VICScript v0.1 User Manual

Bobbi Webber-Manners
bobbi.manners@gmail.com

Introduction

VICScript is a simple scripting language for the Commodore VIC-20 with 32K expansion and disk drive[1]. VICScript includes a minimal line editor which may be used for editing program text (or editing short text files.) Alternatively files may be prepared in an external editor and saved as plain PETSCII files.

VICScript is primarily intended to be a fun simple language for twiddling with hardware. The only native types are 16 bit integers and arrays of 16 bit integers. A full range of C-style operations for logical and bitwise manipulation of values is provided. Constants may be expressed in decimal or hex and values may also be displayed in either base. VICScript also aims to provide a basic set of structured programming primitives, such as named subroutines, multi-line **`if`** / **`else`** / **`endif`** conditionals and a flexible **`while`** loop. The flavour of the language may be described as a cross between C and BASIC.

The VICScript prototype implementation is written in the C programming language and compiled using the **`cc65`** compiler under Linux. The resulting code is rather large[2] and slow when compared to hand-tuned assembly code. As a future project I may rewrite some or all of VICScript directly in assembly. Like BASIC, VICScript is an interpreted language. However, unlike CBM BASIC, the current implementation of VICScript does not tokenize the program text, which makes parsing slower than it otherwise would be. The upside of this design decision is that the VICScript line editor may be used to edit any type of plain PETSCII text file and conversely VICScript program text may be composed in any editor that saves plain PETSCII text files.

Warning: The VICScript language may change without notice! Future versions may or may not be backwards compatible!

1 VICScript has been tested on a real VIC-20 with an SD2IEC type drive and under VICE emulation. It should work with a genuine Commodore 1541 or 1571 drive, but this has not been confirmed.
2 The code is around 15KB at the time of writing. It could probably be crammed into around half this amount if hand written in assembly.

Quick Start

This section is intended to give a quick flavour of VICScript.

To start VICScript, load it from floppy disk and then start it executing using the following CBM BASIC commands:

```
LOAD "VICSCRIPT", 8
RUN
```

The VICScript sign on message will be shown confirming that VICScript is running. The flashing cursor indicates that it is ready to accept input in immediate mode. VICScript operates in mixed-case mode by default and commands are normally entered in lower case.

First, try a simple mathematical expression using integer values:

```
println 3 * (5 + 2)
21
```

Logical and bitwise operators are supported in the same manner as in C. For example a logical and may be expressed:

```
println 1 && 0
0
```

or a left shift as follows:

```
println 1 << 4
16
```

Variables must be declared before use. This statement:

```
int foo ← 1
```

declares the 16 bit integer variable foo and initializes it to 1. The ← character is the special CBM left-arrow character (which maps to underscore in ASCII.) Once declared, variables may be freely assigned and used in expressions:

```
print foo * 100
foo ← foo + 1
```

The currently defined variables may be viewed using the following command:

```
vars
```

The command:

```
clear
```

will erase all defined variables and free the memory.

Our First VICScript Program

First ensure there are no program statements in the editor buffer:

```
new
```

Then enter the command to start inserting text into the empty buffer:

```
:i0
```

A little explanation is due. All editor commands begin with the colon character. The character after the colon signifies the command (“insert” in this case) and the number is the line number of the buffer to apply the editing command. Putting it all together, the command above tells VICScript to begin inserting text into an empty buffer (at line zero.)

Once in insert mode, the VICScript prompt changes to a reverse video green '>' character. This serves to remind the programmer that she is editing program text, not typing VICScript commands in immediate mode. Once in insert mode lines may be entered one after another. To stop entering program lines, enter the period character '.' by itself. Enter the following program:

```
:i0                          Editor command to insert text
'Test program                This is a comment
int i ← 0
for i ← 1 : 10
 prstr "i:"                  prstr prints a string literal, no CR
 print i                     print prints an integer, no CR
 prstr ", sq:"
 println i * i               println prints an integer, with CR
endfor
end
.                            Not part of the program, go back to
                             immediate mode
```

To check the program, you may list it as follows:

```
:l                           Lower case letter L
```

If there are errors, you may use the Commodore screen editor to modify the code, in a similar way to CBM BASIC. Be sure to hit RETURN to accept any modified line. Other editing commands include **`:d`** (delete line(s)), **`:a`** (append lines).

When editing or listing a program, line numbers are displayed in reverse video red before each line of program text. Unlike BASIC, however, VICScript does not have line numbers. The numbers shown are used by the line editor only in order to navigate the file. They are not stored on disk with the file contents. This means that whenever you insert lines into the middle of a program, the line numbers for the program text after the insertation point will increase[3].

Once you are satisfied, you may save the program text to disk using the write command as follows:

```
:w"testprog"
```

Unit 8 is hard-coded. If the file already exists, it may be overwritten using the normal CBM overwrite syntax:

```
:w"@:testprog"
```

Previously-saved programs may be loaded using the read command:

```
:r"testprog"
```

Finally, to run our example program:

```
run
```

I hope this quick start has given you a feel for what VICScript can do. The remainder of this manual provides a detailed reference for all commands.

3 If you list the program and then insert some lines, is is a good idea to list the modified region using the `:l` command before performing further edits to see the correct line numbers.


: i0
'Test program
int i←0
for i←1:10
prstr "i:"
print i
prstr ", sq:"
println i*i
endfor
end
.
run
i:1, sq:1
i:2, sq:4
i:3, sq:9
i:4, sq:16
i:5, sq:25
i:6, sq:36
i:7, sq:49
i:8, sq:64
i:9, sq:81
i:10, sq:100

Line Editor

The line editor is very loosely modelled after UNIX line editors such as **`ed`** or **`ex`**. Because the program is not tokenized the file that is stored on disk is a plain PETSCII[4] text file. The line editor may be used for editing small files for any purpose – not just VICScript programs!

When entering text into the editor, the prompt changes to a green reverse video '>' character to remind the user that she is inserting program text, not entering VICScript commands in immediate mode.

Read

`:r"filename"`

Load a Vicscript program (or other PETSII file) from the SEQ file 'filename'. Unit #8 is hard-coded.

Write

`:w"filename"`

Save a Vicscript program into the SEQ file 'filename'. Unit #8 is hard-coded. If you wish to overwrite an existing file, prefix the filename with '@:' (note though that this is not safe with original Commodore floppy drives). **After saving, check the error light on your drive.** If it is not flashing, then the save was OK.

List

`:l[`*`expr1`*`[,`*`expr2`*`]]` (Lower case letter L)

Shows the stored program lines from *expr1* to *expr2* inclusive. Line numbers are shown alongside the listing. Unlike BASIC, these are *not* part of the code. These sequential numbers are used for line editor commands only. If *expr2* is not provided, lists to the end. If *expr1* is not provided lists the entire file.

Examples:

`:l`	List whole file
`:l10`	List line 10 to end
`:l10,20`	List lines 10 to 20
`:l10,10+15`	List lines 10 to 25
`:la,b`	List lines from value of variable a to variable b

Insert

`:i`*`expr`*

Insert one or more lines of program text before line *expr*. Enter a period on its own to stop

4 Ignoring the famous graphical characters provided in PETSCII, the main difference between PETSCII and ASCII is that the upper and lower case characters are swapped. VICScript programs use lower case characters, but if VICScript source is edited on another platform, the characters will appear as upper case.

inserting lines and return to immediate mode.

Examples:

`:i0`	Start inserting at beginning of buffer (or empty buffer)
`:i99`	Start inserting before line 99
`:iline`	Start inserting before line number in variable line

Append

`:a`***`expr`***

Append one or more lines of program text after line *expr*. Enter a period on its own to stop appending lines and return to immediate mode.

Examples:

`:a99`	Start appending after line 99
`:aline`	Start appending after line number in variable line

Delete

`:d`***`expr1`***`[,`***`expr2`***`]`

Delete lines from *expr1* to *expr2*. If *expr2* is not provided deletes the single line referenced by *expr1*.

Examples:

`:d37`	Delete line 37
`:dbad`	Delete the line number in variable bad

Change

`:c`***`exprreplacementtext`***

Replace line number *expr* with the text that appears after the expression. This command is not usually used directly, but is automatically invoked if a line begins with a digit. This allows the CBM screen editor to be used the same way as in CBM BASIC by listing the code with and then editing on screen. Be sure to hit Return to accept any modifications made in this way.

Example:

`:c33 prstrln "Fixed it!"`	Change line 33 using explicit ‘:c’ command

VICScript Language Reference

General

White space is ignored, so you can indent your source however you please.

Multiple statements are permitted on a line, separated by semicolons (`;`).

Comments are introduced with the single quote character (`'`). The comment extends to the end of the current program line.

Variables

Only 16 bit signed ints and one-dimensional arrays of 16 bit signed ints are currently supported.

The first four characters of the name are significant, so **`variable`** and **`variety`** refer to the same object.

Variables must be declared before use or an error is reported. They do not 'autovivify' as in BASIC.

Integer variables are declared as follows:

```
int name ← expr
```

The expression on the right hand side is evaluated and used as an initializer for the new integer variable.

Integer arrays are declared like this:

```
int name[expr1] ← expr2
```

The expression *expr1* is evaluated and used as the size of the new array. Expression *expr2* is evaluated and all elements of the new array are initialized to this value. VICScript array indices are zero-based so that an array of size *N* has valid indices from 0 to *N*-1. Accesses are bounds checked.

Examples:

```
int aa ← 42
int qwerty ← 23 * 2
```

```
int foobar[aa*2] ← 0
```

Once, declared variables may be used in any expression.

Examples:
```
aa ← qwerty << 2
i ← i + 1
foobar[idx] ← foobar[idx + 1]
```

Constants / Literals

Integer constants may be expressed in decimal, or in hexadecimal with a $ prefix.

String literals are enclosed in double quotes.

Examples:
```
int theanswer ← 42
int v ← $ff
prstrln “Hello world!”
```

Expressions

The same symbols are used as in C, where available in PETSCII. However the following substitutions are performed:

VIC-20	Conventional C Notation
#	|
##	||
@	^
.	~

While not all C operators are supported, the arithmetic, logical and bitwise operators are provided. The asterix pointer dereference operator is implemented and may be used to read (‘peek’) or write (‘poke’) memory. The operator precedence order is the same as that of C:

Level 11 (highest precedence):	
-	unary minus
+	unary plus
*	unary dereference operator

`!` `.`	unary logical not unary bitwise not (`~` in C)

Level 10:
`^`
`*`
`/`
`%`
	
	exponent
	multiply
	divide
	modulus

Level 9:
`+`
`-`
	
	add
	subtract

Level 8:
`<<`
`>>`
	
	left shift
	right shift

Level 7:
`>`
`>=`
`<`
`<=`
	
	greater than
	greater than or equal
	less than
	less than or equal

Level 6:
`==`
`!=`
	
	equality
	inequality

Level 5: `&`	 bitwise AND
Level 4: `@`	 Bitwise XOR (`^` in C)
Level 3: `#`	 Bitwise OR (`|` in C)
Level 2: `&&`	 Logical AND
Level 1: `##`	 Logical OR

Examples:

`(1 + 3) * 2^3`	1 plus 2, times 2 to power of 3
`(foo & $0f) >> 8`	Bitwise and of foo and hex $0f, right shift 8 bits
`(.a & b)`	Bitwise not a, then bitwise and with b
`(a ## !b)`	Logical not of b, logical or with a

`*(a + 10)`	Add 10 to value of a, return byte at that address (peek)

`int addr ← $f009`	Declare addr, initialize to hex $f009 (decimal 36879)
`*addr ← 8`	Poke value 8 into address 36879 (VIC register)
`a ← *addr`	Obtain the value stored at address 36879, store in a

Accessing Memory

expr1* ← *expr2 ("POKE")
Evaluates *expr1* and *expr2*. Stores the value of *expr2* in the address corresponding to the value of *expr1*.

val ← *expr1* ("PEEK")
The prefix * operator returns the byte value at the address given by *expr1*.

Program Control

run
Start running the stored program. A running program may be interrupted by hitting the STOP key.

new
Clear stored program.

clear
Clears all variables.

vars
Shows all variables in a table.

free
Shows amount of free heap memory available for code and variables. See the section 'Memory Layout' for more details.

end
Terminates program execution. This statement is sometimes required to avoid having the program terminate with an error by running into a subroutine definition.

quit
Terminates VICScript and returns to CBM BASIC.

Output

`print` ***`expr`***
Evaluates integer expression *expr* and displays on screen in decimal.

`println` ***`expr`***
Evaluates integer expression *expr* and displays on screen in decimal, with carriage return.

`prhex` ***`expr`***
Evaluates integer expression *expr* and displays on screen in hex.

`prhexln` ***`expr`***
Evaluates integer expression and displays on screen in hex, with carriage return.

`prstr "String literal"`
Displays string literal on screen. Commodore control codes for colours, cursor movement etc. are supported the same was as in CBM BASIC.

`prstrln "String literal"`
Displays string literal on screen, with carriage return. Commodore control codes for colours, cursor movement etc. are supported the same was as in CBM BASIC.

Examples:

```
prstr "The answer is "; print (40 + 2); prstrln "..."
prstr "Addr:"; prhex addr; prstr " val:"; prhexln *addr
prstr "[CLEAR][BLU]"
```

Subroutines

Named subroutines may be defined and called. At present, subroutines do not accept parameters or return results to the caller.

`sub sublabel`
Defines a subroutine with name “sublabel”. The label extends to the end of the line.

`call sublabel`
Calls a subroutine with name “sublabel”. The label extends to the end of the line.

`return`
Return from subroutine. The flow of execution returns to the line following the **`call`** instruction.

Example:
```
call foo
prstrln "Done"
end

sub foo
prstrln "This is foo!"
return
```

Flow Control

VICScript provides the following flow control structures:

- Unconditional **`jump`** to named label (**`lbl`**).
- Multi-line **`if`** / **`else`** / **`endif`** conditional.
- **`for`** loops
- **`while`** loops.

Unconditional Jump

`lbl label`
Defines a jump target “label”. The label extends to the end of the line, so it is possible to define labels with spaces if you so wish.

`jump label`
Unconditional jump to label. The label extends to the end of the line.

It is legal to jump out of loops. It is not legal to jump into loops or into or out of subroutines. Undefined behaviour may result.

Conditionals

```
if expr
 ...
endif
```

Evaluates the expression *expr*. If true, continues executing the following statement(s). If false, skips all statements up until the **endif**.

```
if expr
 ...
else
 ...
endif
```

Evaluates the expression *expr*. If true, continues executing the following statement(s) until the **else** then skips to the **endif**. If false, skips to the **else** and executes the statements up until the **endif**.

Example:

```
if i<20
  prstrln "less"
else
 prstrln "more"
 jump abort
endif
end

lbl abort
prstrln "Error"
end
```

Single line conditionals may also be composed as shown in the following example.

Example:

```
if i==1; prstrln "yes"; endif
if i==1; prstrln "yes"; else; prstrln "no"; endif
```

For Loop

```
for var ← expr1 : expr2
 [[stmt];]*
endfor
```

Initiates a **for** loop using loop control variable **var** (which must be defined.)

The loop control variable is initialized to *expr1* and is incremented by one on each iteration. The last loop execution occurs with **var** = *expr2*. The test is done at the end of the loop, not the beginning, so **for** loops always execute at least once.

Single line **for** loops may be composed using semicolons as shown in the following example (the **for** statement must begin the line):

Example:
```
for i ← 1 : 10; println i; endfor
```

While Loop

```
while expr
  [[stmt];]*
endw
```

Initiates a **while** loop. Evaluates *expr*, and if true then enters the loop. When **endw** is encountered jumps back to top of loop and evaluates *expr* again. **while** loops are more flexible than **for** loops but are less efficient in execution.

Single line **while** loops may be composed as shown in the following example (**while** must begin the line):

Example:
```
while i < 10; println i; i ← i + 1; endw
```

Memory Layout

VICScript is designed to run on the VIC-20 with a 32KB RAM expansion. The VICScript executable loads from address $1200 up, and occupies around 15KB of memory taking up BLK0, 1, 2 and some of BLK 3. The remainder of BLK 3 is made available for the storage of variables. All of 8KB of BLK5 is used for storing the VICScript program text.

Example Program – Sieve of Eratosthenes

This is a simple algorithm for determining which integers are prime.

The entire source code fits on one VIC-20 screen:

And running it will give the prime numbers from 2 to **nr**^{2}. In the example above, **nr** is set to 10, so the first 100 primes are computed. The limitation on how high **nr** may be set is available memory for array **A[]** (two bytes per element.)