Neo6502 Assembly - Messaging API

Paul Robson
Bill Auger

2024-03-09

Contents

1 Neo6502 Messaging API **2**

2 API Functions **5**

3 Console Codes **23**

4 Graphics **24**

5 Tile Maps **27**

6 Sprites **28**

7 Sound **30**

1 Neo6502 Messaging API

The Neo6502 API is a messaging system. There are no methods to access the hardware directly. Messages are passed via the block of memory from `$FF00` to `$FF0F`, as specified in the "API Messaging Addresses" table on the following page.

The kernel include file `documents/release/neo6502.inc` specifies the beginning of this address range as the identifier `ControlPort`, along with the addresses of `WaitMessage` and `SendMessage` (described later), various related kernel jump vectors, and some helper functions.

The application include files `examples/assembly/neo6502.asm.inc` and `examples/C/neo6502.h` also specify the beginning of this address range as the identifier `ControlPort`. The assembly include also specifies `ControlPort` and the other controls as `API_COMMAND`, `API_FUNCTION`, `API_ERROR`, `API_STATUS`, and `API_PARAMETERS`. The C header also specifies `ControlPort` and the other controls as `API_COMMAND_ADDR`, `API_FUNCTION_ADDR`, `API_ERROR_ADDR`, `API_STATUS_ADDR`, and `API_PARAMETERS_ADDR`.

API Commands/Functions are grouped by functionality. For example, Group 1 are system-related, and Group 2 are console-I/O-related. All Groups and their Commands/Functions are shown in the following table.

Command/Function Parameters are notated in this document as `Params[0]` through `Params[7]`, or as a list or range (eg: `Params[1,2]`, `Params[0..7]`). Note that these are referring to a mapping to memory locations. The numbers represent offsets from the Parameters base address `$FF04`. Ie: the actual bytes are not necessarily all distinct "parameters" in the conventional sense. Depending on the routine, a logical parameter may be an individual byte, one or more bits of a byte interpreted as a composite or bit-field, or multiple adjacent bytes interpreted as 16 or 32 bit values. For example, the list `Params[0,1]` would indicate a single logical parameter, comprised of the two adjacent bytes `$FF04` and `$FF05`. The range `Params[4..7]` would indicate a single logical parameter, spanning consecutive bytes between `$FF08` and `$FF0B`.

Note that strings referenced by Parameters are not ASCIIZ, but are length-prefixed. The first byte represents the length of the string (not counting itself). The string begins at the second byte. Consequently, strings must be 256 bytes or less (not counting the length header).

API Messaging Addresses

Meta	Address		Contents
Group	$FF00		Group selector and status. Writing a non-zero value to this location triggers the routine specified in $FF01. The system will respond by setting the 'Error', 'Information', and 'Parameters' values appropriately. Upon completion, this memory location will be will cleared.
Function	$FF01		A command or function within the selected Group. For example, Group 1 Function 0 writes a value to the console; and Group 1 Function 1 reads the keyboard.
Error	$FF02		Return any error values, 0 = no error.
Information	$FF03	bit-7	Set (1) if the ESCape key has been pressed. This is not automatically reset.
		bit-6	*unused*
		bit-5	*unused*
		bit-4	*unused*
		bit-3	*unused*
		bit-2	*unused*
		bit-1	*unused*
		bit-0	*unused*
Parameters	$FF04..B		This memory block is notated in this document as Params[0] through Params[7], or as a composite list or range (eg: Params[1,2], Params[0..7]). Some Functions require Parameters in these locations and some return values in these locations; yet others do neither.
Reserved	$FF0C..F		Reserved.

1.1 API Interfacing Protocol

Neo6502 application programmers should interact with the API per the following algorithm:

1. Wait for any pending command to complete. There is a subroutine `WaitMessage` which does this for the developer.

2. Setup the Function code at `$FF01`; and any Parameters across `$FF04..$FF0B`. To print a character for example, set `$FF01` to `$06` and set `$FF04` to the character's ASCII value. To draw a line, set `$FF01` to `$02` and set `$FF04..$FF0B` as four 16-bit pixel coordinates:

   Params

P0	P1	P2	P3	P4	P5	P6	P7
$FF04	$FF05	$FF06	$FF07	$FF08	$FF09	$FF0A	$FF0B
srcX lo	srcX hi	srcY lo	srcY hi	destX lo	destX hi	destY lo	destY hi

3. Setup the command code at `$FF00`. This triggers the routine; so mind that the Function code and Parameters are setup sanely prior. On a technical point, both implementations process the message immediately on write.

4. Optionally, wait for completion. Most commands (eg: writing to the console) do not require waiting, as any subsequent command will wait/queue as per step 1. Query commands (e.g. reading from the keyboard queue), return a value in a parameter. Programs must wait until the control address `$FF00` has been cleared before reading the result of a query.

There is a support function `SendMessage`, which in-lines the command and function. E.g.: this code from the Kernel:

```
jsr KSendMessage ; send message for command 2,1 (read keyboard)
.byte 2,1
jsr KWaitMessage ; wait to receive the result message
lda DParameters  ; read result
```

The instructions above are equivalent to the following explicit steps:

```
lda #1
sta DFunction
lda #2
sta DCommand    ; trigger API function 2,1 (read keyboard)
Loop:
lda DCommand    ; load the result - non-zero until the routine's completion
bne Loop        ; wait for API routine to complete
lda DParameters ; read result (a key-code)
```

2 API Functions

The following tables are a comprehensive list of all supported API functions.

For the convenience of application programmers, the application include files `examples/C/neo6502.h` and `examples/assembly/neo6502.asm.inc` define macros for these groups, their functions, and common parameters (colors, musical notes, etc).

Group 1 - System Functions

Function	Assembly	Description
0 Reset	`LDA #$00` `STA #$FF01`	Resets the messaging system and component systems. Normally, should not be used.
1 Timer	`LDA #$01` `STA #$FF01`	Deposit the value (32-bits) of the 100Hz system timer into `Params[0..3]`.
2 Key Status	`LDA #$02` `STA #$FF01`	Deposit the state of the specified keyboard key into `Params[0]`. The key which to query is specified in `Params[0]`.
3 Basic	`LDA #$03` `STA #$FF01`	Loads and allows the execution of BASIC via a indirect jump through address zero.
4 Credits	`LDA #$04` `STA #$FF01`	Print the Neo6502 project contributors (stored in flash memory).
5 Serial Status	`LDA #$05` `STA #$FF01`	Check the serial port to see if there is a data transmission.
6 Locale	`LDA #$06` `STA #$FF01`	Set the locale code specified in `Params[0,1]` as upper-case ASCII letters. `Params[0]` takes the first letter and `Params[1]` takes the second letter. For example: English: `Params[0]<-$45` ('E') and `Params[1]<-$4E` ('N') French: `Params[0]<-$46` ('F') and `Params[1]<-$52` ('R')
7 System Reset	`LDA #$07` `STA #$FF01`	System Reset. This is a full hardware reset. It resets the RP2040 using the Watchdog timer, and this also resets the 65C02.

Group 2 - Console Functions

Function	Assembly	Description
0 Write Character	`LDA #$00` `STA #$FF01`	Function 0 is console out (duplicate of Function 6 for backward compatibility).
1 Read Character	`LDA #$01` `STA #$FF01`	Read and remove a key press from the keyboard queue into `Params[0]`. This is the ASCII value of the keystroke. If there are no key presses in the queue, `Params[0]` will be zero. Note that this Function is best for text input, but not for games. Function 7,1 is more optimal for games, as this only detects key presses, you cannot check whether the key is currently down or not.
2 Console Status	`LDA #$02` `STA #$FF01`	Check to see if the keyboard queue is empty. If it is, `Params[0]` will be `$FF`, otherwise it will be `$00`.
3 Read Line	`LDA #$03` `STA #$FF01`	Input the current line below the cursor into `Params[0,1]` as a length-prefixed string; and move the cursor to the line below. Handles multiple-line input.
4 Define Hotkey	`LDA #$04` `STA #$FF01`	Define the function key F1..F10 (`$01..$0A`) specified in `Params[0]` to emit the length-prefixed string stored at the memory location specified in `Params[2,3]`. For example, in a block of in-line assembly within a BASIC program, the string: `06,12,108,105,115,116,13` would clear the screen (`12`), then list the program (`108`='l',`105`='i',`115`='s',`116`='t',`13`='ENTER'). F11 and F12 cannot currently be defined.
5 Define Character	`LDA #$05` `STA #$FF01`	Define a font character specified in `Params[0]` within the range of 192..255. Fill bits 0..5 (columns) of `Params[1..7]` (rows) with the character bitmap.
6 Write Character	`LDA #$06` `STA #$FF01`	Write the character specified in `Params[0]` to the console at the cursor position. Refer to Section #3 "Console Codes" for details.
7 Set Cursor Pos	`LDA #$07` `STA #$FF01`	Move the cursor to the screen character cell `Params[0]`<-X, `Params[1]`<-Y.
8 List Hotkeys	`LDA #$08` `STA #$FF01`	Display the current function key definitions.
9 Screen Size	`LDA #$09` `STA #$FF01`	Fetches the console size, in characters, to `Params[0]` and `Params[1]`, the height and width respectively.
10 Insert Line	`LDA #$010` `STA #$FF01`	Insert Line This is a support function which inserts a blank line in the console and should not be used.
11 Delete Line	`LDA #$011` `STA #$FF01`	Delete Line This is a support function which deletes a line in the console and should not be used.
12 Clear Screen	`LDA #$012` `STA #$FF01`	Clears the screen.

Group 2 - Console Functions (continued)

Function	Assembly	Description
13 Cursor Position	`LDA #$013` `STA #$FF01`	Get Cursor Position Returns the current cursor position in `Params[0]` and `Params[1]`
14 Clear Region	`LDA #$014` `STA #$FF01`	Erase all characters within the rectangular region specified in `Params[0,1]` (begin X,Y) and `Params[2,3]` (end X,Y).
15 Set Text Color	`LDA #$015` `STA #$FF01`	Set Text Color Sets the foreground colour to `Params[0]` and the background colour to `Params[1]`
16 Cursor Inverse	`LDA #$016` `STA #$FF01`	Toggles the cursor colour between normal and inverse (ie: swaps FG and BG colors). This should not be used.
17 Tab() implementation	`LDA #$017` `STA #$FF01`	Internal helper.

Group 3 - File I/O Functions

Function	Assembly	Description
1 List Directory	`LDA #$01` `STA #$FF01`	Display the file listing of the present directory.
2 Load File	`LDA #$02` `STA #$FF01`	Load a file by name into memory. On input: • `Params[0,1]` points to the length-prefixed filename string; • `Params[2,3]` contains the location to write the data to. If the address is `$FFFF`, the file will instead be loaded into the graphics working memory, used for sprites, tiles, images. On output: • `Params[0]` contains an error/status code.
3 Store File	`LDA #$03` `STA #$FF01`	Saves data in memory to a file. On input: • `Params[0,1]` points to the length-prefixed filename string; • `Params[2,3]` contains the location to read data from; • `Params[4,5]` specified the number of bytes to styore. On output: • `Params[0]` contains an error/status code.

Group 3 - File I/O Functions (continued)

Function	Assembly	Description
4 File Open	`LDA #$04` `STA #$FF01`	Opens a file into a specific channel. On input: • `Params[0]` contains the file channel to open; • `Params[1,2]` contains the length-prefixed filename; • `Params[3]` contains the open mode. See below. Valid open modes are: • 0 opens the file for read-only access; • 1 opens the file for write-only access; • 2 opens the file for read-write access; • 3 creates the file if it doesn't already exist, truncates it if it does, and opens the file for read-write access. Modes 0 to 2 will fail if the file does not already exist. If the channel is already open, the call fails. Opening the same file more than once on different channels has undefined behaviour and is not recommended.
5 File Close	`LDA #$05` `STA #$FF01`	Closes a particular channel. On input: • `Params[0]` contains the file channel to close.
6 File Seek	`LDA #$06` `STA #$FF01`	Seeks the file opened on a particular channel to a location. On input: • `Params[0]` contains the file channel to operate on; • `Params[1,2,3,4]` contains the file location. You can seek beyond the end of a file to extend the file. Whether the file size changes when the seek happens or when you perform the write is undefined.
7 File Tell	`LDA #$07` `STA #$FF01`	Returns the current seek location for the file opened on a particular channel. On input: • `Params[0]` contains the file channel to operate on. On output: • `Params[1,2,3,4]` contains the file location.

Group 3 - File I/O Functions (continued)

Function	Assembly	Description
8 File Read	`LDA #$08` `STA #$FF01`	Reads data from an opened file. On input: • `Params[0]` contains the file channel to operate on; • `Params[1,2]` points to the destination in memory, or `$FFFF` to write to graphics memory; • `Params[3,4]` contains the amount of data to read. On output: • `Params[3,4]` is updated to contain the amount of data actually read. Data is read from the current seek position, which is advanced after the read.
9 File Write	`LDA #$09` `STA #$FF01`	Writes data to an opened file. On input: • `Params[0]` contains the file channel to operate on; • `Params[1,2]` points to the data in memory; • `Params[3,4]` contains the amount of data to write. On output: • `Params[3,4]` is updated to contain the amount of data actually written. Data is written to the current seek position, which is advanced after the write.
10 File Size	`LDA #$010` `STA #$FF01`	Returns the current size of an opened file. On input: • `Params[0]` contains the file channel to operate on. On output: • `Params[1,2,3,4]` contains the size of the file. This call should be used on open files and takes into account any buffered data which has not yet been written to disk. As a result it may return a different size to the stat API call described below.

Group 3 - File I/O Functions (continued)

Function	Assembly	Description
11 File Set Size	`LDA #$011` `STA #$FF01`	Extends or truncates an opened file to a particular size. On input: • `Params[0]` contains the file channel to operate on; • `Params[1,2,3,4]` contains the new size of the file.
12 File Rename	`LDA #$012` `STA #$FF01`	Renames a file. On input: • `Params[0,1]` points to the length-prefixed string for the old name; • `Params[2,3]` points to the length-prefixed string for the new name. Files may be renamed across directories.
13 Delete File	`LDA #$013` `STA #$FF01`	Deletes a file or directory. On input: • `Params[0,1]` points to the length-prefixed filename string. Deleting a file which is open has undefined behaviour. Directories may only be deleted if they are empty.
14 Create Directory	`LDA #$014` `STA #$FF01`	Creates a new directory. On input: • `Params[0,1]` points to the length-prefixed filename string.
15 Change Directory	`LDA #$015` `STA #$FF01`	Changes the current working directory. On input: • `Params[0,1]` points to the length-prefixed path string.

Group 3 - File I/O Functions (continued)

<table>
<tr><th>Function</th><th>Assembly</th><th>Description</th></tr>
<tr><td>16
Stat File</td><td>LDA #$016
STA #$FF01</td><td>Retrieves information about a file by name. On input:
Params[0,1] points to the length-prefixed filename string.
On output:
Params[0,1,2,3] contains the length of the file;
Params[4] contains the attribute bitfield of the file.
If the file is open for writing, this may not return the correct size due to buffered data not having been flushed to disk. File attributes are a bitfield as follows:

File attributes<table><tr><td>bit-7</td><td>bit-6</td><td>bit-5</td><td>bit-4</td><td>bit-3</td><td>bit-2</td><td>bit-1</td><td>bit-0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>Hidden</td><td>Read only</td><td>Archive</td><td>System</td><td>Directory</td></tr></table></td></tr>
<tr><td>17
Open Directory</td><td>LDA #$017
STA #$FF01</td><td>Opens a directory for enumeration. On input:
Params[0,1] points to the length-prefixed filename string.
Only one directory at a time may be opened. If a directory is already open when this call is made, it is automatically closed; however, an open directory may make it impossible to delete the directory, so closing the directory after use is good practice.</td></tr>
<tr><td>18
Read Directory</td><td>LDA #$018
STA #$FF01</td><td>Reads an item from the currently open directory. On input:
Params[0,1] points to a length-prefixed buffer for returning the filename.
Params[0,1] is unchanged, but the buffer is updated to contain the length-prefixed filename (without any leading path);
Params[2,3,4,5] contains the length of the file;
Params[6] contains the file attributes, as described by the Stat File call.
This call fails if there are no more items to read.</td></tr>
<tr><td>19
Close Directory</td><td>LDA #$019
STA #$FF01</td><td>Closes any directory opened by Open Directory.</td></tr>
</table>

Group 3 - File I/O Functions (continued)

Function	Assembly	Description
20 Copy File	`LDA #$020` `STA #$FF01`	Copies a file. On input: • `Params[0,1]` points to the length-prefixed old filename; • `Params[2,3]` points to the length-prefixed new filename. Only single files may be copied, not directories.
21 Set file attributes	`LDA #$021` `STA #$FF01`	Sets the attributes for a file. On input: • `Params[0,1]` points to the length-prefixed filename; • `Params[2]` is the attribute bitfield. (See Stat File for details.) The directory bit cannot be changed. Obviously.
32 List Filtered	`LDA #$032` `STA #$FF01`	Prints a filtered file listing of the current directory to the console. On input: • `Params[0,1]` points to the filename search string. Files will only be shown if the name contains the search string (via a substring match).

Group 4 - Mathematics Functions

Function	Assembly	Description
0 Addition	`LDA #$00` `STA #$FF01`	Addition Register1 := Register 1 + Register2
1 Subtraction	`LDA #$01` `STA #$FF01`	Subtraction Register1 := Register 1 - Register2
2 Multiplication	`LDA #$02` `STA #$FF01`	Multiplication Register1 := Register 1 * Register2
3 Decimal Division	`LDA #$03` `STA #$FF01`	Decimal Division Register1 := Register 1 / Register2 (floating point)
4 Integer Division	`LDA #$04` `STA #$FF01`	Integer Division Register1 := Register 1 / Register2 (integer result)
5 Integer Modulus	`LDA #$05` `STA #$FF01`	Integer Modulus Register1 := Register 1 mod Register2
6 Compare	`LDA #$06` `STA #$FF01`	Compare Numbers `Params[0]` := Register 1 compare Register2 : returns $FF, 0, 1 for less equal and greater Note: float comparison is approximate because of rounding.
16 Negate	`LDA #$016` `STA #$FF01`	Negate Register1 := -Register 1
17 Floor	`LDA #$017` `STA #$FF01`	Floor Register1 := floor(Register 1)
18 Square Root	`LDA #$018` `STA #$FF01`	Square Root Register1 := square root(Register 1)
19 Sine	`LDA #$019` `STA #$FF01`	Sine Register1 := sine(Register 1) angles in degrees
20 Cosine	`LDA #$020` `STA #$FF01`	Cosine Register1 := cosine(Register 1) angles in degrees
21 Tangent	`LDA #$021` `STA #$FF01`	Tangent Register1 := tangent(Register 1) angles in degrees
22 Arctangent	`LDA #$022` `STA #$FF01`	Arctangent Register1 := arctangent(Register 1) angles in degrees
23 Exponent	`LDA #$023` `STA #$FF01`	Exponent Register1 := e to the power of Register 1
24 Logarithm	`LDA #$024` `STA #$FF01`	Logarithm Register1 := log(Register 1) natural logarithm
25 Absolute Value	`LDA #$025` `STA #$FF01`	Absolute Value Register1 := absolute value(Register 1)
26 Sign	`LDA #$026` `STA #$FF01`	Sign Register1 := sign(Register 1), returns -1 0 or 1

Group 4 - Mathematics Functions (continued)

Function	Assembly	Description
27 Random Decimal	`LDA #$027` `STA #$FF01`	Random Decimal Register1 := random float from 0-1
28 Random Integer	`LDA #$028` `STA #$FF01`	Random Integer Register1 := random integer from 0 to (Register 1-1)
32 Number to Decimal	`LDA #$032` `STA #$FF01`	Number to Decimal Helper function for tokeniser, do not use.
33 String to Number	`LDA #$033` `STA #$FF01`	String to Number Convert the length prefixed string at `Params[4,5]` to a constant in Register1.
34 Number to String	`LDA #$034` `STA #$FF01`	Number to String TODO: Convert the constant in Register1 to a length prefixed string which is stored at `Params[4,5]`

Group 5 - Graphics Functions

<table>
<tr><th>Function</th><th>Assembly</th><th>Description</th></tr>
<tr><td>1
Set Defaults</td><td>LDA #$01
STA #$FF01</td><td>Configure the global graphics system settings. Not all parameters are relevant for all graphics commands; but all parameters will be set by this command. So mind their values. Refer to Section #4.1 "Graphics Settings" for details.

Graphics Settings
<table>
<tr><td>P0</td><td>P1</td><td>P2</td><td>P3</td><td>P4</td><td>P5</td><td>P6</td><td>P7</td></tr>
<tr><td>$FF04</td><td>$FF05</td><td>$FF06</td><td>$FF07</td><td>$FF08</td><td>$FF09</td><td>$FF0A</td><td>$FF0B</td></tr>
<tr><td>AND</td><td>XOR</td><td>Fill</td><td>Extent</td><td>Flip</td><td>unused</td><td>unused</td><td>unused</td></tr>
</table>
$FF08 - Flip Axis Flags
<table>
<tr><td>bit-7</td><td>bit-6</td><td>bit-5</td><td>bit-4</td><td>bit-3</td><td>bit-2</td><td>bit-1</td><td>bit-0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>Vertical</td><td>Horizontal</td></tr>
</table></td></tr>
<tr><td>2
Draw Line</td><td>LDA #$02
STA #$FF01</td><td>Draw a line between the screen coordinates specified in Params[0,1],Params[2,3] (begin X,Y) and Params[4,5],Params[6,7] (end X,Y).

Draw Line Parameters
<table>
<tr><td>P0</td><td>P1</td><td>P2</td><td>P3</td><td>P4</td><td>P5</td><td>P6</td><td>P7</td></tr>
<tr><td>$FF04</td><td>$FF05</td><td>$FF06</td><td>$FF07</td><td>$FF08</td><td>$FF09</td><td>$FF0A</td><td>$FF0B</td></tr>
<tr><td>X lo</td><td>X hi</td><td>Y lo</td><td>Y hi</td><td>X' lo</td><td>X' hi</td><td>Y' lo</td><td>Y' hi</td></tr>
</table></td></tr>
<tr><td>3
Draw Rectangle</td><td>LDA #$03
STA #$FF01</td><td>Draw a rectangle spanning the screen coordinates specified in Params[0,1],Params[2,3] (corner X,Y) and Params[4,5],Params[6,7] (opposite corner X,Y).</td></tr>
<tr><td>4
Draw Ellipse</td><td>LDA #$04
STA #$FF01</td><td>Draw an ellipse spanning the screen coordinates specified in Params[0,1],Params[2,3] (corner X,Y) and Params[4,5],Params[6,7] (opposite corner X,Y).</td></tr>
<tr><td>5
Draw Pixel</td><td>LDA #$05
STA #$FF01</td><td>Draw a single pixel at the screen coordinates specified in Params[0,1],Params[2,3] (X,Y).</td></tr>
<tr><td>6
Draw Text</td><td>LDA #$06
STA #$FF01</td><td>Draw the length-prefixed string of text stored at the memory location specified in Params[4,5] at the screen character cell specified in Params[0,1],Params[2,3] (X,Y).</td></tr>
<tr><td>7
Draw Image</td><td>LDA #$07
STA #$FF01</td><td>Draw the image with image ID in Params[4] at the screen coordinates Params[0,1],Params[2,3] (X,Y). The extent and flip settings influence this command.</td></tr>
<tr><td>8
Draw Tilemap</td><td>LDA #$08
STA #$FF01</td><td>Draw the current tilemap at the screen coordinates specified in Params[0,1],Params[2,3] (top-left X,Y) and Params[4,5],Params[6,7] (bottom-right X,Y) using current graphics settings.</td></tr>
</table>

Group 5 - Graphics Functions (continued)

Function	Assembly	Description
32 Set Palette	`LDA #$032` `STA #$FF01`	Set the palette colour at the index spcified in `Params[0]` to the values in `Params[1]`,`Params[2]`,`Params[3]` (RGB).
33 Read Pixel	`LDA #$033` `STA #$FF01`	Read a single pixel at the screen coordinates specified in `Params[0,1]`,`Params[2,3]` (X,Y). When the routine completes, the result will be in `Params[0]`. If sprites are in use, this will be the background only (0..15), if sprites are not in use it may return (0..255)
34 Reset Palette	`LDA #$034` `STA #$FF01`	Reset the palette to the defaults.
35 Set Tilemap	`LDA #$035` `STA #$FF01`	Set the current tilemap. `Params[0,1]` is the memory address of the tilemap, and `Params[2,3]`,`Params[4,5]` (X,Y) specifies the offset into the tilemap, in units of pixels, of the top-left pixel of the tile.
36 Read Sprite Pixel	`LDA #$036` `STA #$FF01`	Read Pixel from the sprite layer at the screen coordinates specified in `Params[0,1]`,`Params[2,3]` (X,Y). When the routine completes, the result will be in `Params[0]`. Refer to Section #**??** "Pixel Colors" for details.
37 Frame Count	`LDA #$037` `STA #$FF01`	Deposit into `Params[0..3]`, the number of v-blanks (full screen re-draws) which have occurred since power-on. This is updated at the start of each v-blank period.
64 Set Color	`LDA #$064` `STA #$FF01`	Set Color Sets the current drawing colour to `Params[0]`
65 Set Solid Flag	`LDA #$065` `STA #$FF01`	Set Solid Flag Sets the solid flag to `Params[0]`, which indicates either solid fill (for shapes) or solid background (for images and fonts)
66 Set Draw Size	`LDA #$066` `STA #$FF01`	Set Draw Size Sets the drawing scale for images and fonts to `Params[0]`
67 Set Flip Bits	`LDA #$067` `STA #$FF01`	Set Flip Bits Sets the flip bits for drawing images. Bit 0 set causes a horizontal flip, bit 1 set causes a vertical flip.

Group 6 - Sprites Functions

<table>
<tr><th>Function</th><th>Assembly</th><th>Description</th></tr>
<tr><td>1
Sprite Reset</td><td>LDA #$01
STA #$FF01</td><td>Reset the sprite system.</td></tr>
<tr><td>2
Sprite Set</td><td>LDA #$02
STA #$FF01</td><td>Set or update the sprite specified in Params[0].

Sprite Parameters
<table>
<tr><td>P0</td><td>P1</td><td>P2</td><td>P3</td><td>P4</td><td>P5</td><td>P6</td><td>P7</td></tr>
<tr><td>$FF04</td><td>$FF05</td><td>$FF06</td><td>$FF07</td><td>$FF08</td><td>$FF09</td><td>$FF0A</td><td>$FF0B</td></tr>
<tr><td>Sprite</td><td>X lo</td><td>X hi</td><td>Y lo</td><td>Y hi</td><td>Image</td><td>Flip</td><td>Anchor</td></tr>
</table>
$FF09 - Image Parameters
<table>
<tr><td>bit-7</td><td>bit-6</td><td>bit-5</td><td>bit-4</td><td>bit-3</td><td>bit-2</td><td>bit-1</td><td>bit-0</td></tr>
<tr><td>0</td><td>32-bit</td><td colspan="6">Index</td></tr>
</table>
$FF0A - Flip Axis Flags
<table>
<tr><td>bit-7</td><td>bit-6</td><td>bit-5</td><td>bit-4</td><td>bit-3</td><td>bit-2</td><td>bit-1</td><td>bit-0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>Vertical</td><td>Horizontal</td></tr>
</table>
values that are $80 or $8080 are not updated.</td></tr>
<tr><td>3
Sprite Hide</td><td>LDA #$03
STA #$FF01</td><td>Hide the sprite specified in Params[0].</td></tr>
<tr><td>4
Sprite Collision</td><td>LDA #$04
STA #$FF01</td><td>Params[0] is non-zero if the distance is less than or equal to Params[2] between the center of the sprite with index specified in Params[0] and the center of the sprite with index specified in Params[1] .</td></tr>
<tr><td>5
Sprite Position</td><td>LDA #$05
STA #$FF01</td><td>Deposit into Params[1..4], the screen coordinates of the sprite with the index specified in Params[0].</td></tr>
</table>

Group 7 - Controller Functions

<table>
<tr><th>Function</th><th>Assembly</th><th>Description</th></tr>
<tr><td>1
Read Controller</td><td>LDA #$01
STA #$FF01</td><td>This reads the status of the default controller into Params[0]
Initially, the Controller is the keyboard. This Function interprets key presses and releases as a joystick. The system maintains a bit-array of which keys are pressed. Currently, the keyboard is the only available Controller.

This function will become obsolete shortly as a more extensive API using USB controllers becomes available. However, it will maintain backward compatibility.

$FF04 - Controller Flags
<table>
<tr><td>bit-7</td><td>bit-6</td><td>bit-5</td><td>bit-4</td><td>bit-3</td><td>bit-2</td><td>bit-1</td><td>bit-0</td></tr>
<tr><td>0</td><td>0</td><td>B</td><td>A</td><td>Down</td><td>Up</td><td>Right</td><td>Left</td></tr>
</table>
</td></tr>
</table>

Group 8 - Sound Functions

<table>
<tr><th>Function</th><th>Assembly</th><th>Description</th></tr>
<tr><td>1
Reset Sound</td><td>LDA #$01
STA #$FF01</td><td>Reset the sound system. This empties all channel queues and silences all channels immediately.</td></tr>
<tr><td>2
Reset Channel</td><td>LDA #$02
STA #$FF01</td><td>Reset the sound channel specified in Params[0].</td></tr>
<tr><td>3
Beep</td><td>LDA #$03
STA #$FF01</td><td>Play the startup beep immediately.</td></tr>
<tr><td>4
Queue Sound</td><td>LDA #$04
STA #$FF01</td><td>Queue a sound. Refer to Section #7 "Sound" for details.

Queue Sound Parameters
<table>
<tr><td>P0</td><td>P1</td><td>P2</td><td>P3</td><td>P4</td><td>P5</td><td>P6</td><td>P7</td></tr>
<tr><td>$FF04</td><td>$FF05</td><td>$FF06</td><td>$FF07</td><td>$FF08</td><td>$FF09</td><td>$FF0A</td><td>$FF0B</td></tr>
<tr><td>Chan</td><td>Frq hi</td><td>Frq lo</td><td>Dur lo</td><td>Dur hi</td><td>Sld lo</td><td>Sld hi</td><td>Source</td></tr>
</table></td></tr>
<tr><td>5
Play Sound</td><td>LDA #$05
STA #$FF01</td><td>Play the sound effect specified in Params[1] on the channel specified in Params[0] immediately, clearing the channel queue.</td></tr>
<tr><td>6
Sound Status</td><td>LDA #$06
STA #$FF01</td><td>Deposit in Params[0] the number of notes outstanding before silence in the queue of the channel specified in Params[0], including the current playing sound, if any.</td></tr>
</table>

Group 9 - Turtle Graphics Functions

Function	Assembly	Description
1 Turtle Initialise	`LDA #$01` `STA #$FF01`	Initialise the turtle graphics system. `Params[0]` is the sprite number to use for the turtle, as the turtle graphics system "adopts" one of the sprites. The icon is not currently re-definable, and initially the turtle is hidden.
2 Turtle Turn	`LDA #$02` `STA #$FF01`	Turn the turtle right by `Params[0]Params[1]` degrees. Show if hidden. To turn left, turn by a negative amount.
3 Turtle Move	`LDA #$03` `STA #$FF01`	Move the turtle forward by `Params[0]Params[1]` degrees, drawing in colour `Params[2]`, pen down flag in `Params[3]`. Show if hidden.
4 Turtle Hide	`LDA #$04` `STA #$FF01`	Hide the turtle.
5 Turtle Home	`LDA #$05` `STA #$FF01`	Move the turtle to the home position (in the center, pointing upward).

Group 10 - UExt I/O Functions

Function	Assembly	Description
1 UExt Initialise	`LDA #$01` `STA #$FF01`	Initialise the UExt I/O system. This resets the IO system to its d state, where all UEXT pins are I/O pins, inputs and enabled.
2 Write GPIO	`LDA #$02` `STA #$FF01`	(P0,P1) This copies the value `Params[1]` to the output latch for UEX `Params[0]`. This will only display on the output pin if it is en and its direction is set to output.
3 Read GPIO	`LDA #$03` `STA #$FF01`	P0 = GPIO(P0) If the pin is set to input, reads the level on pin on UEX `Params[0]`. If it is set to output this reads the output latch for UEXT port `Params[0]`
4 Set Port Direction	`LDA #$04` `STA #$FF01`	P0 to P1 Set the port direction for UEXT Port `Params[0]` to `Params[1]` can be 1 (Input) 2 (Output) or 3 (Analogue)
5 Write I2C	`LDA #$05` `STA #$FF01`	(P0,P1,P2) Write to I2C Device `Params[0]`, Register `Params[1]` `Params[2]`. Does not fail if device not present.
6 Read I2C	`LDA #$06` `STA #$FF01`	P0 = (P0,P1) Read from I2C Device `Params[0]`, Register `Params[1]`. If the is not present this will cause an error.
7 Read Analog	`LDA #$07` `STA #$FF01`	P1P0 = GPIOAnalogue(P0) Read the analogue value on UEXT Pin `Params[0]` ; this has set to analogue type to work. Returns a value from 0..4095 sto `Params[0,1]`, which represents an input value of 0 to 3.3 volts
8 Check if can read register	`LDA #$08` `STA #$FF01`	P0 = Scan(P0) Try to read from I2C Device `Params[0]`. If present then `Para` will contain a non-zero value.
9 Read I2C Block into memory	`LDA #$09` `STA #$FF01`	Read I2C Block(P0,P2P1,P4P3) Try to read a block of memory from I2C Device `Params[0]` into ory at `Params[1,2]` length `Params[3,4]`
10 Write I2C Block from memory	`LDA #$010` `STA #$FF01`	Write I2C Block(P0,P2P1,P4P3) Try to write a block of memory to I2C Device `Params[0]` from m at `Params[1,2]` length `Params[3,4]`
11 Read SPI Block into memory	`LDA #$011` `STA #$FF01`	Read SPI Block(P2P1,P4P3) Try to read a block of memory from SPI Device into mem `Params[1,2]` length `Params[3,4]`
12 Write SPI Block from memory	`LDA #$012` `STA #$FF01`	Write SPI Block(P2P1,P4P3) Try to write a block of memory to SPI Device from mem `Params[1,2]` length `Params[3,4]`

3 Console Codes

The following are console key codes. They can be printed in BASIC programs using `chr$(n)`, and are also related to the character keys returned by `inkey$()`. The `key()` function uses physical key numbers. Some control codes do not have corresponding keyboard keys; and some console outputs are not yet implemented.

`Backspace` (8), `Tab` (9), `Enter/CR` (13), `Escape` (27), and the printable characters (32..127) are the standard ASCII set. Other typical control keys (eg: Home and arrows) are mapped into the 0..31 range.

Console Key Codes - Non-Printable

ASCII	CTRL+Key	Key	Output
1	A	Left Arrow	Cursor Left
4	D	Right Arrow	Cursor Right
5	E	Insert	Insertion Mode
6	F	Page Down	Cursor Page Down
7	G	End	Cursor Line End
8	H	Backspace	Delete Character Left
9	I	Tab	Tab Character
10	J		Line Feed
12	L		Clear Screen
13	M	Enter	Carriage Return (Accept Line)
18	R	Page Up	Cursor Page Up
19	S	Down	Cursor Down
20	T	Home	Cursor Line Begin
22	V		Cursor Down (8 Lines)
23	W	Up	Cursor Up
24	X		Cursor Color Inverse
26	Z	Delete	Delete Character Right
27	[	Escape	Exit

Console Key Codes - Printable

Hex	Key	Output
20-7F	ASCII Set	Standard ASCII Characters
80-8F		Set Foreground Color
90-9F		Set Background Color
C0-FF		User-definable Characters

4 Graphics

4.1 Graphics Settings

Function 5,1 configures the global graphics system settings. Not all Parameters are relevant for all graphics commands; but all Parameters will be set by this command. So mind their values.

The actual color of each drawn pixel will be computed at runtime by AND'ing the existing pixel color with the value specified in `Params[0]`, then XOR'ing the result with the value specified in `Params[1]`.

The value in `Params[2]` is a flag which determines the paint fill mode for the Draw Rectangle and Draw Ellipse commands: reset (`0`) for outline, set (`1`) for solid fill.

The value in `Params[3]` is the draw extent (window) for the Draw Image command.

The value in `Params[4]` is a bit-field of flags for the Draw Image command, which determine if the image will be inverted (flipped) horizontally or vertically: `bit-0` for horizontal, `bit-1` for vertical, reset (`0`) for normal, set (`1`) for inverted.

For the "Draw Rectangle" and "Draw Ellipse" commands, the given order and position of the coordinates are not significant. To be precise, one is "a corner" and the other is "the opposite corner". For the "Draw Ellipse" command, these corners are referring to the bounding-box. The coordinates for an ellipse will lie outside of the ellipse itself.

4.2 Graphics Data

Graphics data files are conventionally named ending in the .gfx suffix; though this is not mandatory. The format is quite simple.

Graphics Data Format

Offset	Data	Description
0	1	Graphics Data Format ID
1	Count	Number of 16x16 tiles in use
2	Count	Number of 16x16 sprites in use
3	Count	Number of 32x32 sprites in use
4..255		Reserved
256	Raw	Sprites graphics data

The layout of sprites graphics data is all of the 16x16 tiles, followed by all the 16x16 sprites, followed by all the 32x32 sprites, all in their respective orders. As there is currently only about 20kB of Graphics Memory, these should be used somewhat sparingly.

Each byte specifies 2 pixels. The upper 4 bits represent the first pixel colour; and the lower 4 bits represent the second pixel colour. So tiles and 16x16 sprites occupy 16x16/2 bytes (128 bytes) each. Each 32x32 sprite occupies 32x32/2 bytes (512 bytes). Colour 0 is transparent for sprites (colour 9 should be used for a black pixel).

The release package includes Python scripts for creating graphics files, which allow you to design graphics using your preferred editing tools (eg: Gimp, Inkscape, Krita, etc). There is an example in the `crossdev/` directory, which demonstrates how to get started importing graphics into the Neo6502.

4.3 Pixel Colors

Pixel Colors

Byte	Color
$80	Black/Transparent
$81	Red
$82	Green
$83	Yellow
$84	Blue
$85	Magenta
$86	Cyan
$87	White
$88	Black
$89	Dark Grey
$8A	Dark Green
$8B	Orange
$8C	Dark Orange
$8D	Brown
$8E	Pink
$8F	Light Grey

5 Tile Maps

A tile map occupies an area of user memory in 65C02. It is comprised of three meta-data bytes, followed by one byte for each tile, which is it's tile number in the graphic file (refer to the following section).

`F0-FF` are special reserved tile numbers, `F0` is a transparent tile; and `F1-FF` are a solid tile in the current palette colour. The format is very simple.

Tile Maps Format

Offset	Data	Description
0	1	Graphics Data Format ID
1	Width	Width of tile-map (number of tiles)
2	Height	Height of tile-map (number of tiles)
3..	Raw	Tiles graphics data (width * height bytes)

6 Sprites

The Neo6502 graphics system has one sprite layer (z-plane) in the conventional sense. Technically, there is no "sprite layer", per-se. The system uses palette manipulation to create, what is in practice, a pair of 4-bit bit-planes. The sprite graphics are in the upper nibble, the background is in the lower nibble, and the background is drawn only if the sprite graphic layer is zero. It's this top nibble which is read by Function 5,36 "Read Sprite Pixel".

Function 6,2 sets or updates a sprite. These parameters (eg: the X and Y coordinates) cannot be set independently. To retain/reuse the current value of a parameter for a subsequent call, set each of the associated byte(s) to `$80` (eg: `$80,$80,$80,$80` for coordinates).

The 'Sprite' Parameter `Params[0]` specifies the index of the sprite in the graphics system. Sprite 0 is the turtle sprite.

`Params[1,2]`,`Params[3,4]` specifies the X and Y screen coordinates.

Bits 0-5 of the 'Image' Parameter `Params[5]` specify the index of a specific graphic within the sprites data. Bit 6 of the 'Image' Parameter specifies the sprite dimensions: reset (`0`) for 16x16, set (`1`) for 32x32. In practice, the index is the same as the sprite number (`$80-$BF` for 16x16 sprites, `$C0-$FF` for 32x32 sprites), but without bit-7 set.

The value in `Params[6]` specifies a bit-field of flags, which determines if the graphic will be inverted (flipped) horizontally or vertically: `bit-0` for horizontal, `bit-1` for vertical, reset (`0`) for normal, set (`1`) for inverted.

`Params[7]` specifies the anchor alignment. Refer to Section #6.1 "Sprite Anchors" for details.

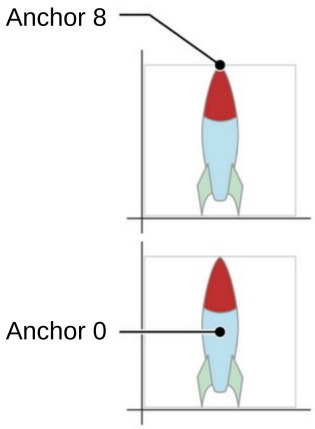
6.1 Sprite Anchors

The table below shows the valid anchor alignments for a sprite. The anchor position is the origin of the relative coordinate given. That is, coordinates 0,0 of the sprite will coincide with one of the positions shown in the table below. The default anchor alignment is zero (middle-center).

Sprite Anchors

7	8	9
4	0/5	6
1	2	3

To the right are two examples. Assume this is a 32x32 sprite. In the upper example, the anchor point is at 8, the top-center. Considering the origin at the bottom-left, this sprite is drawn at 16,32, the midpoint of the top of the square.



In the lower example, the anchor point is at 0; and this sprite is drawn at 16,16 (the middle 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 bottom-center).

7 Sound

Function 8,4 queues a sound. Queued sounds are played sequentially, each after the previous has completed, such that sounds within a channel queue will not conflict, interrupt, or overlap.

Frequency is in units of Hertz. Duration is in units of 100ths of a second. Slide is a gradual linear change in frequency, in units of Hz per 100th of a second. Sound target type 0 is the beeper. Currently, the beeper is the only available sound target.

Queue Sound Parameters

P0	P1	P2	P3	P4	P5	P6	P7
$FF04	$FF05	$FF06	$FF07	$FF08	$FF09	$FF0A	$FF0B
Channel	Freq hi	Freq lo	Duration lo	Duration hi	Slide lo	Slide hi	Target

Function 8,5 plays a sound effect immediately. These will be synthesised to the best ability of the available hardware. These are predefined as :

Number	Effect
0	positive
1	negative
2	error
3	confirm
4	reject
5	sweep
6	coin
7	las70
8	powerup
9	victory
10	defeat
11	fanfare
12	alarm 1
13	alarm 2
14	alarm 3
15	ringtone 1
16	ringtone 2
17	ringtone 3
18	danger
19	expl100
20	expl50
21	expl20
22	las30
23	las10