6502 Basic Outline

Requirements

Free Software

Under the MIT license.

Compatibility

Should run either on a 6502, a 65C02 or similar processors.

Modular design

Modules should be added or removed independently. Communication with modules is through an X dispatch block e.g., jmp (dispatchTable, x), with one entry point for each module, this allows paging etc.

Modern

Should have modern features - Procedures, Local variables, Structures and so on. Three types (int % string $ float #, default variable)

Extensible

Should be extensible with system specific functionality.

Modules

Code for all optional modules is built in but can generated an error and have no actual code. All module functionality is accessed through one function, at a slight cost in speed, so that modules can be paged.

Main

Main evaluation and execution functions, standard integer functions, normal dispatch tables, standard procedures, variables, locals, structures, basic syntax checking routines.

Error

Error handling, which can be localised.

Device

External I/O type functionality, System specific functionality.

Variables

Handle variables, arrays and so on.

FloatingPoint (optional)

Handles all floating-point operations. Code for FP types is built in, but when activated can cause an error.

String (optional)

String functions and manager, handles concrete and temporary strings, garbage collection and so on. The default string handling (Atom style) is built in by default however, though it is length prefixed not ASCIIZ.

Assembler (optional)

65(C)02 inline assembler like the BBC Micro.

Tokenizer (optional)

Line tokenising / detokenising. Contains the text for the keywords, may not be needed for a run-time only system.

Interaction (optional)

Code connected with editing, task switching and so on.

Header

The start of the program

Footer

The end of the program

Extension

Graphics and Sound and other X16 specific stuff

Implementation

All module handlers should use the name of their module, together with a constant which is the name of the module, followed by an underscore followed by the method.

```
e.g., editor_deleteline
```

This should be automatically generated from the files in the module subdirectory, which looks for marks as follows.

```
EditorDeleteLine:   ;; <deleteline>
```

These constants will of course be even. A table of them will be created called <module>vectors: so you will end up with

```
editor_deleteline = 2

..

editorhandler:
```

```
          jmp  (editorvectors,x)
editorvectors:
          .word EditorControlHandler
          .word     EditorDeleteLine
```

for example.

Each module is in its own subdirectory, with a <module>.asm which contains all the module components, which is generated from the module contents (e.g. .asm files other than <module>.asm) .inc files are put up to the start (including module.inc, which appears first)

The first vector is always a routine which takes a parameter in A and is used for control purposes. A = 0 initialises the module.

Multiple files are assembled in alphabetical order.

Macros are generated as follows, along with the constants, in <module>.inc

```
Editor_DeleteLine   sets X, calls DeleteLine (one for each function)
```

The python script that generates these files has an internal list that it uses, as only one list can be used. Note that to allow for multiple device.xxx, the directory is device.xxx but the identifier is system.

Routines normally prefer A as the first parameter, and Y as the second. Stacks are normally indexed on X, so this stack value is passed in A normally.

All labels in the module, except general data labels, should begin with the initial letter of the module they are defined in.

Stack

The stack is the usual design. 6 bytes in sign, and either a 2-byte address (string) 4-byte constant (integer) or 4-byte mantissa and 1 byte exponent (floating point).

The type of byte is:

- Bit 7: indicates a reference (1) or constant (0).
- Bit 6: indicates integer/floating point (0) or string address (1).
- Bits 1...5: unused, currently 0.
- Bit 0: indicates integer (0) / floating point (1)

Note that a reference means something that can accommodate a value, string, fp or int, so while the string may be a reference of itself, this doesn’t mean it is a “storage slot”.

Tokens

Token Grouping

Group/Token	Constant	Contents
ASCII ($00-$39)		Used in identifier $2D (-) is used for underscore, allows A-Z 0-9. and _
Type ($3A-$3F)	TOK_[ISR]<ARR>	Represent integer, string, real and the appropriate arrays, mark end of identifier. % %($ $(# #(
Constant ($40-$7F)		Constant 0-63. Each successive constant shifts this left 6 pixels (in practice, left 8 right 2) and replaces the lower 6 bits.
End of Line ($80)	TOK_EOL	Indicates end of line
Token shift ($81-3)	TOK_SHIFT1...3	Shifts to other token sets, must be followed by $86-$FF
FP Constant ($84)	TOK_FPC	Marks a floating-point constant, which is of size TOK_FPC_SIZE bytes including the constant.
String ($85)	TOK_STR	Marks a string, the length of the string follows (0-255 chars), ASCII 32.127 only.
Binary Operators	TOK_BINARYST	Binary operators: and or xor + - * / mod >> << ^ >= <= = <> > < ! ?
Structures	TOK_STRUCTST	Keywords which affect the structure level.
Unary Operators	TOK_UNARYST	Unary operators (mostly functions), but there are duplicate unaries for! ? and -. Also (is a unary function.
Keywords/Tokens	TOK_TOKENS	Other keywords and tokens and syntactic markers, general.

Keywords as far as TOK_UNARYST have an associated table which either identify them as commands or binary functions, structure modifiers or identify their precedence level.

Anything after TOK_TOKENS is just a straightforward Command entry.

$83 is reserved for floating point functions like LN and so on, and system unary functions,