The Tali Forth 2 Manual (ALPHA)

Scot W. Stevenson

April 13, 2018

Abstract

Tali Forth 2 is a bare-metal ANSI(ish) Forth for the 65c02 8-bit MPU. It aims to be, roughly in order of importance:

Easy to try. Download the source – or even just the binary – and you can immediately run it in an emulater. This lets you experiment with a working 8-bit Forth for the 65c02 without any special configuration.

Simple. The subroutine-threaded (STC) design and happily overcommented source code give hobbyists the chance to study a working Forth at the lowest level. The manual – this document – explains structure and code in detail. The aim is to make it easy to port Tali Forth 2 to various 65c02 hardware projects.

Specific. Many Forths available are 'general' implementations with a small core adapted to the target processor. Tali Forth 2 was written as a "bare metal Forth" for the 65c02 8-bit MPU and that MPU only, with its strengths and limitations in mind.

Standardized. Most Forths available for the 65c02 are based on ancient, outdated templates such as FIG Forth. Learning Forth with them is like trying to learn modern English by reading Chaucer. Tali Forth (mostly) follows the current ANSI Standard.

Tali Forth is hosted at GitHub at https://github.com/scotws/TaliForth2. The discussion thread is at 6502.org at http://forum.6502.org/viewtopic.php?f=9&t=2926.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Why

> Forth is well suited to resource-constrained situations. It doesn't need lots of memory and doesn't have much overhead. It can take full advantage of whatever hardware or interfaces exist.
>
> – Charles Moore, 'Chuck Moore: Geek of the Week', redgate Hub 2009

1.1 The big picture

This section provides background information on Forth, the 6502 processor, and why anybody would want to combine the two. It can be safely skipped if you already know all those things.

1.1.1 The 6502 MPU

It is a well-established fact that humanity reached the apex of processor design with the 6502 in 1976. Created by a team including Chuck Peddle and Bill Mensch, it was the engine that powered the 8-bit home computer revolution of the 1980s.[1] The VIC-20, Commodore PET, Apple II, and Atari 800 all used the 6502, among others.

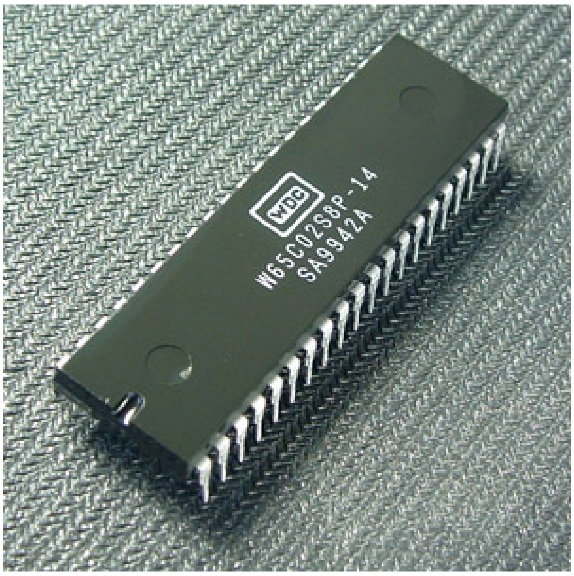

Figure 1.1: *The 65c02 MPU.* Photo: Anthony King, released in the public domain

More than 40 years later, the processor is still in production by the Western Design Center. Apart from commercial uses, there is an active hobbyist scene centered on the website 6502.org.

[1] Rumor has it that there was another MPU called 'Z80', but it ended up being a mere footnote.

Quite a number of people have built their own 8-bit computers based on this chip and the instructions there, including a a primer by Garth Wilson. It is for these systems that Tali Forth 2 was created.

The most important variant of the 65c02 produced today is the 65c02, a CMOS chip with some additional instructions. It is for this chip that Tali Forth 2 was written.

But why program in 8-bit assembler at all?[2] The 65c02 is fun to work with because of its clean instruction set architecture (ISA). This is not the place to explain the joys of assembler. The official handbook for the 65c02 is *Programming the 65816, including the 6502, 65C02 and 65802*[6].

1.1.2 Forth

> If C gives you enough rope to hang yourself, Forth is a flamethrower crawling with cobras.
>
> – Elliot Williams, *Forth: The Hacker's Language*

Forth is the *enfant terrible* of programming languages. It was invented by Charles 'Chuck' Moore in the 1960s to do work with radio astronomy, way before there were modern operating systems or programming languages.[3] As a language for people who actually need to get things done, it lets you run with scissors, play with fire, and cut corners until you've turned a square into a circle. Forth is not for the faint-hearted: It is trivial, for instance, to redefine 1 as 2 and `true` as `false`. Though you can do really, really clever things with few lines of code, the result can be hard for other people to understand, leading to the reputation of Forth begin a 'write-only language'. However, Forth excels when you positively, absolutely have to get something done with hardware that is really too weak for the job.

It should be no surprise that NASA is one of the organizations who use Forth. The *Cassini* mission to Saturn used a Forth CPU, for instance. It is also perfect for small computers like the 8-bit 65c02. After a small boom in the 1980s, more powerful computers led to a decline of the language. The 'Internet of Things' with embedded small processors has led to a certain amount renewed interest in the language. It helps that Forth is easy to implement: It is stack-based, uses reverse polish notation (RPN) and a simple threaded interpreter model.

There is no way this document can provide an adiquate introduction to Forth. There are quite a number of tutorials, however, such as *A Beginner's Guide to Forth* by J.V. Nobel[9] or the classic (but slightly dated) *Starting Forth*[2] by Leo Brodie. Gforth, one of the more powerful free Forths, comes with its own tutorial.[4]

1.2 Writing your own Forth

Even if the 65c02 is great and Forth is brilliant, why got to the effort of writing a new, bare-metal version of the languages? After almost 50 years, shouldn't there be a bunch of Forths around already?

1.2.1 FIG Forth

In fact, the classic Forth availble for the whole group of 8-bit MPUs is FIG Forth – 'FIG' stands for 'Forth Interest Group'. Ported to various architectures, it was original based on an incarnation for the 6502 written by Bill Ragsdale and Robert Selzer. There are PDFs of the 6502 version from September 1980 freely available – Forths are traditionally placed in the public domain – and more than one hobbyist has revised it to his machine.

However, Forth has changed a lot in the past three decades. There is now a standardized version called ANSI Forth standard, which includes such basic changes as how the `do` loop works. Learning the language with FIG Forth is like learning English with *The Canterbury Tales*.

[2] Wilson answers this question in greater detail as part of his 6502 primer

[3] A brief history of Forth can be found at https://www.forth.com/resources/forth-programming-language/

[4] Once you have understood the basics of the language, do yourself a favor and read *Thinking Forth* by Brodie[3], which deals with the philosophy of the language. Like Lisp, exposure to Forth will change the way you think about programming.

1.2.2 A modern Forth for the 65c02

Tali Forth was created to provide an easy to understand modern Forth written especially for the 65c02 that anybody can understand, adapt to their own use, and maybe actually work with. As part of that effort, the source code is heavily commented. And this document tries to explain the internals in more detail.

Chapter 2

Overview of Tali Forth

2.1 Design considerations

When creating a new Forth, there are a bunch of design decisions to be made.[1] Spoiler alert: Tali Forth ended up as a subroutine-threaded variant with a 16-bit cell size and a dictionary that keeps headers and code separate. If you don't care and just want to use the program, skip ahead.

2.1.1 Characteristics of the 65c02

Since this is a bare-metal Forth, the most important consideration is the target processor. The 65c02 only has one full register, the accumulator A, and two secondary registers X and Y. All are 8-bit wide. There are 256 bytes that are more easily addressable on the Zero Page. A single hardware stack is used for subroutine jumps. The address bus is 16 bits wide for a maximum of 64 KiB of RAM and ROM. For the default, simple setup, we assume 32 KiB of each.

2.1.2 Cell size

The 16 bit address bus suggests the cell size should be 16 bits as well. This is still easy enough to realize on a 8-bit MPU, though not as comfortable as working with the 65816, the 65c02's big brother, with an actual 16 bit register size.

2.1.3 Threading technique

A 'thread' in Forth is simply a list of addresses of words to be executed. There are four basic threading techniques:[2]

Indirect threaded (ITC) The oldest, original variant, used by FIG Forth. All other versions are modifications of this model.

Direct threaded (DTC) Includes more assembler code to speed things up, but slightly larger than ITC.

Token threaded (TTC) The reverse of DTC in that it is slower, but uses less space than the other Forths. Words are created as a table of tokens.

Subroutine threaded (STC) This technique converts the words to a simple series of `jsr` combinations.

Our lack of registers and the goal of creating a simple and easy to understand Forth makes subroutine threading the most attractive solution. We will try to mitigate the pain caused by the 12 cycle cost of each and every `jsr/rts` combination by including a relatively large number of native words.

---

[1] The best introduction to these questions is found in *Design Decisions in the Forth Kernel* by Brad Rodriguez

[2] For the 8086 MPU, Guy Kelly compared various Forth implementations in 1992[7]

2.1.4 Register use

The lack of registers – and 16 bit registers at that – becomes apparent when you realize that Forth classically uses at least four 'virtual' registers:

W	Working register
IP	Interpreter Pointer
DSP	Data Stack Pointer
RSP	Return Stack Pointer

Table 2.1: The classic Forth registers

On a modern processor like a RISC-V RV32I CPU with 32 registers of 32 bit each, this wouldn't be a problem. In fact, we'd be trying to figure out what else we could keep in a register. On the 65c02, at least we get the RSP for free with the built-in stack pointer. This still leaves three registers. We cut that number down by one through subroutine threading, which gets rid of the IP. For the DSP, we use the 65c02's Zero Page indirect addressing mode with the X register. This leaves W, which we put on the Zero Page as well.

2.1.5 Data Stack design

We'll go into greater detail on how the Data Stack works in a later chapter when we look at the internals. Briefly, the stack is realized on the Zero Page for speed. For stability, we provid underflow checks in the relevant words, but give the user the option of stripping it out for native compilation.

2.1.6 Dictionary structure

Each Forth word consists of the actual code and the header which holds the meta-data. Part of this data is the single-linked list of words which is searched.

In constrast to Tali Forth 1, which kept the header and body of the words together, Tali Forth 2 keeps them separate. This lets us play various tricks with the code to make it more effective.

2.1.7 Deeper down the rabbit hole

This concludes our overview of the basic Tali Forth 2 structure. For those interested, a later chapter will provide far more detail.

Part II

User Guide

Chapter 3

Installing

3.1 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.

3.1.1 Downloading Tali Forth

The newest version of Tali Forth 2 lives on GitHub at https://github.com/scotws/TaliForth2. You can either clone the code with `git` or simply download it. To just try the program, all you need is the `ophis.bin` binary.

3.1.2 Downloading the py65mon Simulator

Tali was written to run out of the box on the `py65mon` simulator from https://github.com/mnaberez/py65. This is a Python program that should run on various operating systems.

To install py65mon on 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.

3.2 Running the binary

To start the emulator, run:

```
        py65mon -m 65c02 -r ophis.bin
```

Note that the option `-m 65c02` is required, because Tali Forth makes extensive use of the additional commands of the CMOS version and will not run on a stock 6502 MPU.

Chapter 4

Running

> One doesn't write programs in Forth. Forth is the program.
>
> – Charles Moore, *Masterminds of Programming*[1]

4.1 Booting

booting

Out of the box, Tali Forth boots a minimal kernel`kernel` to connect to the py65mon simulator. By default, this stage ends with a line such as

```
Tali Forth 2 default kernel for py65mon (18. Feb 2018)
```

Tali Forth itself boots next, and after setting up various internal things, compiles the high level words. This causes a slight delay, depending on the number and length of these words. As the last step, Forth should spit out a boot string, something to the effect of

```
Tali Forth 2 for the 65c02
Version ALPHA 07. Mar 2018
Copyright 2014-2018 Scot W. Stevenson
Tali Forth 2 comes with absolutely NO WARRANTY
Type 'bye' to exit
```

This functions as a primitive self-test. If you have modified the high level Forth words in either `forth_words.fs` or `user_words.fs`, the boot process might fail with a variant of the error message 'unknown word'. The built-in, native words should still work.

4.2 Available words

Tali Forth comes with the following Forth words out of the box:

```
see within to d.r d. ud.r ud. .r u.r */mod */ mod /mod /
action-of is defer@ defer! while until repeat else then
if .( ( drop dup swap ! @ over >r r> r@ nip rot -rot tuck
, c@ c! +! execute emit type . u. ? false true space 0 1
2 2dup ?dup + - abs dabs and or xor rshift lshift pick char
[char] char+ chars cells cell+ here 1- 1+ 2* = <> < > 0=
0<> 0> 0< min max 2drop 2swap 2over 2variable 2r@ 2r> 2>r
invert negate dnegate c, bounds spaces bl -trailing /string
refill accept unused depth key allot create does> variable
constant value s>d d>s d- d+ erase blank fill find-name '
['] name>int int>name name>string >body defer latestxt
latestnt parse-name parse source source-id : ; compile, [ ]
0branch branch literal sliteral ." s" postpone immediate
compile-only never-native always-native nc-limit abort
abort" do ?do i j loop +loop exit unloop leave recurse quit
```

```
begin again state evaluate base digit? number >number hex
decimal count m* um* * um/mod ud/mod sm/rem fm/mod \ move
cmove> cmove pad >in <# # #s #> hold sign output input cr
page at-xy marker words wordsize aligned align bell dump .s
find word cold bye
```

(Call `words` in Tali Forth for the current list.)

Though the list might look unsorted, it actually reflects the priority in the dictionary, that is, which words are found first. For instance, the native words – those coded in assembler – start with `drop bye`, which is the last word that Tali Forth will find.[1] The words before `drop` are those that are defined in high-level Forth. For more information on the words, use the `see` command.

Note that the built-in words are lower case. Newly defined words can be in any case and will be distinct – 'KASUMI' is a different word than 'Kasumi'.

4.2.1 Standards

Tali Forth is orientated on ANSI Forth, but (currently) doesn't contain the complete set of even the core words. Tali also adopted some words from Gforth such as `bounds`. In practical terms, Tali aims to be a subset of Gforth: If a program runs on Tali, it should run on Gforth the same way or have a very good reason not to.

In addition, there are a few words that are specific to Gforth such as `nc-limit`.

4.3 Native compiling

As the name says, subroutine threaded code encodes the words as a series of subroutine jumps. Because of the overhead caused by these jumps, this can make the code slow. Therefore, Tali Forth enables 'native compiling', where the machine code from the word itself is included instead of a subroutine jump.

The parameter `nc-limit` sets the limit of how small words have to be to be natively compiled. To get the current value (usually 20), check the value of the system variable:

```
nc-limit ?
```

To set a new limit, save the maximal allowed number of bytes in the machine code like any other Forth variable:

```
40 nc-limit !
```

To complete turn off native compiling, set this value to zero.

4.4 Underflow detection

When a word tries to access more words on the stack than it is holding, an 'underflow' error occurs. Whereas Tali Forth 1 didn't check for these errors, this version does.

However, this slows the program down. Because of this, the user can turn off underflow detection for words that are natively compiled into new words. To do this, set the system variable `uf-strip` to `true`. Note this does not turn off underflow detection in the built-in words. Also, words with underflow detection which are not included in new words through native compiling will also retain their tests.

4.5 Restarting

Tali Forth has a non-standard word `cold` that resets the system. Note that this doesn't erase any data in memory, but just moves the pointers back. When in doubt, you might be better off quitting and restarting completely.

[1] If you're going to quit, speed can't be that important

4.6 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.

4.7 Reporting a problem

The best way to point out a bug or make any other form of a comment is on Tali Forth's page on GitHub at https://github.com/scotws/TaliForth2. There, you can 'open an issue', which allows other people who might have the same problem to help even when the author is not available.

Chapter 5

The Editor

(Currently, there is no editor installed.)

Part III

Developer Guide

Chapter 6

How Tali Forth works

6.1 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.

6.1.1 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`.

6.1.2 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:

Test for	Pointer offset
1 cell	dsp0-1
2 cells	dsp0-3
3 cells	dsp0-5
4 cells	dsp0-7

Table 6.1: DSP values for underflow testing

Underflow detection adds nine bytes and five cycles to the words that have it. However, it increases the stability of the program enormously.

6.1.3 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.

6.2 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.

6.2.1 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`:

Flag	Function
CO	Compile Only
IM	Immediate Word
NN	Never Native Compile
AN	Always Native Compile
UF	Underflow dectection

Table 6.2: Header flags

Note there are currently three bits unused. The status byte is followed by the **pointer to the next header** in the linked list, which makes it the name 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'.

6.2.2 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`.

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 strings. 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.

6.3 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  +-------------------+
```

6.4 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`.

6.4.1 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.

6.4.2 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`. The length of the current buffer is stored in `ciblen` – this is the address that `>in` returns. `source` by default returns `cib` and `ciblen` as the address and length of the input buffer.

6.4.3 `evaluate`

`evaluate` is used to execute commands that are in a string. A simple example would be:

```
s" 1 2 + ." evaluate
```

Tali Forth uses `evaluate` to load high-level Forth words from the file `forth_words.asc` and extra, user-defined words from `user_words.asc`.

6.5 `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 http://www.bradrodriguez.com/papers/moving3.htm. There is a discussion of this walkthrough at http://forum.6502.org/viewtopic.php?f=9&t=3153.

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

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

We examine this in three phases or "sequences", following Rodriguez based on[5]:

Sequence 1: 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:

```
        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.[3] `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 discuss those later.

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

```
        : 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 2: 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 `rtrs` 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 3: 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.

[3] This example uses the word `(does>)`, which in Tali Forth 2 is actually an internal routine that does not appear as a separate word. This version is easier to explain.

```
        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.

6.6 Control Flow

6.6.1 Branches

For `if/then`, we need to compile something called a 'conditional forward branch', traditionally called `0branch`.[4] 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` 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.[5]

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`.

6.6.2 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.

[4] Many Forths now use the words `cs-pick` and `cs-roll` instead of the `branch` variants, see http://lars.nocrew.org/forth2012/rationale.html#rat:tools:CS-PICK. Tali Forth might switch to this construction in the future.

[5] 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 http://blogs.msdn.com/b/ashleyf/archive/2011/02/06/loopty-do-i-loop.aspx

`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 `pla` instructions – four for the limit/count of `do`, two for the address pushed to the stack just before it) and `leave` even simpler (four `pla` instructions for the address).

6.7 Native Compiling

In a pure subroutine threaded code, higher-level words are merely a series of subroutine jumps. For instance, the Forth word `[char]`, formally defined in high-level Forth as

```
        : [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, the jumps use far more bytes than the actual code. Take for instance `drop`, which in its naive form is simply

```
                inx
                inx
```

for two bytes and four cycles. If we jump to this word as is assumed with pure subroutine threaded Forth, we add four bytes and 12 cycles – double the space and three times the time required by the actual working code. (In practice, it's even worse, because `drop` checks for underflow. The actual assembler code is

```
                cpx #dsp0-1
                bmi +
                lda #11         ; error code for underflow
                jmp error
*
                inx
                inx
```

for eleven bytes. We'll discuss the underflow checks 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 ;
```

An alternative is `see`, which also displays the length of a word. One way or another, changing `nc-limit` should show differences in the Forth words.

6.7.1 Return Stack special cases

There are a few words that cause problems with subroutine threaded code: Those that access the Return Stack such as `r>`, `>r`, `r@`, `2r>`, and `2>r`. For them to work correctly, we first have to remove the return address on the top of the stack, only to replace it again before we return to the caller. This mechanism would normally prevent the word from being natively compiled at all, because we'd try to remove a return address that doesn't exit.

This becomes clearer when we examine the code for `>r` (comments removed):

```
xt_r_from:
                pla
                sta tmptos
```

```
                ply

                ; --- CUT FOR NATIVE CODING ---

                dex
                dex
                pla
                sta 0,x
                pla
                sta 1,x

                ; --- CUT FOR NATIVE CODING ---

                phy
                lda tmptos
                pha

z_r_from:       rts
```

The first three and last three instructions are purely for housekeeping with subroutine threaded code. To enable this routine to be included as native code, they are removed when native compiling is enabled by the word `compile,`. This leaves us with just the six actual instructions in the center of the routine to be compiled into the new word.

6.7.2 Underflow stripping

As described above, every underflow check adds nine bytes and five cycles to the word being coded. Stripping this check by setting the `uf-strip` system variable to `true` simply removes these nine bytes from new natively compiled words.

It is possible, of course, to have lice and fleas at the some time. For instance, this is the code for `>r`:

```
xt_to_r:
                pla
                sta tmptos
                ply

                ; --- CUT HERE FOR NATIVE CODING ---

                cpx #dsp0-1
                bmi +
                lda #11
                jmp error
*
                lda 1,x
                pha
                lda 0,x
                pha

                inx
                inx

                ; --- CUT HERE FOR NATIVE CODING ---

                phy
                lda tmptos
                pha

z_to_r:         rts
```

This word has *both* native compile stripping and underflow detection. However, both can be removed from newly native code words, leaving only the eight byte core of the word to be compiled.

Chapter 7

Developing

> Programming computers can be crazy-making.
>
> – Leo Brodie, *Thinking Forth*[3]

7.1 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`.

7.2 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.

7.2.1 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.

7.2.2 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.

7.2.3 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.

7.3 Testing

There is no automatic or formal test suite available at this time, and due to space considerations, there probably never will be. This manual includes a list of test cases that can be applied by hand in the appendix.

7.4 Code Cheat Sheet

7.4.1 The Stack Drawing

This is your friend and should probably go on your wall or something.

```
              +--------------+
              |          ... |
              +-            -+
              |              |   ...
              +-  (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
              +--------------+
```

7.4.2 Coding idioms

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.

drop cell of top of the Data Stack

```
                inx
                inx
```

push a value to the Data Stack. Remember the Data Stack Pointer (DSP, the X register of the 65c02) points to the LSB of the TOS value.

```
                dex
                dex
                lda $<LSB>      ; or pla, jsr kernel_getc, etc.
                sta 0,x
                lda $<LSB>      ; or pla, jsr kernel_getc, etc.
                sta 1,x
```

pop a value off the Data Stack

```
                lda 0,x
                sta $<LSB>      ; or pha, jsr kernel_putc, etc
                lda 1,x
                sta $<MSB>      ; or pha, jsr kernel_putc, etc
                inx
                inx
```

7.4.3 vi shortcuts

One option for these is to add abbreviations to your favorite editor, which should of course be vim, because vim is cool. There are examples for that further down. They all assume that auto-indent is on and we are two tabs in with the code, and use # at the end of the abbreviation to keep them separate from the normal words. My `~/.vimrc` file contains the following lines for work on `.asm` files:

```
ab drop# inx<tab><tab>; drop<cr>inx<cr><left>
ab push# dex<tab><tab>; push<cr>dex<cr>lda $<LSB><cr>sta $00,x<cr>lda
    $<MSB><cr>sta $01,x<cr><up><up><up><up><end>
ab pop# lda $00,x<tab><tab>; pop<cr>sta $<LSB><cr>lda $01,x<cr>sta $<
    MSB><cr>inx<cr>inx<cr><up><up><up><up><up><end>
```

Chapter 8

Future plans

(See the file TODO.txt)

Appendix A

FAQ

A.1 What happened to Tali Forth 1?

Tali Forth 1, formally just Tali Forth, was my first Forth. As such, it is fondly remembered as a learning experience. You can still find it online at GitHub at https://github.com/scotws/TaliForth. When Tali Forth 2 entered BETA, Tali Forth was discontinued and does not receive bug fixes either.

A.2 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.3 Why 'Tali' Forth?

I like the name, and we're probably not going to have any more 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.4 Who is 'Liara'?

Liara Forth is a STC Forth for the big sibling of the 6502, the 65816. Tali Forth 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

Tali Forth 2 operates in a hardware environment that makes built-in tests pretty much impossible. The following collection of Forth snippets provides a basic way of testing at least some party by hand.

For stress testing, the `user_words.fs` routines contain routines such as a Mandelbrot program that can be uncommented and run.

B.1 The interpreter

B.1.1 `>in`

From https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/The-Text-Interpreter.html

```
: lat ." <foo>" ;
: flat ." <bar>" >in dup @ 3 - swap ! ;
```

A simple `flat` should print `<bar><foo>`. A bit more complicated:

```
char & parse jack& type
```

This should print `jack`.

B.2 Defining words

B.2.1 `create` and `does>`

The simplest test is to redefine `constant`:

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

B.2.2 `literal`

```
: aaa [ 1 ] literal ;
```

This should put 1 on the Data Stack at run time.

B.2.3 `bracket-char`

```
: aaa [char] q emit ;
```

This should print the letter ‘q’.

B.2.4 bracket-tick

```
: aaa ['] words execute ;
```

This should print all words in the dictionary.

B.2.5 postpone

`postpone` is aord that is complicated to code, but very useful. The following simple test is from *Starting Forth*:

```
: say-hello ." Hello" ; immediate
: greet postpone say-hello ." I speak Forth" ;
```

Here, `greet` won't print `Hello` right away.

B.2.6 find-name

This word walks through the dictionary until it finds the word presented.

```
s" words" find-name name>string type
```

should print `words`

B.2.7 word vs parse

`word` was used in ancient Forths and should be considered obsolete today. The following piece of code, taken from the *Forth Programmer's Handbook*[4], shows the differences:

```
: test1 ( "name" -- )  32 word count type ;
: test2 ( "name" -- )  32 parse type ;
```

Results of calls with 'abc' should give identical result if there are no leading spaces. However, with leading spaces, `test2` will find an empty string and abort, then throw an error because ABC will not be found in the dictionary.

B.3 Movement words

ANSI Forth comes with three words to move characters: `cmove`, `cmove>`, and `move`. Testing them is somewhat annoying, but this is how to start. Note we're mostly concerned with the correct overlapping behavior.

First, clear a region to work the tests in and define a word that saves the numbers 1 to 5 in memory:

```
here 1000 + constant start
start 20 erase
: save-five 6 1 do i dup start + c! loop ;
```

To check the initial state, run

```
start 10 dump
```

which should give the number sequence

```
00 01 02 03 04 05 00 00 00 00
```

B.3.1 cmove

The test routine for `cmove` and its correct output is:

```
start  start 3 +  5  cmove
00 01 02 00 01 02 00 01 00 00
```

B.3.2 `cmove>`

For `cmove>`, we have:

```
start  start 3 +  5  cmove>
00 01 02 00 01 02 03 04 00 00
```

The trick to interpreting these moves is to remember that `cmove>` start with the high bytes of the source string ('on the right' here), and `cmove` starts with lower bytes ('on the left'). When there is overlap, some of the string has already been overwritten and is then copied itself.

B.4 Looping

The looping constructs are the most complicated parts of Tali Forth. Currently, we use the Return Stack instead of the more traditional Data Stack as the mythical Forth 'Control Stack'.

B.4.1 Basic looping

Test normal loop:

```
: aaa 11 1 do i . loop ;
```

This should simply produce the numbers 1 to 10. Then, break up the loop over multiple lines because it turns out this can be tricky:

```
: bbb 11 1 do
  i . loop ;
```

Following Gforth, we then should test other variants:

```
: bbb1 -1 0 ?do i . -1 +loop ;
```

This should produce '0 -1', while

```
: bbb2 0 0 ?do i . -1 +loop ;
```

should print nothing.

B.4.2 Looping with `if`

Including `if` in a loop is the ultimate test if everything is correct with our stack machanics. First, a simple version will do:

```
: ccc 11 1 do i dup 5 > if . then loop ;
```

This should produce `6 7 8 9 10`. The next step is to continue with some instructions after the `then` – this actually revealed a stupid bug in the ALPHA version of Tali Forth 2.

```
: ccc 11 1 do i dup 2 = if ." two! " then . loop ;
```

This should print the string right before the number two.

B.4.3 Nested loops

A simple test of nested loops is found in *Starting Forth*:

```
: ddd cr 11 1 do
    11 1 do
        i j * 5 u.r
    loop cr loop ;
```

This should print a multiplication table from 1x1 to 10x10.

B.4.4 `exit`

```
: eee1 true if exit then ." true" ;
: eee2 false if exit then ."false" ;
```

First word should just return with `ok`, second word prints `false`.

B.4.5 `unloop`

```
: fff 11 1 do i dup 8 = if drop unloop exit then . loop ." Done
  " ;
```

should produce `1 2 3 4 5 6 7` (with no `Done`)

B.4.6 `leave`

```
: ggg 11 1 do i dup 8 = if leave then . loop ." Done" drop ;
```

should produce `1 2 3 4 5 6 7 Done` (note `Done` printed)

The Data Stack should be empty after all of these words, check with `.s`

B.4.7 `bounds`

This is a word we took from Gforth because it is just too useful.

```
: ggg s" tali" bounds do i c@ emit loop ;
```

When this is run, it should print 'tali'.

B.4.8 `recurse`

```
: hhh ( a b -- gcd ) ?dup if tuck mod recurse then ;
```

Which should produce 16 for 784 48 hhh.

Also, the classic (here from the ANSI Forth documentation):

```
: factorial ( u -- u )
   dup 2 < if drop 1 exit then
   dup 1- recurse * ;
```

For 5, the result should be 120.

B.5 Stack Stuff

B.5.1 `pick`

Use of `pick` is considered bad form in Forth, but it's a word, so we test for it. The short version: `0 pick` should be the same as `dup` and `1 pick` the same as `over`. See this brief session:

```
1 2 3 .s <3> 1 2 3  ok
0 pick .s <4> 1 2 3 3  ok
drop  ok
1 pick .s <4> 1 2 3 2  ok
drop  ok
2 pick .s <4> 1 2 3 1  ok
```

Another way to think about this is that the TOS is 'element 0', which goes nicely with the *Mass Effect* theme, of course.

B.6 Math routines

B.6.1 `fm/mod`

```
: fm swap s>d rot fm/mod swap . . ;
```

Should give you:

Input		Result	
10	7	3	1
-10	7	4	-2
10	-7	-4	-2
-10	-7	3	1

Table B.1: Correct values for `fm/mod` tests

B.6.2 `sm/rem`

```
: sm swap s>d rot sm/rem swap . . ;
```

Should give you:

Input		Result	
10	7	3	1
-10	7	-3	-1
10	-7	3	-1
-10	-7	-3	1

Table B.2: Correct values for `sm/rem` tests

Appendix C

Thanks

Tali Forth would never have been possible without the help of a very large number of people, very few of whom I have actually even met.

First, there is the crew at 6502.org who not only helped me build a 6502 computer, but also introduced me to Forth. Tali Forth would not exist without their inspiration, support, and feedback.

A special mention goes to Mike Barry, who repeatedly suggested improvements to the assembler code, saving everybody dozens of bytes and (by now) thousands of cycles.

Bibliography

[1] Federico **Biancuzzi**, *Masterminds of Programming*, O'Reilly Media 1st edition, 2009.

[2] Leo **Brodie**, *Starting Forth*, New edition 2003, https://www.forth.com/starting-forth/

[3] Leo **Brodie**, *Thinking Forth*, 1984, http://thinking-forth.sourceforge.net/#21CENTURY

[4] Edward K. **Conklin**, Elizabeth D. **Rather** *Forth Programmer's Handbook*, 3rd Edition 2010.

[5] Mitch **Derick**, Linda **Baker** *Forth Encyclopedia*, Mountain View Press 1982.

[6] David **Eyes**, Ron **Lichty** *Programming the 65816, including the 6502, 65C02 and 65802*, (Currently not available from the WDC website)

[7] Guy **Kelly** 'Forth Systems Comparisons', *Forth Dimensions* V13N6 March/April 1992 http://www.forth.org/fd/FD-V13N6.pdf

[8] Lance A. **Leventhal**, *6502 Assembly Language Programming*, OSBORNE/McGRAW-HILL 1979.

[9] J.V. **Nobel**, *A Beginner's Guide to Forth*, http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm.

Index