Basic Reference

Binary Operators

Precedence	Operator	Notes
4	*	
	/	Forward slash is floating point divide. 22/7 is 3.142857
	\	Backward slash is integer divide, 22/7 is 3
	%	Modulus of integer division ignoring signs
	>>	Logical shifts up to 32 places, inserting zeros at the appropriate ends.
	<<	
3	+	
	-	
2	<	Compares as numbers or strings. If either is floating point it is compared as such, and the match is not exactly equal, but about 1 part in 100,000. Returns -1 for true, 0 for false.
	<=	
	>	
	>=	
	<>	
	=	
1	&	Binary operators on integers, but can be used as logical operators. Equivalent to and, or and exclusive or.
	|	
	^	

Unary Operators (General)

Operator	Notes
alloc(n)	Allocate n bytes of 65C02 memory, return adress
analog(n)	Read voltage level on pin n – returns a value from 0 to 4095
asc(s$)	Return ASCII value of first character or zero for empty string
atan(n)	Arctangent of n in degrees
chr$(n)	Convert ASCII to string
cos(n)	Cosine of n, n Is in degrees.
deek(a)	Read word value at a
err	Current error number
erl	Current error line number
event(v,r)	event takes an integer variable and a fire rate (r) in 1/100s, and uses the integer variable to return -1 at that rate. If the value in ‘v’ is zero, it resets (if you pause say), if the value in v is -1 the timer will not fire – to unfreeze, set it to zero and it will resynchronise.
exp(n)	e to the power n
false	Return constant 0, improves boolean readability
havemouse()	Return non zero if a mouse is connected.
himem	First byte after end of memory – the stack is allocated below here, and string memory below that.
inkey$()	Return the key stroke if one is in the keyboard buffer, otherwise returns a n empty string.
idevice(device)	Returns true if i2c device present.
iread(device,register)	Read byte from I2C Device Register
instr(str$,search$)	Returns the first position of search$ in str$, indexed from 1. Returns zero if not found.
int(n)	Whole part of the float value n. Integers are unchanged.
isval(s$)	Converts string to number, returns -1 if okay, 0 if fails.
joycount()	Read the number of attached joypads, not including keyboard

	emulation of one.
joypad([index],dx,dy)	Reads the current joypad. The return value has bit 0 set if A is pressed, bit 1 set if B is pressed. Values -1,0 or 1 are placed into dx,dy representing movement on the D-Pad. If there is no gamepad plugged in (at the time of writing it doesn't work) the key equivalents are WASDOP and the cursor keys. If [index] is provided it is a specific joypad (from 1,0 is the keyboard), otherwise it is a composite of all of them.
key(n)	Return the state of the given key. The key is the USB HID key scan code.
left$(a$,n)	Left most n characters of a$
len(a$)	Return length of string in characters.
locale a$	Sets the locale to the 2 character country code a$ e.g. locale "de"
log(n)	Natural Logarithm (e.g. ln2) of n.
lower$(a$)	Convert a string to lower case
max(a,b)	Return the largest of a and b (numbers or strings)
mid$(a$,f[,s])	Characters from a$ starting at f (1 indexed), s characters, s is optional and defaults to the rest of the line.
min(a,b)	Return the smaller of a and b (numbers or strings)
mos(command)	Like the mos command, but returns an non zero error code if the command caused an error.
mouse(x,y[,scroll])	Reads the mouse. The return value indicates button state (bit 0 left, bit 1 right), and the mouse position and also the scrolling wheel position are updated into the given variables.
notes(c)	Return the number of notes outstanding on channel c including the one currently playing – so will be zero when the channel goes silent.
page	Return the address of the program base (e.g. the variable table)
peek(a)	Read byte value at a
pin(n)	Return value on UEXT pin n if input, output latch value if output.
point(x,y)	Read the screen pixel at coordinates x,y. This is graphics data only.
rand(n)	Random integer 0 < x < n (e.g. 0 to n-1)

right$(a$,n)	Rightmost n characters of a$
rnd(n)	Random number 0 < x < 1, ignores n.
sin(n)	Sine of n, n Is in degrees.
spc(n)	Returns a string of n spaces.
spoint(x,y)	Reads the colour index on the sprite layer. 0 is transparency
sqr(n)	Square root of n
str$(n)	Convert n to a string
tab(n)	Advance to screen column n if not past it already.
tan(n)	Tangent of n, n Is in degrees.
true	Return constant -1, improves boolean readability
time()	Return time since power on in 100th of a seconds.
uhasdata()	Return true if there is data in the UART Receive buffer.
upper$(a$)	Convert a string to upper case
val(s$)	Convert string to number. Error if bad number.
vblanks()	Return the number of vblanks since power on. This is updated at the start of the vblank period.

BASIC Commands (General)

Command	Notes
‘ <string>	Comment. This is a string for syntactic consistency. The tokeniser will process a line that doesn’t have speech marks as this is not common. REM this is a comment is now ‘ “this is a comment” and can be typed in as ‘ this is a comment
assert <expr>[,<msg>]	Error generated if <expr> is zero, with optional message.
call <name>(p1,p2,p3)	Call named procedure with optional parameters.
cat [<pattern>]	Show contents of current directory, can take an optional string which only displays filenames containing those characters, so cat “ac” only displays files with the sequence ac in them.
clear [<address>]	Clear out stack, strings, reset all variables. If an address is provided then memory above that will not be touched by BASIC. Note because this resets the stack, it cannot be done in a loop, subroutine or procedure – they will be forgotten. Also clears the sprites and the sprite layer.
cls	Clear the graphics screen to current background colour. This does not clear sprites.
cursor <x>,<y>	Set the text cursor position
data <const>,….	DATA statement. For syntactic consistency, strings must be enclosed in quote marks e.g. data “John Smith”.
defchr ch,….	Define UDG ch (192-255) as a 6x7 font – should be followed by 7 values from 0-63 representing the bit pattern.
delete [<from>][,][<to>]	Delete a line or range of lines in the program.
dim <array>(n,[m]), ...	Dimension a one or two dimension string or number array, up to 255 elements in each dimension (e.g. 0-254)
do … exit … loop	General loop you can break out of at any point.
doke <addr>,<data>	Write word to address
edit	Basic Screen Editor
end	End Program
fkey	Lists the defined function keys
fkey <key>,<string>	Define the behaviour of F1..F10 – the characters in the string are entered as if they are typed (e.g. fkey

	1,chr$(12)+"list"+chr$(13) clears screen and lists the program
for <var> = <start> to/downto <end> … next	For loop. Note this is non standard,Limitations are : the index must be an integer. Step can only be 1 (to) or -1 (downto). Next does not specify an index and cannot be used to terminate loops using the 'wrong' index.
gload <filename>	Load filename into graphics memory.
gosub <expr>	Call subroutine at line number. For porting only. See goto.
goto <expr>	Transfer execution to line number. For porting only. Use in general coding is a capital offence. If I write RENUMBER it will not support these.
if <expr> then ….	Standard BASIC if, executes command or line number. (IF .. GOTO doesn't work, use IF .. THEN nn)
if <expr>: .. else .. endif	Extended multiline if, without THEN. The else clause is optional.
ink fgr[,bgr]	Set the ink foreground and optionally background for the console.
input <stuff>	Input has an identical syntax and behaviour to Print except that variables are entered via the keyboard rather than printed.
ireceive <d>,<a>,<s> itransmit <d>,<a>,<s>	Send or receive bytes starting at a, count s to or from device d.
isend <device>,<data>	Send data to i2c <device> ; this is comma seperated data, numbers or strings. If a semicolon is used as a seperator e.g. 4137; then the constant is sent as a 16 bit value.
iwrite <dev>,<reg>,<b>	Write byte to I2C Device Register
let <var> = <expr>	Assignment statement. The LET is optional.
library [<from>][,][<to>]	Librarise / Unlibrarise code.
list [<from>][,][<to>] list <procedure>()	List program to display by line number or procedure name.
load "file"[,<address>]	Load file to BASIC space or given address.
local <var>,<var>	Local variables, use after PROC, restored at ENDPROC variables can be simple strings or numbers *only*.
mon	Enter the machine code monitor
mos <command>	Execute MOS command.

mouse cursor <n>	Select mouse cursor <n> [0 is the default hand pointer]
mouse show|hide	Make mouse cursor visible/invisible
mouse to <x>,<y>	Position mouse cursor
new	Erase Program
old	Undoes a new. This can fail depending on what has been done since the ‘new’.
on error <code>	Install an error handler that is called when an error occurs. Effectively this is doing a GOTO that code, so recovery is dependent on what you actually do.
palette c,r,g,b	Set colour c to r,g,b values – these are all 0-255 however it is actually 3:2:3 colour, so they will be approximations.
palette clear	Reset palette to default
pin <pin>,<value>	Set UEXT <pin> to given value.
pin <pin> input|output|analog	Set UEXT<pin> direction (default is input)
poke <addr>,<data>	Write byte to address
print <stuff>	Print strings and numbers, standard format - , is used for tab ; to seperate elements.
proc <n>([ref] p1,p2,…) endproc	Delimits procedures, optional parameters, must match call. Parameters can be defined as reference parameters and will return values. Parameters cannot be arrays.
read <var>,…	Read variables from data statements. Types must match those in data statements.
renumber [<start>]	Renumber the program from start, or from 1000 by default. This does *not* handle GOTO and GOSUB. Use those, you are on your own.
repeat .. until <expr>	Execute code until <expr> is true
restore	Restore data pointer to program start
return	Return from subroutine called with gosub.
run	Run Program
run “<program>”	Load & Run program.
save “file”[,<adr>,<sz>]	Save BASIC program or memory from <adr> length <sz>

sreceive <a>,<s> stransmit <a>,<s>	Send or receive bytes starting at a, count s to SPI device
ssend <data>	Send data to SPI device ; this is comma seperated data, numbers or strings. If a semicolon is used as a seperator e.g. 4137; then the constant is sent as a 16 bit value.
stop	Halt program with error
sweet <address>	Run Sweet 16 code using the registers at the given address ; the registers are a 32 byte block of memory, and can be accessed easily using the [] operators.
sys <address>	Call 65C02 machine code at given address. Passes contents of variables A,X,Y in those registers.
tilemap addr,x,y	Define a tilemap. The tilemap data format is in the API. The tilemap is stored in memory at addr, and the offset into the tilemap is x,y in pixels (1 tile = 16 pixels).
uconfig <baud>[,<prt>]	Set the baud rate and protocol for the UART. Currently only 8N1 is supported.
ureceive <d>,<a>,<s> utransmit <d>,<a>,<s>	Send or receive bytes to/from the UART starting at a, count s
isend <device>,<data>	Send data to UART ; this is comma seperated data, numbers or strings. If a semicolon is used as a seperator e.g. 4137; then the constant is sent as a 16 bit value.
wait <cs>	Waits for <cs> hundredths of a second
while <expr> .. wend	Repeat code while expression is true
who	Display contributors list.

The Inline Assembler

The inline assembler works in a very similar way to that of the BBC Micro, except that it does not use the square brackets [and] to delimit assembler code. Assembler code is in normal BASIC programs.

A simple example shown below (in the samples directory) :

It prints a row of 10 asterisks.

100	**mem = alloc(32)**	Allocate 32 bytes of memory to store the program code.
110	**for i = 0 to 1**	We pass through the code twice because of forward referenced labels. This actually doesn't apply here.
120	**p = mem**	P is the code pointer – it is like * = <xx> - it means put the code here
130	**o = i * 3**	Bit 0 is the pass (0 or 1) Bit 1 should display the code generated on pass 2 only, this is stored in 'O' for options.
140	**.start**	Superfluous – creates a label 'start' – which contains the address here
150	**ldx #10**	Use X to count the starts
160	**.loop1**	Loop position. We can't use loop because it's a keyword
170	**lda #42**	ASCII code for asterisk
180	**jsr $fff1**	Monitor instruction to print a character
190	**dex**	Classic 6502 loop
200	**bne loop1**	
210	**rts**	Return to caller
220	**next**	Do it twice and complete both passes
230	**sys mem**	BASIC instruction to 'call 6502 code'. Could do sys start here.

Most standard 65C02 syntax is supported, except currently you cannot use lsr a ; it has to be just lsr (and similarly for rol, asl, ror,inc and dec)

You can also pass A X Y as variables. So you could delete line 150 and run it with

X = 12: sys start

which would print 12 asterisks.

[ ] Operator

The [] operator is used like an array, but it is actually a syntactic equivalent of deek and doke, e.g. reading and writing 16 bytes. mem[x] means the 16 bit value in mem + x * 2, so if mem = 813 then mem[2] = -1 writes a 16 bit word to 817 and 818, and print mem[2] reads it. The index can only be from 0..127

The purpose of this is to provide a clean readable interface to data in 65C02, Sweet16 and other programs running under assembly language ; often accessing elements in the ‘array’ as a structure.

Zero Page Usage

Neo6502 is a clean machine, rather like the Sharp machines in the 1980s. When BASIC is not running it has no effect on anything, nor does the firmware. It is not like a Commodore 64 (for example) where changing some zero page locations can cause crashes.

However, BASIC does make use of zero page. At the time of writing this is memory locations $10-$41.

These can however be used in machine code programs called via SYS. Only 4 bytes of that usage is system critical (the line pointer and the stack pointer), those are saved on the stack by SYS, so even if you overwrite them it does not matter.

However, you can’t use this range to store intermediate values *between* sys calls. It is advised that you work usage backwards from $FF (as BASIC is developed forwards from $10). It is very unlikely that these will meet in the middle.

$00 and $01 are used on BASIC boot (and maybe other languages later) but this should not affect anything.

Basic Commands (Graphics)

The graphics commands are MOVE, PLOT (draws a pixel), LINE (draws a line) RECT (draws a rectangle) ELLIPSE (draws a circle or ellipse) IMAGE (draws a sprite or tile), TILEDRAW (draws a tilemap) and TEXT (draws text)

The keywords are followed by a sequence of modifiers and commands which do various things as listed below

FROM x,y	Sets the origin position, can be repeated and optional.
TO x,y	Draw the element at x,y or between the current position and x,y depending on the command. So you could have **text “Hello” to 10,10** or **rect 0,0 to 100,50**
BY x,y	Same as to but x and y are an offset from the current position
X,y	Set the current position without doing the action
INK c	Draw in solid colour c
INK a,x	Draw by anding the colour with a, and xoring it with x.
SOLID	Fill in rectangles and ellipses. For images and text, forces black background.
FRAME	Just draw the outline of rectangles and ellipses
DIM n	Set the scaling to n (for TEXT, IMAGE, TILEMAP only), so **text “Hello” dim 2 to 10,10 to 10,100** will draw it twice double size. Tiles can only be 1 or 2 (when 2, tiles are drawn double size giving a 32x32 tile map)

These can be arbitrarily chained together so you can do (say) LINE 0,0 TO 100,10 TO 120,120 TO 0,0 to draw an outline triangle. You can also switch drawing type in mid command, though I probably wouldn’t recommend it for clarity.

State is remember until you clear the screen so if you do INK 2 in a graphics command things will be done in colour 2 (green) until finished.

TEXT is followed by one parameter, which is the text to be printed, these too can be repeated e,g, TEXT “Hello” TO 10,10 TEXT “Goodbye” DIM 2 fTO 100,10

IMAGE is followed by two parameters, one specifies the image, the second the ‘flip’. These can be repeated as for TEXT.

The image parameter is 0-127 for the first 128 tiles, 128-191 for the first 64 16x16 sprites and 192-255 for the first 64 32x32 sprites. The flip parameter, which is optional, is 0 (no flip) 1 (horizontal flip) 2 (vertical flip) 3 (both).

An example would be image 4 dim 2 to 10,10 image 192,3 dim 1 to 200,10

Note that images are *not* sprites or tiles, they use the image to draw on the screen in the same way that LINE etc. do.

Sprite Commands

Sprite commands closely resemble the graphics commands.

They begin with SPRITE <n> which sets the working sprite. Options include IMAGE <n> which sets the image, TO <x>,<y> which sets the position, FLIP <n> which sets the flip to a number (bit 0 is horizontal flip, bit 1 is vertical flip), ANCHOR <n> which sets the anchor point and BY <x>,<y> which sets the position by offset.

With respect to the latter, this is the position from the TO and is used to do attached sprites e.g. you might write.

SPRITE 1 IMAGE 2 TO 200,200 SPRITE 2 IMAGE 3 BY 10,10

Which will draw Sprite 1 and 200,200 and sprite 2 offset at 210,210. It does not offset a sprite from its current position.

As with Graphics these are not all required. It only changes what you specify not all elements are required each time *SPRITE 1 IMAGE 3* is fine.

SPRITE can also take the single command CLEAR ; this resets all sprites and removes them from the display

Sprite 0 is used for the turtle sprite, so if the turtle is turned on, then it will adjust its graphic, size to reflect the turtle position. If turtle graphics are not used, it can be used like any other.

Implementation notes

Up to 128 sprites are supported. However, sprite drawing is done by the Pico and is not hardware, so more sprites means the system will run slower.

Additionally, the sprites are currently done with XOR drawing, which causes effects when they overlap. This should not be relied on (it may be replaced by a clear/invalidate system at some point), but the actual implementation should not change.

This is an initial sprite implementation and is quite limited.

(The plan is to add a feature like the animation languages on STOS and AMOS which effectively run a background script on a sprite)

Sprite Support

=spritex(n) =spritey(n)

These return the x and y coordinates of the sprites draw position (currently the centre) respectively.

= hit(sprite1,sprite2,distance)

The hit function is designed to do sprite collision. It returns true if the pixel distance between the centre of sprite 1 and the centre of sprite 2 is less than or equal to the distance.

So if you wanted to move a sprite until it collided with another sprite, assuming both are 32x32, the collision distance would be 32 (the distance from the centre to the edge of both sprites added together), so you could write something like :

```
x = 0
repeat
     x = x + 1: sprite 1 to x,40
until hit(1,2,32)
```

In my experience of this the distance needs to be checked experimentally, as it affects the ‘feel’ of the game ; sometimes you want near exact collision, sometimes it’s about getting the correct feel. It also depends on the shape and size of the sprites, and how they move.

I think it’s better than a simple box collision test, and more practical than a pixel based collision test which is very processor heavy.

Sound Commands

The Neo6502 has one sound channel by default, which is a beeper. This is channel 0.

Sound

The main sound command is called “sound” and has the following forms.

Sound clear

Resets the entire sound system, silences all channels, empties all queues

Sound <channel> clear

Resets a single channel ; silences it, and empties its queue

Sound <channel>,<frequency>,<time>[,<slide>]

Queues a note on the given channel of the given frequency (in Hz) and time (in centiseconds). These will be played in the background as other notes finish so you can ‘queue up’ an entire phrase and let it play by itself. The slide value adds that much to the frequency every centisecond allowing some additional effects (note, done in 50Hz ticks)

A mixture of the two syntaxes SOUND 0 CLEAR 440,200 is now supported.

Sfx

Sfx plays sound effects. Sound effects are played immediately as they are usually in response to an event.

It’s format is ***sfx <channel>,<effect***> .

Screen Editor

The ***edit*** command starts the screen editor. This currently supports left, right, home, end, backspace and delete on the current line, enter, up, down, page up and page down to change lines.

Esc exits the editor, and Ctrl+P and Ctrl+Q insert and delete a whole line.

Note the editor is slightly eccentric ; it is not a text editor, what it is doing is editing the underlying program on the fly – much the same as if you were typing lines in.

The editor uses line numbers, so is *not* compatible with their use in programs. Any program will be renumbered from 1 upwards in steps of 1 (except library routines).

You shouldn’t be using line numbers anyway !

Graphic Data

The graphic date for a game is stored in what is named by default “graphics.gfx”. This contains up to 256 graphics objects, in one of three types. One can have multiple graphics files.

Each has 15 colours (for sprites, one is allocated to transparency) which are the same as the standard palette.

16x16 tiles (0-127, $00-$7F)

These are 128 16x16 pixel solid tiles which can be horizontally flipped

16x16 sprites (128-191, $80-$BF)

These are 64 16x16 sprites which can be horizontally and/or vertically flipped

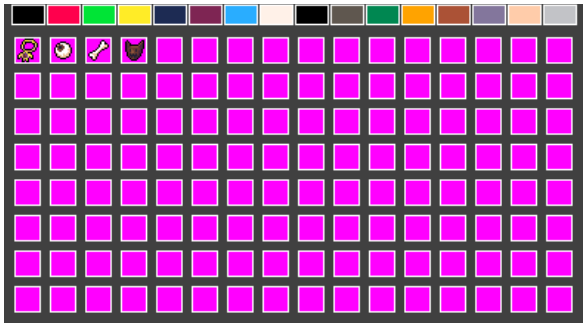
32x32 sprites (192-255, $C0-$FF)

These are 64 32x32 sprites which can be horizontally and/or vertically flipped

These are created using two scripts, which are written in Python and require the installation of the Python Imaging Library, also known as PIL or Pillow.

Empty graphics files

The script “createblanks.zip” creates three files, tile_16.png, sprite_16.png and sprite_32.png which are used for the three types of graphic.



The picture is of sprite_16 though the others look very similar. The palette is shown at the top (in later versions this will be configurable at this point), and some sample sprites are shown. Each box represents a 16x16 sprite. 32X32 sprite looks the same except the boxes are twice the size and there are half as many per row.

Tiles are almost identical ; in this the background is black. The solid magenta (RGB 255,0,255) is used for transparency, this colour is not in the palette.

Running createblanks.zip creates these three empty files. To protect against accidents it will not overwrite currently existing files, so if you want to start again then you have to delete the current ones.

Compiling graphics files

There is a second script “makeimg.zip”. This converts these three files into a file “graphics.gfx” which contains all the graphic data.

This can be loaded into graphics image memory using the gload command, and the address 65535 e,g, ***gload “graphics.gfx”***

There is an example of this process in the repository under basic/images which is used to create graphic for the sprite demonstation program

Libraries

Libraries are part of the BASIC program, placed at the start. However, their line numbers are all set to zero. (So you cannot use GOTO or GOSUB in libraries, but you should only use them for porting old code).

NEW will not remove them, LIST does not show them (except LIST 0), You cannot edit them using the line editors.

However RUN does not skip them. This is so you can have initaialisation code e.g.

```
<do initialisation code>
if false
proc myhw.dosomething(p1) ….
proc myhw.panic()
endif
```

To support this, there is a LIBRARY command. LIBRARY on its own undoes the library functionality. It renumbers the whole program from the start, starting from line number 1000.

Otherwise LIBRARY works like LIST. You can do LIBRARY <line> or LIBRARY <from>,<to> and similar. Instead of listing this code, it causes them to “vanish” by setting their line numbers to zero.

They are also supported in the makebasic script. Adding the command library makes all code so far library code.

e.g.

```
python makebasic.zip mylib.bsc library mainprogram.bsc
```

Turtle Graphics

The Neo6502 has a built in turtle graphics system. This uses sprite $7F as the turtle, which it will take over :)

The following commands are supported.

Command	Purpose
forward <n>	Move turtle forward n (pixel distance)
left <n> right <n>	Rotate turtle at current position
penup	Stop drawing
pendown	Start drawing
pendown <n>	Start drawing in given colour
turtle home	Reset turtle to home position
turtle hide	Hide the turtle
turtle fast	The turtle is deliberately slowed to give it an animated feel so you can see the drawing, this is because it's primary purpose is educational. This makes it go full speed.

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.

Memory Map

Note *all this is actually RAM* and functions as it, except for the command area ; writing non zero values into the Command byte may affect other locations in that 16 byte area or elsewhere (e.g. reading a file in).

This block is however moveable.

It is also possible that the top of free memory will move down from $FC00.

Addresses	Contents
0000-FBFF	Free memory. Not used for anything.
FC00-FEFF	Kernel Image. Contains system functions and WozMon, which it currently boots into. This is RAM like everything else.
FFF0-FF0F	Command, Error, Parameters, Information space. This can be moved to accommodate other systems.
FF10-FFF9	Vectors to Kernel routines, not actually mandatory either.
FFFA-FFFF	65C02 Vectors for NMI, IRQ and Reset. This one is mandatory :)