The Tali Forth 2 Manual

Scot W. Stevenson

2. March 2018

Abstract

Tali Forth 2 is a bare-metal ANSI(ish) Forth for the 65c02 8-bit MPU.

Contents

List of Tables

Part I

Introduction

Chapter 1

Overview

1.1 A (very) brief introduction to Forth

Part II

Using Tali Forth

Chapter 2

Installation

2.1 Downloading

Tali Forth was created to be easy to get started with. In fact, all you should need is the `ophis.bin` binary file and the `py65mon` simulator.

2.1.1 Downloading Tali Forth

Tali Forth 2 lives on GitHub at **FEHLT**.

2.1.2 Downloading the py65mon Simulator

Tali comes with an assembled version that should run out of the box with the `py65mon` simulator from https://github.com/mnaberez/py65. This is a Python program that should run on various operating systems.

To install py65mon with Linux, use the command `sudo pip install -U py65`. If you don't have PIP installed, you will have to add it first with `sudo apt-get install python-pip`. There is a `setup.py` script as part of the package, too.

2.2 Running the binary

To start the emulator, run: `py65mon -m 65c02 -r ophis.bin`.

Chapter 3

Running Tali Forth

3.1 Native compiling

In a pure subroutine-threaded Forth, higher-level words are merely a series of subroutine jumps. For instances, the Forth word `[char]`, formal Forth definition

```
: [char] char postpone literal ; immediate
```

in assembler is simply

```
                jsr xt_char
                jsr xt_literal
```

as an immediate, compile-only word. Theare are two obvious problems with this method: First, it is slow, because each `jsr/rts` pair consumes four bytes and 12 cycles overhead. Second, for smaller words, it uses far more bytes. Take for instance `drop`, which in its naive form is simply

```
                inx
                inx
```

for two bytes and four cycles. The jump to `drop` uses more space and takes far longer than the word itself. (In practice, `drop` checks for underflow, so the actual assembler code is

```
                cpx #dsp0-3
                bmi +
                lda #11         ; error code for underflow
                jmp error
*
                inx
                inx
```

for eleven bytes. We'll discuss the underflow check further below.)

To get rid of this problem, Tali Forth supports **native compiling**. The system variable `nc-limit` sets the threshhold up to which a word will be included not as a subroutine jump, but machine language. Let's start with an example where `nc-limit` is set to zero, that is, all words are compiled as subroutine jumps. Take a simple word such as

```
        : aaa 0 drop ;
```

and check the actual code with SEE:

```
        see aaa
          nt: 7CD  xt: 7D8
         size (decimal): 6

        07D8  20 52 99 20 6B 88  ok
```

(The actual addresses might be different, this is from the ALPHA release). Our word `aaa` consists of two subroutine jumps, one to zero and one to DROP. Now, if we increase the threshhold to 20, we get different code, as this console session shows:

```
        20 nc-limit !  ok
        : bbb 0 drop ;  ok
        see bbb
          nt: 7DF  xt: 7EA
         size (decimal): 17

        07EA  CA CA 74 00 74 01 E0 77  30 05 A9 0B 4C C7 AC E8
        07FA  E8  ok
```

Even though the definition of `bbb` is the same as `aaa`, we have totally different code: The number 0001 is pushed to the Data Stack (the first six bytes), then we check for underflow (the next nine bytes), and finally we `drop` by moving X, the Data Stack Pointer. Our word is definitely longer, but have just saved 12 cycles.

To experiment with various parameters for native compiling, the Forth word `words&sizes` is included in `user_words.fs` (but commented out by default). The Forth is:

```
: words&sizes ( --)
        latestnt
        begin
                dup
        0<> while
                dup name>string type space
                dup wordsize u. cr
                2 + @
        repeat
        drop ;
```

Changing `nc-limit` should show differences in the Forth words.

3.2 Underflow stripping

Checking for underflow helps during the design and debug phases of writing Forth code, but once it ready to ship, those nine bytes per check hurt, as we see in the case above. To allow those checks to be stripped, we can set the system variable `uf-strip` to `true`.

3.3 Gotchas

Tali has a 16-bit cell size (use `1 cells 8 .` to get the cells size in bits with any Forth), which can trip up calculations when compared to the *de facto* standard Gforth with 64 bits. Take this example:

```
( Gforth )      decimal 1000 100 um* hex swap u. u.  186a0 0  ok
( Tali Forth)   decimal 1000 100 um* hex swap u. u.  86a0 1  ok
```

Tali has to use the upper cell of a double-celled number to correctly report the result, while Gforth doesn't. If the conversion from double to single is only via a `drop` instruction, this will produce different results.

3.4 Reporting a problem

Chapter 4

The Editor

(Currently, there is no editor installed.)

Part III

The Internals

Chapter 5

How Tali Forth works

5.1 Stack

Tali Forth 2 uses the lowest part of the top half of Zero Page for the Data Stack (DS). This leaves the lower half of the Zero Page for any kernel stuff the user might require. The DS therefore grows towards the initial user variables. See the file `definitions.asm` for details. Because of the danger of underflow, it is recommended that the user kernel's variables are keep closer to `$0100` than to `$007f`.

The X register is used as the Data Stack Pointer (DSP). It points to the least significant byte of the current top element of the stack ('Top of the Stack', TOS).[1]

Initially, the DSP points to `$78`, not `$7F` as might be expected. This provides a few bytes as a 'floodplain' in case of underflow. The initial value of the DSP is defined as `dsp0` in the code.

5.1.1 Single cell values

Since the cell size is 16 bits, each stack entry consists of two bytes. They are stored little endian (least significant byte first). Therefore, the DSP points to the LSB of the current TOS.[2]

Because the DSP points to the current top of the stack, the byte it points to after boot – `dsp0` – will never be accessed: The DSP is decremented first with two `dex` instructions, and then the new value is placed on the stack. This means that the initial byte is garbage and can be considered part of the floodplain.

```
%                +--------------+
%                |          ... |
%                +-            -+
%                |              |   ...
%                +-  (empty)   -+
%                |              |  FE,X
%                +-            -+
%          ...   |              |  FF,X
%                +==============+
%       \$0076   |           LSB|  00,X   <-- DSP (X Register)
%                +-    TOS     -+
%       \$0077   |           MSB|  01,X
%                +==============+
%       \$0078   |  (garbage)   |  02,X   <-- DSP0
%                +--------------+
%       \$0079   |              |  03,X
%                + (floodplain) +
%       \$007A   |              |  04,X
%                +--------------+
```

[1]In the first versions of Tali, the DSP pointed to the next *free* element of the stack. The new system makes detecting underflow easier and parallels the structure of Liara Forth.

[2]Try reading that last sentence to a friend who isn't into computers. Aren't abbreviations fun?

Snapshot of the Data Stack with one entry as Top of the Stack (TOS). The DSP has been increased by one and the value written.

Note that the 65c02 system stack – used as the Return Stack (RS) by Tali – pushes the MSB on first and then the LSB (preserving little endian), so the basic structure is the same for both stacks.

Because of this stack design, the second entry ('next on stack', NOS) starts at `02,X` and the third entry ('third on stack', 3OS) at `04,X`.

5.1.2 Underflow detection

In contrast to Tali Forth 1, this version contains underflow detection for most words. It does this by comparing the Data Stack Pointer (X) to values that it must be smaller than (because the stack grows towards 0000). For instance, to make sure we have one element on the stack, we write

```
                cpx \#dsp0-1
                bmi okay

                lda \#11         ; error string for underflow
                jmp error
okay:
                (...)
```

For the most common cases, this gives us:

1 cell	dsp0-1
2 cells	dsp0-3
3 cells	dsp0-5

Though underflow detection slows the code down slighly, it adds enormously to the stability of the program.

5.1.3 Double cell values

The double cell is stored on top of the single cell. Note this places the sign bit at the beginning of the byte below the DSP.

```
%               +----------------+
%               |                |
%               +================+
%               |             LSB|  \$0,x   <-- DSP (X Register)
%               +-+  Top Cell   -+
%               |S|           MSB|  \$1,x
%               +-+--------------+
%               |             LSB|  \$2,x
%               +- Bottom Cell  -+
%               |             MSB|  \$3,x
%               +================+
```

Tali Forth 2 does not check for overflow, which in normal operation is too rare to justify the computing expense.

5.2 Dictionary

Tali Forth 2 follows the traditional model of a Forth dictionary – a linked list of words terminated with a zero pointer. The headers and code are kept separate to allow various tricks in the code.

5.2.1 Elements of the Header

Each header is at least eight bytes long:

```
              8 bit     8 bit
               LSB       MSB
```

```
%nt\_word ->  +--------+--------+
%          +0 | Length | Status |
%             +--------+--------+
%          +2 | Next Header     | nt\_next\_word
%             +-----------------+
%          +4 | Start of Code   | xt\_word
%             +-----------------+
%          +6 | End of Code     | z\_word
%             +--------+--------+
%          +8 | Name   |        |
%             +--------+--------+
%             |        |        |
%             +--------+--------+
%             |        |  ...   |
%          +n +--------+--------+
```

Each word has a `name token` (nt, `nt_word` in the code) that points to the first byte of the header. This is the length of the word's name string, which is limited to 255 characters.

The second byte in the header (index 1) is the `status byte`. It is created by the flags defined in the file `definitions.asm`:

CO	Compile Only
IM	Immediate Word
NN	Never Native Compile
AN	Always Native Compile

Note there are currently four bits unused. The status byte is followed by the **pointer to the next header** in the linked list, which makes it the named token of the next word. A `0000` in this position signales the end of the linked list, which by convention is the word `bye`.

This is followed by the current word's **execution token** (xt, `xt_word`) that points to the start of the actual code. Some words that have the same functionality point to the same code block. The **end of the code** is referenced through the next pointer (`z_word`) to enable native compilation of the word if allowed.

The **name string** starts at the eighth byte. The string is *not* zero-terminated. By default, the strings of Tali Forth 2 are lower case, but case is respected for words the user defines, so 'quarian' is a different words than 'QUARIAN'.

5.2.2 Structure of the Header List

Tali Forth 2 distinguishes between three different list sources: The **native words** that are hard-coded in the file `native_words.asm`, the **forth words** which are defined as high-level words and then generated at run-time when Tali Forth starts up, and **user words** in the file `user_words.asm` which is empty when Tali Forth ships.

Tali has an unusually high number of native words in an attempt to make the Forth as fast as possible on the 65c02. The first word in the list – the one that is checked first – is always `drop`, the last one – the one checked for last – is always `bye`. The words which are (or are assumed to be) used more than others come first. Since humans are slow, words that are used more interactively like `words` come later.

The list of Forth words ends with the intro string. This functions as a primitive form of a self-test: If you see the string and only the string, the compilation of the Forth words worked.

5.3 Memory Map

Tali Forth 2 was developed with a simple 32 KiB RAM, 32 KiB ROM design.

```
%   \$0000  +-------------------+  ram\_start, zpage, user0
%           |  User varliables  |
%           +-------------------+
%           |                   |
%           |  \^  Data Stack    |  <-- dsp
%           |  |                |
```

```
%    \$0078  +-------------------+  dsp0, stack
%            |                   |
%            |   (Reserved for   |
%            |      kernel)      |
%            |                   |
%    \$0100  +===================+
%            |                   |
%            |  \^  Return Stack  |  <-- rsp
%            |  |                |
%    \$0200  +-------------------+  rsp0, buffer, buffer0
%            |  |                |
%            |  v  Input Buffer  |
%            |                   |
%    \$0300  +-------------------+  cp0
%            |  |                |
%            |  v  Dictionary    |
%            |       (RAM)       |
%            |                   |
%            ~~~~~~~~~~~~~~~~~~~~~  <-- cp
%            |                   |
%            |                   |
%            |                   |
%    \$7fff  ####################  ram\_end
%    \$8000  |                   |  forth, code0
%            |                   |
%            |                   |
%            |    Tali Forth     |
%            |     (24 KiB)      |
%            |                   |
%            |                   |
%    \$e000  +-------------------+
%            |                   |  kernel\_putc, kernel\_getc
%            |      Kernel       |
%            |                   |
%    \$f000  +-------------------+
%            |   I/O addresses   |
%            +-------------------+
%            |                   |
%            |      Kernel       |
%            |                   |
%    \$fffa  +-------------------+
%            |  65c02 vectors    |
%    \$ffff  +-------------------+
```

5.4 Input

Tali Forth 2, like Liara Forth, follows the ANSI input model with `refill` instead of older forms. There are up to four possible input sources in Forth (see C&D p. 155):

1. The keyboard ('user input device')
2. A character string in memory
3. A block file
4. A text file

To check which one is being used, we first call `blk` which gives us the number of a mass storage block being used, or 0 for the 'user input device' (keyboard). In the second case, we use SOURCE-ID to find out where input is coming from: 0 for the keyboard, `-1 ($FFFF)` for a string in memory, and a number `n` for a file-id. Since Tali currently doesn't support blocks, we can skip the `blk` instruction and go right to `source-id`.

5.4.1 Starting up

The intial commands after reboot flow into each other: `cold` to `abort` to `quit`. This is the same as with pre-ANSI Forths. However, `quit` now calls `refill` to get the input. `refill` does different things based on which of the four input sources (see above) is active:

Keyboard entry This is the default. Get line of input via `accept` and return `true` even if the input string was empty.

`evaluate` string Return a FALSE flag.

Input from a buffer Not implemented at this time.

Input from a file Not implemented at this time.

5.4.2 The Command Line Interface

Tali Forth accepts input lines of up to 256 characters. The address of the current input buffer is stored in `cib` and is either `ibuffer1` or `ibuffer2`, each of which is 256 bytes long. The length of the current buffer is stored in `ciblen` – this is the address that `>in` returns.

When a new line is entered, the address in `cib` is swapped, and the contents of `ciblen` are moved to `piblen` (for 'previous input buffer'). `ciblen` is set to zero. When the previous entry is requested, the address in `cib` is swapped back, and `ciblen` and `piblen` are swapped as well. `source` by default returns `cib` and `ciblen` as the address and length of the input buffer.

At some point, this system might be expanded to a real history list.

5.4.3 `save-input` and `restore-input`

5.4.4 evaluate

(Automatically calls SAVE-INPUT and RESTORE-INPUT)

5.4.5 state

5.5 Create/Does

`create/does>` is the most complex, but also most powerful part of Forth. Understanding how it works in Tali Forth is important if you want to be able to modify the code. In this text, we walk through the generation process for a Subroutine Threaded Code (STC) such as Tali Forth. For a more general take, see Brad Rodriguez' series of articles at There is a discussion of this walkthrough at

We start with the following standard example, the Forth version of `constant`:

```
        : constant create , does> @ ;
```

We examine this in three phases or "sequences", based on Derick and Baker (see Rodriguez for details):

SEQUENCE I: Compiling the word `CONSTANT`

CONSTANT is a "defining word", one that makes new words. In pseudocode, and ignoring any compilation to native 65c02 assembler, the above compiles to:

```
        ((Header "CONSTANT"))
        jsr CREATE
        jsr COMMA
        jsr (DOES>)          ; from DOES>
   a:   jsr DODOES           ; from DOES>
   b:   jsr FETCH
```

```
        rts
```

To make things easier to explain later, we've added the labels 'a' and 'b' in the listing. Note that `does>` is an immediate word that adds not one, but two subroutine jumps, one to `(does>)` and one to `dodoes`, which is a pre-defined system routine like `dovar`. we'll get to it later.

In Tali Forth, a number of words such as `defer` are 'hand-compiled', that is, instead of using forth such (in this case,

```
        : defer create ['] abort , does> @ execute ;
```

we write an opimized assembler version ourselves (see the actual `defer` code). In these cases, we need to use `(does>)` and `dodoes` instead of `does>` as well.

SEQUENCE II: Executing the word CONSTANT / creating LIFE

Now when we execute

```
        42 constant life
```

this pushes the RTS of the calling routine – call it 'main' – to the 65c02's stack (the Return Stack, as Forth calls it), which now looks like this:

```
        ((1)) RTS               ; to main routine
```

Without going into detail, the first two subroutine jumps of `constant` give us this word:

```
        ((Header "LIFE"))
        jsr DOVAR               ; in CFA, from LIFE's CREATE
        4200                    ; in PFA (little-endian)
```

Next, we `jsr` to `(does>)`. The address that this pushes on the Return Stack is the instruction of `constant` we had labeled 'a'.

```
        ((2)) RTS to CONSTANT ("a")
        ((1)) RTS to main routine
```

Now the tricks start. `(does>)` takes this address off the stack and uses it to replace the `dovar` `jsr` target in the CFA of our freshly created `life` word. We now have this:

```
        ((Header "LIFE"))
        jsr a                   ; in CFA, modified by (DOES>)
   c:   4200                    ; in PFA (little-endian)
```

Note we added a label 'c'. Now, when `(does>)` reaches its own `rts`, it finds the RTS to the main routine on its stack. This is Good Thingi™, because it aborts the execution of the rest of `constant`, and we don't want to do `dodoes` or `fetch` now. We're back at the main routine.

SEQUENCE III: Executing LIFE

Now we execute the word `life` from our 'main' program. In a STC Forth such as Tali Forth, this executes a subroutine jump.

```
        jsr LIFE
```

The first thing this call does is push the return address to the main routine on the 65c02's stack:

```
        ((1)) RTS to main
```

The CFA of `life` executes a subroutine jump to label 'a' in `constant`. This pushes the `rts` of `life` on the 65c02's stack:

```
        ((2)) RTS to LIFE ("c")
        ((1)) RTS to main
```

This `jsr` to a lands us at the subroutine jump to `dodoes`, so the return address to `constant` gets pushed on the stack as well. We had given this instruction the label 'b'. After all of this, we have three addresses on the 65c02's stack:

```
        ((3)) RTS to CONSTANT ("b")
        ((2)) RTS to LIFE ("c")
        ((1)) RTS to main
```

`dodoes` pops address 'b' off the 65c02's stack and puts it in a nice safe place on Zero Page, which we'll call 'z'. More on that in a moment. First, `dodoes`. pops the `rts` to `life`. This is 'c', the address of the PFA or `life`, where we stored the payload of this constant. Basically, `dodoes` performs a `dovar` here, and pushes 'c' on the Data Stack. Now all we have left on the 65c02's stack is the `rts` to the main routine.

```
        [1] RTS to main
```

This is where 'z' comes in, the location in Zero Page where we stored address 'b' of `constant`. Remember, this is where `constant`'s own PFA begins, the `fetch` command we had originally codes after `does>` in the very first definition. The really clever part: We perform an indirect `jmp` – not a `jsr`! – to this address.

```
        jmp (z)
```

Now `constant`'s little payload programm is executed, the subroutine jump to `fetch`. Since we just put the PFA ('c') on the Data Stack, `fetch` replaces this by 42, which is what we were aiming for all along. And since `constant` ends with a `rts`, we pull the last remaining address off the 65c02's stack, which is the return address to the main routine where we started. And that's all.

Put together, this is what we have to code:

`does>`: Compiles a subroutine jump to `(does>)`, then compiles a subroutine jump to `dodoes`.

`(does>)`: Pops the stack (address of subroutine jump to `dodoes` in `constant`, increase this by one, replace the original `dovar` jump target in `life`.

`dodoes`: Pop stack (`constant`'s PFA), increase address by one, store on Zero Page; pop stack (`life`'s PFA), increase by one, store on Data Stack; `jmp` to address we stored in Zero Page.

Remember we have to increase the addresses by one because of the way `jsr` stores the return address for `rts` on the stack on the 65c02: It points to the third byte of the `jsr` instruction itself, not the actual return address. This can be annoying, because it requires a sequence like:

```
        inc z
        bne +
        inc z+1
*       (...)
```

Note that with most words in Tali Forth, as any STC Forth, the distinction between PFA and CFA is meaningless or at least blurred, because we go native anyway. It is only with words generated by `create/does>` where this really makes sense.

5.6 Branches

3

For `if/then`, we need to compile something called a 'conditional forward branch', traditionally called `0branch`. Then, at run-time, if the value on the Data Stack is false (flag is zero), the branch is taken ('branch on zero', therefore the name). Execpt that we don't have the target of that branch yet – it will later be added by `then`. For this to work, we remember the address after the `0branch` instruction during the compilation of `if`. this is put on the Data Stack, so that `then`

[3] This section and the next one are based on a discussion at http://forum.6502.org/viewtopic.php?f=9&t=3176, see there for more details. Another take on this subject that handles things a bit differently is at

knows where to compile it's address in the second step. Until then, a dummy value is compiled after `0branch` to reserve the space we need.

In Forth, this can be realized by

```
: if  postpone 0branch here 0 , ; immediate
```

and

```
: then here swap ! ; immediate
```

Note `then` doesn't actually compile anything at the location in memory where it is at. It's job is simply to help `if` out of the mess it created. If we have an `else`, we have to add an unconditional `branch` and manipulate the address that `if` left on the Data Stack. The Forth for this is:

```
: else  postpone branch here 0 , here rot ! ; immediate
```

Note that `then` has no idea what has just happened, and just like before compiles its address where the value on the top of the Data Stack told it to – except that this value now comes from `else`, not `if`.

5.6.1 Loops

Loops are far more complicated, because we have `do ?do loop +loop`, `unloop`, and `leave` to take care of. These can call up to three addresses: One for the normal looping action (`loop/+loop`), one to skip over the loop at the beginning (`?do`) and one to skip out of the loop (`leave`).

Based on a suggestion by Garth Wilson, we begin each loop in run-time by saving the address after the whole loop construct to the Return Stack. That way, `leave` and `?do` know where to jump to when called, and we don't interfere with any `if/then` structures. On top of that address, we place the limit and start values for the loop.

The key to staying sane while designing these constructs is to first make a list of what we want to happen at compile-time and what at run-time. Let's start with a simple `do/loop`.

`do` at compile-time:

- Remember current address (in other words, `here`) on the Return Stack (!) so we can later compile the code for the post-loop address to the Return Stack
- Compile some dummy values to reserve the space for said code
- Compile the run-time code; we'll call that fragment (`do`)
- Push the current address (the new `here`) to the Data Stack so `loop` knows where the loop contents begin

`do` at run-time:

- \- Take limit and start off Data Stack and push them to the Return Stack

Since `loop` is just a special case of `+loop` with an index of one, we can get away with considering them at the same time.

`loop` at compile time:

- \- Compile the run-time part `(+loop)`
- \- Consume the address that is on top of the Data Stack as the jump target for normal looping and compile it
- Compile `unloop` for when we're done with the loop, getting rid of the limit/start and post-loop addresses on the Return Stack
- Get the address on the top of the Return Stack which points to the dummy code compiled by `do`
- At that address, compile the code that pushes the address after the list construct to the Return Stack at run-time

`loop` at run-time (which is `(+loop)`)

- Add loop step to count
- Loop again if we haven't crossed the limit, otherwise continue after loop

At one glance, we can see that the complicated stuff happens at compile-time. This is good, because we only have to do that once for each loop.

In Tali Forth, these routines are coded in assembler. With this setup, `unloop` becomes simple (six PLAs – four for the limit/count of `do`, two for the address pushed to the stack just before it) and `leave` even simpler (four PLAs for the address).

Chapter 6

Developing

6.1 Adding new words

The easiest way to add new words to Tali Forth is to include them in the file `forth_code/user_words.fs`.

6.2 Deeper changes

Tali Forth was not only placed in the public domain to honor the tradition of giving the code away freely. It is also to let people play around with it and adapt it to their own machines. This is also the reason it is (perversely) overcommented.

To work on the internals of Tali Forth, you will need the Ophis assembler.

6.2.1 The Ophis Assembler

Michael Martin's Ophis Cross-Assember can be downloaded from http://michaelcmartin.github.io/Ophis/. It uses a slightly different format than other assemblers, but is in Python and therefore will run on almost any operating system. To install Ophis on Windows, use the link provided above. For Linux:

```
git clone https://github.com/michaelcmartin/Ophis
cd src
sudo python setup.py install
```

Switch to the folder where the Tali code lives, and assemble with the primitive shell script provided: `./assemble.sh` The script also automatically updates the file listings in the `docs` folder. Note that Ophis will not accept math operation characters in label names because it will try to perform those operations. Because of this, we use underscores for label names. This is a major difference to Liara Forth.

6.3 General notes

- The X register should not be changed without saving its pointer status.
- The Y register is free to be changed by subroutines. This means it should not be expected to survive subroutines unchanged.
- All words should have one point of entry – the `xt_word` link – and one point of exit at `z_word`. In may cases, this means a branch to an internal label `done` right before `z_word`.
- Because of the way native compiling works, the usual trick of combining JSR/RTS pairs to a single JMP (usually) doesn't work.

6.4 Coding style

Until I get around to writing a tool for Ophis assembler code that formats the source file the way gofmt does for Go (golang), I work with the following rules:

- Actual opcodes are indented by **two tabs**
- Tabs are **eight characters long** and converted to spaces
- Function-like routines are followed by a one-tab indented 'function doc' based on the Python 3 model: Three quotation marks at the start, three at the end it its own line, unless it is a one-liner. This should make it easier to automatically extract the docs for them at some point.
- The native words have a special commentary format that allows the automatic generation of word list by a tool in the tools folder, see there for details.
- Assembler mnenomics are lower case. I get enough uppercase insanity writing German, thank you very much.
- Hex numbers are also lower case, such as `$FFFE`
- Numbers in mnemonics are a stripped-down as possible to reduce visual clutter: `lda 0,x` instead of `lda $00,x`.
- Comments are included like popcorn to help readers who are new both to Forth and 6502 assembler.

6.5 Testing

There is no automatic or formal test suite available at this time, and due to space considerations, there probably never will be. The file `docs/testwords.md` includes a collection of words that will help with some general cases.

6.6 Code Cheat Sheet

While coding a Forth, there are certain assembler fragments that get repeated over and over again. These could be included as macros, but that can make the code harder to read for somebody only familiar with basic assembly.

Some of these fragments could be written in other variants, such as the "Push value" version, which could increment the DSP twice before storing a value. We try to keep these in the same sequence (a "dialect" or "code mannerism" if you will) so we have the option of adding code analysis tools later.

But first:

Appendix A

FAQ

A.1 Why does Tali Forth take so long to start up?

After the default kernel string is printed, you'll notice a short pause that didn't occur with Tali Forth 1. This is because Tali Forth 2 has more words defined in high-level Forth (see `forth-words.asm`) than Tali did. The pause happens because they are being compiled on the fly.

A.2 Why 'Tali' Forth?

I like the name, and we're probably not going to have anymore kids I can give it to.

(If it sounds vaguely familiar, you're probably thinking of Tali'Zorah vas Normandy, a character in the 'Mass Effect' universe created by EA / BioWare. This software has absolutely nothing to do with either the game or the companies and neither do I, expect that I've played the games and enjoyed them, though I do have some issues with *Andromeda*. Like what happened to the quarian ark?)

A.3 Then who is 'Lara'?

Liara Forth is a STC Forth for the big sibling of the 6502, the 65816. Tali 1 came first, then I wrote Liara with that knowledge and learned even more, and now Tali 2 is such much better for the experience. Oh, and it's another 'Mass Effect' character.

Appendix B

Forth tests

Appendix C

Thanks

Tali Forth 2 would not have been possible without the kind support of a very large number of individuals.

Index