into account. All registers are preserved upon exit, allowing the calling program to safely put this inside a loop.

SSSPRINTS

Prints a string of characters following the **JSR SSSPRINTS** call, until reaching a NULL byte. CBM color (\$F0-\$FF) and carriage control (\$0D) codes are also interpreted as part of the output.

```
JSR SSSINIT
JSR SSSPRINTS
.byte $F2 ; red text
.asciiz "HELLO, WORLD!"
LDY #0 ; render immediate
JSR SSSFLIP
```

SSSREAD

Similar in functionality by calling **SSSPEEKXY** – **SSSPLOTS** – **SSSPEEK**, but this simpler and faster routine returns the character code from the **PENDING** frame buffer that is located within the visible sprite's pixel coordinate system.

Pass **X** index register with the pixel column number on the sprite playing field, counted from 16 and up to, but not including, the value computed in **SSSCLIPX**.

Pass **Y** index register with the pixel row number of the sprite playing field line, counted from 16 and up to, but not including, the value computed in **SSSCLIPY**.

If either **X** or **Y** are not within the visible sprite's pixel coordinate system ($\$10 - \text{SSSCLIPX}/Y$), the return value in the **Accumulator** defaults to **SSSNULL**, which is typically assigned a ROM space character value (\$A0), but you are free to redefine.

This is used by the software sprite rendering process, but may be useful by the main program.

SSSWRITE

This routine writes the character code to the **PENDING** frame buffer that is located within the visible sprite's pixel coordinate system. If **X/Y** are visible, then it completes the write by calling **SSSPLOTS – SSSPOKE**.

Pass **X** index register with the pixel column number on the sprite's visible playing field, counted from 16 and up to, but not including, the value computed in **SSSCLIPX**.

Pass **Y** index register with the pixel row number of the sprite playing field line, counted from 16 and up to, but not including, the value computed in **SSSCLIPY**.

This is used by the software sprite rendering process, but may be useful by the main program.



Other Considerations

Fixed binaries

A **fixed** folder is provided with pre-made binaries that may be useful for your game development. Perhaps you are using BASIC or are already familiar with another 6502 assembler and would rather just invoke SSS functions with **SYS** or **JSR** calls to fixed addresses. Use the accompanying **SYM** file for each SSS function address resolution.

FIXED-0400.PRG is for VIC owners or emulators to make use of that 3K RAM expansion space reserved at \$0400 - \$0FFF. It has the lead 2-byte address loader contained within its binary, so it can be easily loaded into the expansion area with: **LOAD "FIXED-0400.PRG",8,1**

FIXED-3400.BIN is an example of a binary made for inclusion into an assembler project, requiring your project to use a compiler directive such as **.incbin** within it. It can be made into PRG format by placing a **.word \$3400** at the top and re-making the binary.



VIC screen geometry

The default VIC screen geometry is 23 rows by 22 columns (506 character cells). If you want a different sized screen, modify the VIC control registers before calling **SSSINIT**, so that it can initialize its variables, **PLAYCOLS** / **PLAYROWS** and **SSSCLIPX** / **SSSCLIPY**, appropriately. By default, the maximum row size can be 24 unless appropriate adjustments are made to the provided **SSS**, **DIRTYLINE**, and **DIRTYLINE2** memory locations.



No facility for expanding or shrinking the visible area is provided. While it is possible to implement a smaller “window”, these variables are sized to allow sprites to appear anywhere on the screen. Those boundaries are enforced by program control.

Optimization

If the game allows, try to keep your sprites' coordinates aligned with “8”s. Maze games are a great example of when sprites can only move in either an X or Y motion within a fixed width floor area. It is not necessary to change **SPRITEDEF** to force a “non-floating” attribute, as the sprite update routine is smart enough to only display those character cells that actually have a portion of the sprite image rendered within it. So keeping values divisible by 8 on the non-floating ordinate makes for a smaller sprite cell matrix, and thus less manipulation to the sprite image overall.



It is better to avoid arbitrary calls to **SSSANIM** when there is no change in the sprite's image data. The same goes for **SSSMOVEXY** when there is no change in that sprite's X/Y coordinate. Both these calls force the sprite to be rendered fully from the source image with any background image, when it may not be necessary to do any part of those operations.

Sprite Ordering

When organizing your sprites, keep in mind that the order the sprites are created (**SSSCREATE**) is the same order they get processed (**SSSFLIP**), for rendering and displaying with optional collision-detection. Thus, the higher the sprite number, the more priority its image, color, and collision-detection has over any lower-numbered sprites.

VIC Memory Map

The following tables are VIC machine memory locations as used by VIC-SSS. They are maintained to remain compatible with the internal workings of VIC's Kernal and BASIC. The programmer is free to relocate these addresses to suit the needs of their implementation.



VIC zero page

	<u>Address</u>	<u>Symbol</u>	<u>Description</u>
199	\$C7	RVSFLAG	character reverse flag
204	\$CC	CURSOR	cursor enable (0=flash)
209	\$D1	SCRNLINE	2-byte pointer to cursor's screen line
211	\$D3	CRSRCOL	index pointer to cursor's position on screen line (0-21)
214	\$D6	CRSRROW	screen row of cursor's position (0-22)
243	\$F3	COLORLINE	2-byte pointer to cursor's color line

VIC addresses

	<u>Address</u>	<u>Symbol</u>	<u>Description</u>
646	\$0286	COLORCODE	current character color to be used when printing text
647	\$0287	CRSRCOLOR	existing color under the cursor
648	\$0288	SCRNPAGE	active screen memory page (unexpanded = \$1E)
657	\$0291	SHIFTMODE	switch between graphic/lowercase: 0=allow, 128=locked
658	\$0292	SCROLLFLAG	auto-scroll down flag

VIC registers

	<u>Address</u>	<u>Symbol</u>	<u>Description</u>
36866	\$9002	VIC+\$02	bit 7: screen memory +\$0200
36869	\$9005	VIC+\$05	bits 4-7: screen memory \$C = \$1000, \$F = \$1C00; bits 0-3: character table: \$0 = ROM, \$F = \$1C00
37888	\$9400	VICCOLOR1	1 st color frame buffer
38400	\$9600	VICCOLOR2	2 nd color frame buffer

Software Sprite Stack Memory Map

SSS zero page

	<u>Address</u>	<u>Symbol</u>	<u>Description</u>
1	\$01	VECTORBG	2-byte pointer to PLAYFIELD character image (8x8)
217	\$59	DIRTYLINE2	24-byte array containing last dirty cell (+1) in that row
177	\$BF	NEWDIRT	all new character cell updates will add these bits into PLAYCOLOR : 7=VIDEO1; 6=VIDEO2; 5=PLAYFIELD; 4=STATIC
200	\$C8	PLAYROWS	current screen row length (16-24)
213	\$D5	PLAYCOLS	current screen line length (16-24)
217	\$D9	DIRTYLINE	24-byte array containing first dirty cell in that row
241	\$F1	DIRTMAP	2-byte pointer to PLAYCOLOR for dirty-bit updates
247	\$F7	VECTORFG	2-byte pointer to SPRITE bit-mapped image
249	\$F9	VECTOR1	2-byte pointer (sprite target image – 1 st column)
251	\$FB	VECTOR2	2-byte pointer (sprite target image – 2 nd column)
253	\$FD	VECTOR3	2-byte pointer (sprite target image – 3 rd column)

SSS addresses

These memory locations are used throughout the software sprite stack. There are 5 unused but named registers for the game program's discretion. **SSSCLIPX** and **SSSCLIPY** are useful symbols for the game program as a means of soft-coding any screen boundary checks:

	<u>Address</u>	<u>Symbol</u>	<u>Description</u>
645	\$0285	FPS	number of VIC video flip operations every 64-jiffies
659	\$0293	PENDING	next frame to display: VIDEO1 or VIDEO2
660	\$0294	ACTUAL	video page at VIC startup (\$10 or \$1E)
661	\$0295	VSYNC	wait for vertical sync to occur (>1 for multiple syncs)
662	\$0296	VSYNC2	number of skipped frames, to maintain timing
663	\$0297	VCOUNT	current SSSFLIP count (for FPS stat)
664 - 668	\$0298 \$029C	R0 - R4	5 (unused) registers
669	\$029D	SSSCLIPX	pixels to right border: 8 * (PLAYCOLS + 2)
670	\$029E	SSSCLIPY	pixels to bottom border: 8 * (PLAYROWS + 2)

SSS registers

The placement of these registers (each repeats for each sprite up to **SPRITEMAX**) are ultimately up to the programmer; this table is for reference purposes and for a 16-sprite (8x8) game:



	<u>Address</u>	<u>Symbol</u>	<u>Description</u>
6144	\$1800	SSSBUF	rendered sprite image buffer (make)
6656	\$1A00	SPRITEBACK	if collision detection is on, the character code it “touched”
6672	\$1A10	SPRITEBUFH	hi-byte pointer to image copy in sprite buffer
6688	\$1A20	SPRITEBUFL	lo-byte pointer to image copy in sprite buffer
6704	\$1A30	SPRITEC1H	hi-byte pointer to 1 st image copy in custom character space
6720	\$1A40	SPRITEC1L	lo-byte pointer to 1 st image copy in custom character space
6736	\$1A50	SPRITEC2H	hi-byte pointer to 2 nd image copy in custom character space
6752	\$1A60	SPRITEC2L	lo-byte pointer to 2 nd image copy in custom character space
6768	\$1A70	SPRITECOL	4-bit VIC color code
6784	\$1A80	SPRITECX	if collision detection is on, X sprite coordinate of “touching”
6800	\$1A90	SPRITECY	if collision detection is on, Y sprite coordinate of “touching”
6816	\$1AA0	SPRITEDEF	sprite image definitions: - bit 0: height (0 = 8px; 1 = 16px) - bit 1: width (0 = 8px; 1 = 16px) - bit 2: float Y (0=fixed cell, 1=vertical float) - bit 3: float X (0=fixed cell, 1=horizontal float) - bit 4: repeating (0=own, 1=use rendering from previous) - bit 5: ghost (0=merge image; 1=invert image) - bit 6: collision (0=ignore; 1=detect) - bit 7: display (0 = disabled; 1 = enabled)
6832	\$1AB0	SPRITEH	required: sprite height (1-16)
6848	\$1AC0	SPRITEIMGH	hi-byte pointer to your bit-mapped image
6864	\$1AD0	SPRITEIMGL	lo-byte pointer to your bit-mapped image
6880	\$1AE0	SPRITEX	X pixel coordinate (0-15; visible 16 – 191; 192-255)
6896	\$1AF0	SPRITEY	Y pixel coordinate (0-15; visible: 16 – 199; 200-255)
6912	\$1B00	SPRITEZ	sprite rendering flags: - bit 0: last rendered (0 = SPRITEC1; 1 = SPRITEC2) - bit 3: sprite collision (1 = true) - bit 4: sprite image is clipped by a static cell (1 = true) - bit 5: background is all SSSNULLs (1 = true) - bit 6: dirty flag – copy/merge into alternate character pool - bit 7: render flag – copy/shift sprite image into its buffer



VIC-SSS Run-Time Memory Map

Run-Time heap

Change these values if you want to use a different screen geometry and/or need to use VIC's internal memory differently:

	<u>Address</u>	<u>Constant</u>	<u>Description</u>
4096	\$1000	VICFRAME1	1 st video frame buffer
4608	\$1200	VICFRAME2	2 nd video frame buffer
5120	\$1400	PLAYFIELD	background video frame buffer
5632	\$1600	PLAYCOLOR	background color frame buffer, with dirty map: - bit 0-2: character color - bit 3: multi-color mode - bit 4: static cell – sprites cannot overwrite - bit 5: update pending video frame buffer - bit 6: update frame buffers #2 only - bit 7: update frame buffers #1 only
6144	\$1800	SSSBUF	sprite image buffers
6656	\$1A00	SPRITE*	sprite registers
6928	\$1B10	SSS, SSSNUM, . . .	sprite run-time storage for internal variables

Run-Time free

\$033C - \$03FB (192-bytes) is **DATASETTE** buffer working storage typically free to use.

\$1B80 - \$1BFF (128-bytes) with its size varying dependent upon number of sprites in use.

\$1C00 - \$1FFF is the default custom VIC character space, for your graphics and software sprites.

Run-Time program code

\$0400 - \$0FFF for VIC machines with the 3kb memory expansion module.

\$1000 - \$1FFF can be used ONCE until **SSSINIT** is invoked, i.e., startup splash screen, instructions, etc.

\$2000 - \$3FFF for 8kb memory expanded VIC games.

\$4000 - \$5FFF for 16kb memory expanded VIC games.

\$6000 - \$7FFF for 24kb memory expanded VIC games.



Resources



Author

For more details on my personal and professional background, feel free to visit my blog at <http://robert.hurst-ri.us>. I can be reached on Facebook and LinkedIn, too.

I maintain a classic computing tribute page with my history, software downloads, and useful Internet links to other 8-bit related sites at <http://robert.hurst-ri.us/retrocomputing>

There you can find several other video game projects, in addition to the included **Sprite Invaders**, such as **Berzerk MMX**, **Break-Out!**, **Quikman+**, and **Omega Fury** as working examples that integrate this software sprite stack.



Online

These have been useful and instrumental web sites to me, without which I would have not had any fun with these new VIC 20 projects:

6502 C Compiler and Assembler

<http://www.cc65.org>

Commodore Books

<http://www.bombjack.org/commodore/books.htm>

Denial: The Commodore VIC 20 Community

<http://www.sleepingelephant.com/denial>

MAME Debugger Help [M.E.S.S.]

<http://mess.redump.net/debugger>

SDLMESS: SDL port for Multiple Emulator Super System

http://rbelmont.mameworld.info/?page_id=163

The VICE Team

<http://www.viceteam.org>



Any technology distinguishable from [magic](#) is insufficiently advanced.

