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
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
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. See samples.
exp(n)	e to the power n
false	Return constant 0, improves boolean readability
inkey$()	Return the key stroke if one is in the keyboard buffer, otherwise returns a n empty string.
int(n)	Whole part of the float value n. Integers are unchanged.
isval(s$)	Converts string to number, returns -1 if okay, 0 if fails.
joypad(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. Currently this is the keyboard keys Z(left) X(right) K(up) M(down) L(A) ; (B)
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$)	Return length of string in characters.
log(n)	Natural Logarithm (e.g. ln2) of n.
max(a,b)	Return the largest of a and b (numbers or strings)
mid$(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)
page	Return the address of the program base (e.g. the variable table)

peek(a)	Read byte value at a
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.
sqr(n)	Square root of n
str$(n)	Convert n to a string
tan(n)	Tangent of n, n Is in degrees.
true	Return constant -1, improves boolean readability
time()	Return time since power on in 100th of a seconds.
val(s$)	Convert string to number. Error if bad number.

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>	Error generated if <expr> is zero. Used for checking parameters and/or enforcing contracts.
call <name>(p1,p2,p3)	Call named procedure with optional parameters.
cat	Show contents of current directory
clear	Clear out stack, strings, reset all variables.
cls	Clear screen to current background colour.
data <const>,….	DATA statement. For syntactic consistency, strings must be enclosed in quote marks e.g. data “John Smith”.
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
end	End Program
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.
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 on the screen.
let <var> = <expr>	Assignment statement. The LET is optional.
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*.
new	Erase Program
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.
poke <addr>,<data>	Write byte to address
print <stuff>	Print strings and numbers, standard format - , is used for tab ; to seperate elements.
proc <nm>(p1,p2,p3 …) endproc	Delimits procedures, optional parameters, must match call.
read <var>,…	Read variables from data statements. Types must match those in data statements.
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
save “file”[,<adr>,<sz>]	Save BASIC program or memory from <adr> length <sz>
stop	Halt program with error
sys <address>	Call 65C02 machine code at given address. Passes contents of variables A,X,Y in those registers.
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.

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) and TEXT (draws text)

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

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
FRAME	Just draw the outline of rectangles and ellipses
DIM n	Set the scaling to n (for TEXT only), so **text “Hello” dim 2 to 10,10 to 10,100** will draw it twice double size

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.

The default colours are below. There are 16 colours in a 256 colour screen because it uses the palette system to create to layers, one for tiles and graphics and one for sprites.

0	Black (Transparent for sprites)
1	Red
2	Green
3	Yellow
4	Blue
5	Purple
6	Cyan
7	White
8	Black (always)
9	Dark Grey
10	Dark Green
11	Orange
12	Brown
13	Lavendar
14	Light-Peach
15	Light-Grey

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) 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

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.

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

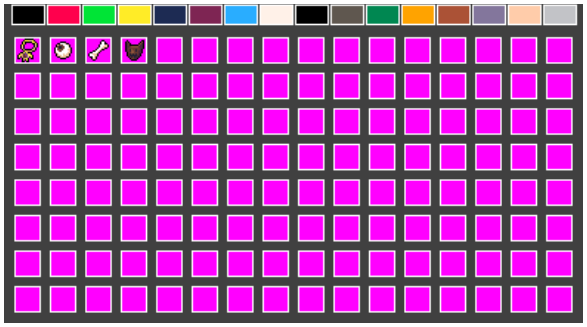
32x32 sprites (192-255, $C0-$FF)

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 load command, and the address 65535 e,g, ***load “graphics.gfx”,$FFFF***

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