Neo6502 API

The Neo6502 API is a messaging system. Messages are passed through a block of memory stored from $FF00 to $FF0F which is allocated as below.

There are *no* methods of directly accessing the hardware.

Address	Contents		
$FF00	Group. API Commands are grouped by functionality ; so group 1 is the system function, and group 2 is the console I/O function for example. Once a non zero value is written here the system will respond by setting values in the other registers appropriately, at the end they will clear this location.		
$FF01	Function. A command within the group, so for example Group 1 command 0 writes a value to the console, and Group 1 command 1 reads the keyboard		
$FF02	Return any error values, 0 = no error.		
$FF03	Information	7	Set to ‘1’ if the ESCape key has been pressed. This is not automatically reset.
		6	Unused
		5	Unused
		4	Unused
		3	Unused
		2	Unused
		1	Unused
		0	Unused
$FF04-B	Parameters, known as Params 0 through 7. These can be combined to form 16 or 32 bit integers.		
$FF0C-F	Reserved		

In the include file neo6502.inc the value of the first is the identifier ControlPort. This also has addresses of WaitMessage, SendMessage and some helper functions.

String parameters are *not ASCIIZ,* but are length prefixed ; the length of the string is the first byte in memory, the first character is the second byte.

Messages are sent as follows.

1	Wait for any pending command to complete. There is a subroutine WaitMessage which does this for the developer
2	Set up the parameters if any. For example printing a character to the console is done by putting its ASCII value into $FF04.
3	Setting the function code at $FF01
4	Writing the command to $FF00. This h*as* to be done last as setting it to non zero before the parameters are set up may cause the message to be processed. On a technical point, both implementations process the message immediately on write.
5	Optionally, wait for completion. Some commands, e.g. 2,2 which reads from the keyboard queue return a value in a parameter. Things like writing to the console do not need to wait for completion, as any subsequent command will wait for the command to complete as per 1.

There is a support function SendMessage which inlines the command and function e.g. this code from the Kernel.

```
    jsr  KSendMessage        ; send command 2,1 read keyboard
    .byte 2,1
    jsr  KWaitMessage        ; waiting for message to be sent back
    lda  DParameters         ; read result
```

You could write this as the following – it's just more longwinded.

```
    lda  #1                  ; do command 2,1
    sta  DFunction
    lda  #2
    sta       DCommand
Loop:
    lda  DCommand            ; signal done by this being zero
    bne  Loop
    lda  DParameters         ; get result
```

Group	Func	Description
1	0	Resets the messaging system and its components. Should not normally be used.
	1	Return the value of the 100Hz system timer in Params 0-3
	2	Return the state of keyboard key Param0 in Param0
	3	Execute BASIC *(Loading currently does not work)*
	4	Print the list of people involved, stored in Flash to save memory.
	5	Check the serial port to see if there is a data transmission. This is done automatically in Key Read.
	6	Sets the locale to P0P1 where P0P1 is the character code, e.g. P0='F' and P1 = 'R'
2	1	Read and remove a key press from the keyboard queue into Param0 , this is the ASCII value of the keystroke. If there are no key presses in the queue, Param0 is zero. Note that this method is *not* for games, where key presses and releases replace a joystick. The system maintains a bit array to check if keys are pressed.
	2	Check to see if the keyboard queue is empty. If it is Param0 is $FF, otherwise it is $00
	3	Input the line the screen is currently on to YX as a length prefixed string, put the cursor on the line below the line input, handles multiple line input.
	4	Define function key Param0 to be the length prefixed string at Param2/3
	5	Use bits 0..5 of P1-7 to define a font character P0 (192-255)
	6	Write character Param0 to the console. 32-127 are standard ASCII, 8 is Backspace, 13 Return. There are other console codes documented later.
	7	Set cursor position to Param0, Param1
	8	Display the current settings of the function keys
3	1	Display the directory
	2	Load a file from name Param0/1 (Length prefixed) to address Param2/3, error code in Param0. If the address is $FFFF the file is loaded into the graphic memory area used for sprites, tiles, images. For the extended format to work, the 65C02 needs to do a Jsr Param4 (e.g. jsr $FF08 if the param block is at $FF00) to implement any execution. This is done automatically by the kernel function LoadExtended
	3	Save a file to name Param0/1 (Length prefixed) from address Param2/3

		length Param4/5 bytes, error code in Param0.
	4-31	I/O functionality *under active development.*
	32	Display the directory using the string at P0P1 as a selector. (see CAT)
4	0-15	Binary mathematics operations
	16-31	Unary mathematics operations
	32-47	Miscellaneous operations.
5	1	Set Colour by performing the operation And P0, Xor P1 on the screen pixel. Sprite masking because of the two plane bitmap design is handled by the system. Solid flag in P2. Dimension in P3, Flip bits in P4 (0 = horizontal, 1 = vertical)
	2	Draw line. P01,P23 → P45,P67
	3	Draw rectangle. P01,P23 → P45,P67
	4	Draw ellipse. P01,P23 → P45,P67
	5	Draw pixel. P01,P23
	6	Draw string at P01,P23 text at P45 (length prefixed) in current col/size
	7	Draw image at P01,P4 is image ID, current size and flip
	8	Draw tilemap on screen from P0P1,P2P3 → P45P67 using current settings.
	32	Set palette colour P0 to P1,P2,P3 (RGB)
	33	
	34	Reset to default palette
	35	Set the current tilemap. P0P1 is the address in 6502 memory, P2P3 the x offset from the top left in pixels, and P4P5 the y offset in pixels.
	36	Read Pixel at P01,P23 from the sprite layer (0-15, 0 = transparency)
	37	Return the number of vblanks since power on in P0P1P2P3. This is updated at the start of the vblank period.
	64	Set the drawing colour to P0
	65	Set the solid flag to P0 (non-zero = solid)
	66	Set the dimension scaling flag to P0

	67	Set the bit flip flag to P0
6	1	Reset the sprite system
	2	Update Sprite P0 : Position is (P1P2,P3P4) Image is P5 (bits 0-5 are sprite number, bit 6 indicates 32 bit – NOT the same as the image number in the graphics system, bit 7 is clear), P6 the flip value (bit 0 horizontal, bit 1 vertical, bit 2-7 clear, P7 sets the anchor point. To not update a value set its byte values to $80 (or $8080 for a coordinate). The coordinates cannot be set independently Sprite 0 is the turtle sprite.
	3	Hide sprite P0
	4	P0 is non-zero if the distance between the centre of sprites P0 and P1 is less than or equal to P2
	5	Return coordinates of sprite P0 in P1P2,P3P4
7	1	Read default controller. Bits are (from zero) Left, Right, Up, Down, A, B ; active high, unused are zero. Keys are WASDOP and the cursor keys.
8	1	Reset the sound system, empty queue, silence
	2	Reset a sound channel, P0 = channel, indexed from zero
	3	Play the startup beep
	4	Queue a sound : P0 = channel, P1P2 = frequency P3P4 = duration in cs, P5P6 = slide change per cs, P7 = sound type (beeper = 0, the only type currently supported)
	5	Play sound effect P1 on channel P0 immediately, clearing the queue.
	6	Return in P0 the number of notes outstanding on channel P0 before silence, including any current playing note.
9	1	Initialise the turtle graphics sytem. P0 is the sprite number to use for the turtle, as the turtle graphics system “adopts” one of the sprites. The graphic is not currently redefinable, and initially the turtle is hidden.
	2	Turn the turtle right by P0P1 degrees. Show if hidden.
	3	Move the turtle forward by P0P1 degrees, drawing in colour P2, pen down flag in P3. Show if hidden.
	4	Hide the turtle
	5	Move the turtle to the home position (pointing up in the centre)
10	1	Initialise UEXT system
	2	Set UEXT Pin P0 to state P1. Returns error if not available.

	3	Read UEXT Pin P0 to P0. Returns error if not available.
	4	Set UEXT Direction for Pin P0 to P1 (1 = Input, 2 = Output)
	5	Write byte P2 to I2C Device P0 Register P1
	6	Read byte from I2C Device P0 Register P1 to P0

Anchor points

This shows the valid anchor positions for a sprite. The default is zero (the centre of the sprite). When you move a sprite, the sprite is aligned so that its anchor point is at the specified position.

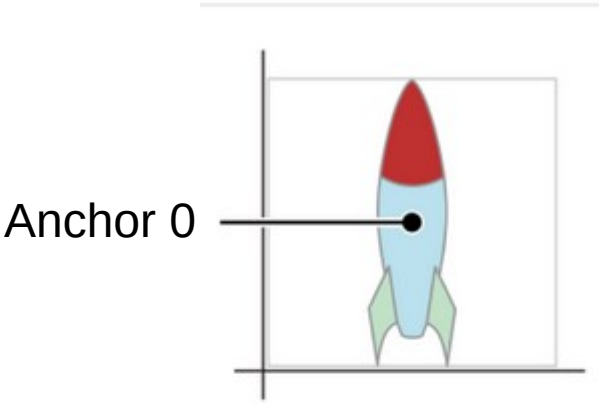
7	***8***	***9***
4	***0/5***	***6***
1	***2***	***3***

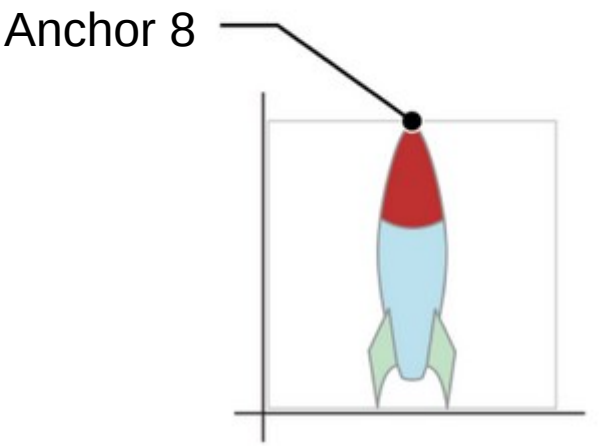
There are two examples here. Assume this is a 32x32 sprite.

In the upper example the anchor point is at 0, and this sprite is drawn at 16,16 (the middle of the square)

In the lower example, the anchor point is at 8, the upper central. This sprite is drawn at 16,32, the midpoint of the top of the square.

The anchor point should be something like the centre point. So for a walking character, this might be anchor point 2, the lower central point.





Console Codes

The following are console codes, and can be printed using chr$(n) and also related to the character keys returned by inkey$(). The key() function uses physical key numbers.

Not all control keys have matching keyboard keys, and not all console outputs are implemented ; some do nothing.

ASCII	Ctrl	Key	Console Output
1	A	Cursor Left	Cursor Left
4	D	Cursor Right	Cursor Right
5	E	Insert	Insert
6	F	Page Down	
7	G	End	
8	H	Backspace	Backspace
9	I	Tab	Horizontal Tab
10	J		Line Feed
12	L		Clear Screen
13	M	Enter	Carriage Return/Enter
18	R	Page Up	
19	S	Cursor Down	Cursor Down
20	T	Cursor Home	Cursor Home
22	V	Vertical Tab	Vertical Tab
23	W	Cursor Up	Cursor Up
24	X		Cursor Reverse
26	Z	Delete	Delete
27	[	Escape	General break – e.g. exits BASIC
20-7F		Standard ASCII	Standard ASCII
80-8F			Set Foreground to 0-F
90-9F			Set Background to 0-F
C0-FF			User definable characters

Tile Maps

A tile map occupies an area of memory in 65C02, and has one byte for each tile, which is the tile number in the graphic file, except for F0-FF. F0 is a transparent tile, and F1-FF are a solid tile in that palette colour.

The format is very simple.

Offset	Byte	Contents
0	1	Format ID
1	Width	Width of tilemap in tiles
2	Height	Height of tilemap in tiles
3 +	Data	Tile data size width * height

Graphic Data

Graphic files are most commonly identified as .gfx files, though this is not mandatory. Their format is quite simple.

Offset	Byte	Contents
0	1	Format ID
1	Tiles	Number of 16x16 tiles in use
2	Sprites16	Number of 16x16 sprites in use
3	Sprites32	Number of 32x32 sprites in use
4..255		Reserved
256	Data	Sprite data

Sprite data is all the 16x16 tiles, then all the 16x16 sprites, then all the 32x32 sprites in order. Each data item has 2 pixels per byte, the upper 4 bits being the first pixel colour and the lower 4 bits being the second pixel colour. So each tile takes 16x16/2 bytes (64 bytes) as does each sprite16, and each sprite32 takes 32x32/2 bytes (512 bytes).

Colour 0 is transparent for sprites (colour 9 should be used for a black pixel on a sprite)

As there is only about 20k of Graphics Memory at present these should be used somewhat sparingly.

There are tools for creating graphics files in OS independent Python in the release, which allows you to use tools like Gimp, Krita and Paint.Net to design graphics.

There is an example in the crossdev folder which gives some idea on how to get started.

Load file format

There is an extended file format which allows the loading of multiple files and optional execution. This is as follows

Offset	Contents	Notes
0	$03	Not a valid 65C02 opcode, nor can it be the first byte of a program.
1	$4E	ASCII ‘N’
2	$45	ASCII ‘E’
3	$4F	ASCII ‘O’
4,5	$00,$00	Minimum major/minor version required to work.
6,7	$FF,$FF	Execute address. To autorun a BASIC program set to $806
8	Control	Control bits, currently only bit 7 is used, which indicates another block follows this one
9,10	Load	Load address (16 bits) $FFFF loads into graphic object memory, $FFFD loads to the BASIC workspace.
11,12	Size	Size to load in bytes.
13…	Comment	ASCIIZ string which is a comment, filename, whatever
….	Data	The data itself
The block then repeats from ‘Control’ as many times as required.		

The Python application ‘exec.zip’ both constructs executable files, or displays them. This has the same execution format as the emulator, as listed below.

python exec.zip -d<file> dumps a file

python exec.zip <command list> -o<outputfile> builds a file

- <file>@page Loads BASIC program
- <file>@ffff Loads Graphics Object file
- <file>@<hex address> Loads arbitrary file
- run@<hex address> Sets the executable address
- exec runs BASIC program

for example, you can build a frogger executable with:

```
python exec.zip frogger.bas@page frogger.gfx@ffff exec -ofrogger.neo
```

Loading a file can be done by calling the kernel function LoadExtended (which does the autorun for you) or using the normal messaging system.

If you handle it yourself bear in mind that on return, it is always possible that the code you write to call the execution routine may already have been overwritten by the loaded file.