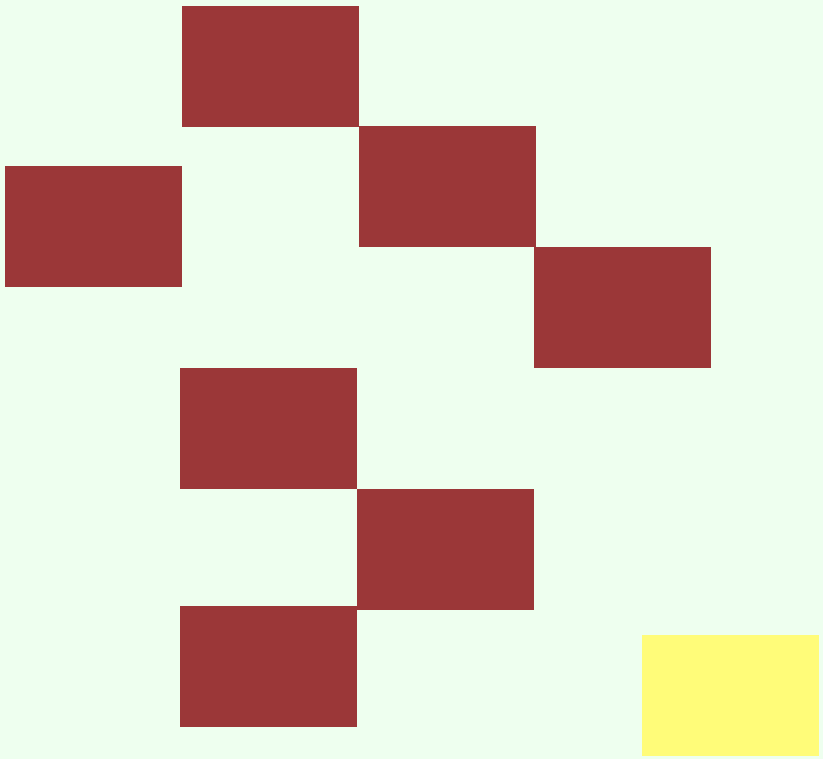


J.J. HOEKSTRA

VERSION 010

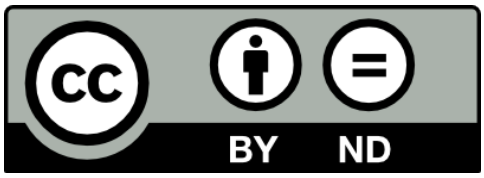
WABIFORTH PROGRAMMERS-GUIDE

FOR
WABI32V4.1.0

WARNING

a Raspberry running wabiForth
must be cooled adequately

Copyright and license



The transcendental floating point functions in wabiForth are based on (parts of) the NE10 library from ARM limited.

The NE10 library is licensed under the BSD-3 clause license. See the source-code of wabiForth for the complete license text.

Foreword

I got my first homecomputer in 1983, and I was blown away by it. No matter that it was slow and had limited memory, what fun I had with it!
I still fondly remember the long nights. With an old television for a monitor, programming, debugging, playing - all the while struggling with a crappy cassette-recorder interface.
At that time the fastest, 'bestest', baddest computer in the world was the CRAY-1. And naturally I fantasised about a homecomputer with the raw power of a CRAY-1. Later I added something to the dream; the build-in programming language should be Forth.

In 2016 I started a project to fulfil that old fantasy: build a homecomputer as fast as a CRAY-1 and with Forth as language. WabiFORTH is the result of this project.

The development-goals for wabiForth were straightforward: fun, fast, and stable. I feel these goals have been met, and I am very happy with it.

As for a speed-comparison with the CRAY-1? Well, wabiForth is around 50 times faster than optimised Fortran on a CRAY-1. And the 1 GB memory of Raspberry Pi3b+ is 128 times bigger than a full blown CRAY-1.

Dream fulfilled? I should think so!

Home computing in the early 80s focused on DIY-programming, with basic graphics, sound-generation and interfacing capabilities. The best homecomputers also had an assembler build in.

WabiForth follows this original philosophy pretty closely. Also with regard to something else; like any homecomputer from that time, wabiFORTH is 'blessed' with faults and quirks.
For instance: most modern technologies are not supported. Things like USB-ports, internet-connectivity and wireless or wired networking.
In addition wabiForth is not fully compatible with the ANSI-Forth standard, and not all ANSI words are implemented.
And worst of all, and in full parallel with the 80s, the user-guide, which you are reading now, is incomplete and a constant work-in-progress...

It is unlikely that this all will change a lot in future. WabiFORTH's raison d'être is to support programming as a hobby. Like the homecomputers from the 80s, but with more memory and much, much faster. 'Vintage computing on a lot of steroids'

WabiFORTH is what it is. Enjoy it and have fun programming, or move on to other, better, systems. (like noForth, CiForth, iForth, eForth or Mecrisp)

Jeroen Hoekstra

Content

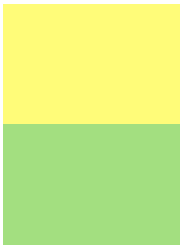
Introduction

This manual presumes knowledge of Forth. It will not teach you programming in Forth. There are excellent books available for that purpose.
The classic books "Starting Forth" and "Thinking Forth" by Leo Brodie are a good starting point. And Elisabeth Rather has co-authored an excellent book: "Forth Programmer's Handbook".

Membership of a local Forth Interest Group is also a very good way of starting with Forth.

As of May 2022 wabiForth contains over 2200 definitions. Obviously the normal Forth words are there. Of the many other words only a few are essential for a programmer.

A lot of the extra words are related to homecomputer functionality, like the sound-system and the graphical system. And around 25% of the words are related to the assembler. Finally the optimising functions account for another 200 words.



WabiForth – it's all in the name

The name wabiForth comes from the Japanse concept '**wabi-sabi**'.

Translating these two words is the easy part:

wabi: "nothing is ever finished or perfect"
sabi: "everything comes to an end"

But explaining the meaning and the role of these two words in Japanese Culture is complex, if not impossible. But part of the explanation is: You can only be happy if you accept that something is never perfect and will not last forever.

And so

wabiForth is not finished or perfect

and that is OKAY!

The wabiFORTH system – cooling

The wabiFORTH system is based on a Raspberry 3b+. A Raspberry is the easiest and cheapest way of getting access to a fast processor coupled to a large memory. The other hardware on the Raspberry (USB, network-connection, Bluetooth etc.) is irrelevant for wabiForth and not used.

Minimum wabiForth system:

1. Raspberry 3b+ with an excellent cooling solution
2. Power supply unit 5v, 2.5A or better
3. serial to USB conversion cable
4. SD-card with wabiForth

A good power-supply is critical for a reliable Raspberry. But rather than the overall power, it is more useful to focus on the delivered voltage. Even a very short dip below 5v will result in a reset or crash. If this happens, try to raise the voltage of the power-supply. Aim for 5.2-5.3 volt.

STEP 1: ADD COOLING – COOLING IS ESSENTIAL

A Raspberry running Linux does not need cooling of the CPU. This is because Linux constantly measures the temperature of the CPU, and lowers the CPU-frequency if it gets too hot. In addition Linux switches off unused CPU-cores and other unused parts of the CPU like the floating point co-processors.

WabiForth does the opposite

It overclocks the CPU, runs 3 of the 4 cores at 100%, switches 'on' all parts of the CPU and at all times, even when waiting for user-input, generates high memory-traffic. And to add insult to injury: wabiForth does not check the temperature of the CPU.

And so:

A Raspberry running wabiForth must be cooled!

Add a large cooling-element to it. Ideal are the cooling cases of Joy-It. Or, even better, heat-pipe based cooling. Larger cooling fins for SMD chips also work, but only just. The cheap 14x14mm cooling fins sold everywhere are inadequate!

Examples of good cooling solutions for the CPU of a Raspberry

STEP 2: CONNECT THE REST

The Raspberry is connected via the serial-to-USB cable to a PC. On the PC a terminal program serves to communicate with the Raspberry. In the terminal-program choose **115200-8N1** as protocol for the setup for the connection, resulting in a speed of 115.2 Kb/s.

A monitor can be connected to the Raspberry using a HDMI cable.

The video-output is fixed @ 1024x768 pixels, for a homecomputer user of the 80s a dream-resolution. The colour-depth per pixel is 24 bits.

STEP 3: LOAD WABIFORTH ON SD

Load the images in the top-directory of the sd-card, put the sd-card carefully in the Raspberry and you are done.

OPTIONAL STEPS - BETTER DO THIS LATER

Optional extra's

Build in in wabiForth are I2C drivers for a 20x4 character LCD-screen, 384 kB eeprom and a RTC. Obviously the user can develop other drivers, for I2C or SPI or anything else. **Project Forth Works** on GitHub has some interesting examples for other I2C and SPI drivers.

Operating-system

There is no operating system in the wabiFORTH-system. No Linux, no Android, no RTOS, no nuthink...

After the setup of the CPU during the boot-proces, there are 3 tasks running in the background:

1. the input-buffer system of the UART, mainly on core 2 with core 0 handling user-aborts.
2. the graphics-system, called PIXEL, on CPU core 1
3. the sound-system management, called WAVE, on CPU core 2

All other functionality normally provided by an OS is either handled by wabiFORTH (for instance an I2C interface), is provided/programmed by the user, or is not possible.

ACCESS RIGHTS

An important goal during the development of wabiFORTH was to give the user the fullest possible access to all parts of the ARM-processor. This makes wabiForth an excellent system to experiment with an ARM Cortex-a53 processor.
There is NO management of access-rights, not in any way. If you want to write a value in a CPU setup-register you can. The system might crash, but wabiForth allows you to try.
The only part of the system which is not available directly to the user are the registers related to the setup and management of the ARM Hyper-mode. The rest, which is around 99.9% of the CPU, is open.
It is, by the way, possible to program a Hyper-mode management system with wabiForth. That is left as an interesting little exercise for the user, but maybe not as an exercise for an absolute beginner.

MULTITASKING

There is no multi-tasking. This has to be provided by the user. It is a tentative intention to add this functionality in future, but this is not guaranteed.

Starting, aborting and exiting wabiFORTH

Starting wabiFORTH

Wabi starts by switching the power 'on'.

Reset of wabiFORTH

COLD restarts wabiFORTH without resetting the CPU. Most system-settings are overwritten and any previous program is not present anymore, but memory is not cleared by COLD. Using DUMP you can see the remnants of old stuff after using COLD.
The settings for the compiler and the screen are not reset. This allows the programmer to experiment with the effect of the settings of the compiler on the Sieve benchmark.

A reset of the CPU is done with **RELOAD. RELOAD** resets every part and every core of the system in accordance with the architectural specifications of ARM and Broadcom. It then reloads wabiForth from the SD-card. Any program and any data in memory is lost and all system settings are re-written.

Rebooting by using the boot-vector at address 0x0 (by typing '**0 execute**') restarts core 0. Core 1 and core 2 are left as they are. So this is a rather insecure way of resetting, but it does function. You can overwrite the vector at address 0x0 if you want to.

Exit of a running program

Pressing the keys 'fn', 'ctrl' and 'backspace' at the same time, on an Apple OSX computer using 'Terminal', will abort a running wabiForth program and return wabiFORTH to where commands can be typed in.
The memory is not cleared by this abort, giving you the chance to check values or restart the program or whatever else you want to do.
If 'fn', 'ctrl' and 'backspace' does not function then something is seriously wrong with wabiForth. Or you are not using an Apple OSX computer, which is the same. In that case only a hard reset, by switching the system 'off' and 'on', will help. Any programs and data in the system will be lost.

Due to the 'debug-over-reboot' feature of the ARM-CPU, in rare cases it is necessary to switch off for 10-20 second or so.

Aborting a running program from within the program

Aborting a running program is done with the words **ABORT** or **ABORT". ABORT** exits a running program immediately, giving control back to the user.
ABORT" takes a values from the stack. If the value is not zero, it prints the string between the quotes and aborts the program.

Exit of wabiFORTH

is done by switching it ‘off’. The classic word '**BYE**' is available but has a peculiar will of its own and thus its use is discouraged (try it...).

COLD (--) restarts wabiForth and resets most of the system-variables. The compiler optimisation settings and the screen settings are not reset.

RELOAD (--) restarts the Raspberry Pi, reloads the firmware of the Raspberry and reloads wabiForth. (**REBOOT** is a synonym)

ABORT (--) immediately exits a running program and puts control back with the user. It does not clear memory, allowing the user to inspect things like variables etc.

ABORT" (flag --) ABORT" is followed by a string ending with a '"'. If it finds a 'true' flag on stack it prints the string between the quotes and exits a running program. The message can be used by the user to signal where the abort occurred and what the reason for the abort was. **ABORT"** only function inside a definition.

BYE (--) has no helpful functionality. Where would you go? Outside of wabiForth there is just bare emptiness.

Limits in the wabiFORTH system

WabiFORTH is a 32 bit system and technically has access to the first 4 GB of memory in the system.

maximum size dictionary	903 MB
maximum size allotted memory within the dictionary	any size within the available memory
maximum size string	strings have a 32 bit index and can use all available memory
maximum size of a single word/definition	32 MB per definition. The system might, but must not, become unstable if a definition is larger than 32MB. This is irrelevant in normal coding, but could be relevant in automatically generated code.
maximum number of definitions in the dictionary	between 2079677 (worst case) and 2446677 (best case). This equates to around 200-400 MB of code, which should be enough for most applications. If a larger hash-table, with space for more definitions, is required, please contact the author to negotiate a price for a customised system.
maximum number of vocabularies in the dictionary	Up to 255 vocabularies, known as word-lists in the ANSI-standard, can be defined in the system. Upto 8 can be put in the search-order-list at the same time.
depth of stacks	There are four stacks in wabiForth: data, return, cpu-return and floats. Each stack contains 2048 cells.

Hardware specific wabiFORTH limitations:

The words **2@** (two-fetch) and **2!** (two-store) take an address as argument. The limitation is that this address must be word-aligned. The CPU will generate a data-abort if it is not. Ensuring that the address is aligned is the responsibility of the programmer. It is possible to construct versions of these words which are able to handle unaligned addresses, but the resulting code is somewhat slower than the present version. In the chapter describing the assembler, examples of these words can be found which do not need aligned addresses.
Please note that **@** (fetch) and **!** (store) can handle unaligned addresses. This is, for instance, useful for character and half-word based arrays.

2@: (addr -- double) gets a double from address. The address on the stack for the word **2@** must be word-aligned. The CPU will generate a data-abort if the address is not aligned.

2!: (double, addr --) stores a double at address. The address on the stack for the word **2!** must be word-aligned. The CPU will generate a data-abort if the address is not aligned.

Assembler related limitations

Several opcodes in the ARMv8 assembler have specific limitations with regard to address-alignment. That is how the ARM-cooperation designed them. For specifics see the ARMv8 manuals from the ARM-cooperation.

Overview of dictionary

The wabiFORTH dictionary, like most Forth-implementations, is a linked list of definitions. Listing words in the dictionary can be done using the following words:

WORDS: (--) lists all words in the dictionary and prints the total number of definitions in the dictionary. Hidden words are not shown. Prints the definitions sorted by wordlist in the search-order.

LWORDS: (--) prints out the most recent words. The number of words to be printed is contained in the value **#WORDSLW**. The default is 50. Prints the definitions sorted by wordlist in the search-order.

AWORDS (<"ccc"> --) prints all the words starting with the character specified after AWORDS. For instance: typing **"AWORDS d"** will show all words starting with 'D' or 'd'. **AWORDS** prints all words irrespective of word-lists.

WORDSL: (--) prints out a list of all words in the dictionary including the execution-token and the inline-length. A word can be inlined if the inline-length is larger then zero. A word will be inlined if inlining is switched 'on' and the inline-length of the word is not larger than the length contained in the variable **MAXINLINE**. The default behaviour is that primitives up to a length of 8 opcodes are inlined. Longer primitives are not inlined by default, as the speed-advantage is small or non-existent on the processor used in de Raspberry 3b+.

WORDSCOMMA: (--) prints a comma-delimited list of all words in the dictionary, including the execution-token, the inlining-length and the length of the name. This list can easily be imported into a spreadsheet.

LWORDSL: (--) combined functionality of **WORDSL** & **LWORDS**

LWORDSCOMMA: (--) equivalent to **WORDSCOMMA**

WORD#: (-- n) returns the total number of definitions in the dictionary, including redefined definitions and user-definitions.

#WORDSLW: (-- n) The value #WORDSLW contains the number of words to be printed by the word LASTWORDS. By default this is 50 but it can be changed by the user.

If you for instance enter:

```
10 TO #WORDSLW
```

LASTWORDS from hen onwards will print the last 10 words.

FINDING DEFINITIONS

Printing the name of definition

Normally if you want to know the execution token of a definition you use ' (TICK). But what if you have an XT and you want to know which definition belongs to it? One way is to use **.XTNAME.**

More complicated is the case where you only have an address and you want to know whether it is part of a definition and, if 'yes', from what definition. For this you can use **'ADDR>XT'.** This word, for a given address, finds the corresponding XT or returns a 'zero' if not part of any XT.

The words **.XTNAME** and **ADDR>XT** are handy when writing debugging-routines.

The following examples shows two definition:
.MYSELF prints the name of definition from which it is called.
.BYWHOM which prints by which definition was called.

```
: .MYSELF r14@ addr>xt .xtname space ;
: TST1 .myself ;

: .BYWHOM ( -- )
  r13@ 4+ @
  addr>xt dup ." called by: " .xtname
  space ." with XT: 0x" .hex ;

: WHO? cr .myself .bywhom ;
: TST2 cr .myself .bywhom who? ;
```

If you execute **TST2** you will see that it reports it was called by **INTERPRET,** and **WHO?** will report it was called by **TST.**
The definitions **.MYSELF** and **.BYWHOM** rely on **R13@** and **R14@.** Words which get the present value of R13, the CPU return-stack pointer, respectively R14, the CPU link register. They are explained elsewhere in this document.

.XTNAME (xt --) given a valid XT, prints the name of a definition. If there is a 0x0 on the stack, **.XTNAME** will print "not part of the dictionary".

ADDR>XT (addr -- XT/0) for a given address, finds the corresponding valid **XT,** or returns 'false' if the address is not part of the dictionary.

Size statistics of the wabiFORTH system

The wabiFORTH system consists of 4 main parts:

1. System-data: a block of memory used by the system. It contains internal variables, constants and strings. It also contains the FORTH source-code which is loaded during the boot-process. This block starts at **0x8000** and stretches till **STARTDICT.**
2. Base-dictionary: this is where all assembly code resides, including the primitives and the code for the setup of the CPU and other hardware. Parts 1 and 2 together form a basic FORTH system able to run independently. The base-dictionary starts at **STARTDICT** and ends at **STARTFREE.**
3. Tools-dictionary. This part of the dictionary is defined in Forth and adds useful functionality like drivers and tools. It is where wabiFORTH is defined in its final form. For instance the I2C drivers and most date and time related words are defined here. And the assembler is mostly defined here. This part of the dictionary is compiled real-time during the booting-process using **LOADBOOT. LOADBOOT** loads the source code contained in System-data.
   This part of the dictionary starts at **STARTFREE** and ends at **ENDBOOT.**
4. User-defined definitions. They occupy the memory starting at **ENDBOOT** upto **HERE.**

The following words return statistics on the size of different parts of the system.

SZSYSDATA: (-- n) size of the internal block of memory.

SZBASEDICT: (-- n) size of the base-dictionary.

SZTOOLDICT: (-- n) size of the tools-dictionary.

SZUSERDICT: (-- n) size of the user-part of the dictionary.

SZSYSTOTAL: (-- n) size of wabiForth system.

SZDICT: (-- n) size of dictionary.

UNUSED: (-- n) returns available memory in bytes available for new definitions and memory-blocks in the dictionary. At the time of writing, a freshly rebooted system has around 900 MB available for new dictionary definitions and data.
In addition there is a text block which has an additional 16 MB available to the user.

.AVSIZEWORD: (--) prints the average number of opcodes (1 opcode is 4 bytes) in a word. The average is based on all words in the dictionary. It is for informal purposes only as it is not a very reliable statistic. This is because allotted blocks of memory are also part of the dictionary. If a more accurate statistic is important, then use **ALLOCATE** instead of **ALLOT** to define blocks of memory. Allocated blocks of memory do not influence the calculation of **.AVSIZEWORD.**

DO...LOOP & FOR...NEXT

DO...LOOP

WabiForth follows the ANSI standard: **DO, LOOP, ?DO, +LOOP** and the indexes **'I', 'J'** and **'K'** are available and **LEAVE, UNLOOP** and **EXIT** can be used.

WabiForth specifics

WabiForth introduces **!DO (exclamation-DO).** **!DO** does the opposite of **?DO:** if the two input-values are equal **!DO** will **ABORT.**

!DO (n m --): same function as **DO.** But aborts on equal input-values.

This 'early abort' principle is very helpful in finding bugs early.

Limits of DO...LOOP in wabiForth

The distance of the jump back from **LOOP** to **DO** must be smaller than 32MB. For normal programming not an issue, but it could be an issue with algorithmically generated computer programs.

LEAVE, if used, must be located within the **DO...LOOP.** It cannot be part of a word called from the **DO...LOOP.**

The DO...LOOP is highly optimised

The **DO...LOOP** is the most optimised part of the wabiForth system. Two CPU-registers are dedicated permanently to the **DO...LOOP.** This in itself is faster, but also allows for additional optimisation with inlining and compounding.
The following example compares moving a memory-block from source to a destination with **DO...LOOP** and with **FASTMOVE. FASTMOVE** uses the fastest method according to the ARM-corporation. Obviously it is faster but the **DO...LOOP** comes close!

```
align here constant tstdest   1000000 allot
      here constant tstsource 1000000 allot

: fillsource 250000 0 do    \ random numbers in datablock
    rndm32 tstsource i 4* + !
  loop ;

: tst1  250000 0 do
    tstsource i 4 * + @     \ move 1 word at a time
    tstdest   i 4 * + !
  loop ;

: tst2 tstsource tstdest 250000 fastmove ;

fillsource t[ tst1 ]t.
fillsource t[ tst2 ]t.
```

version	longest execution time seen (microseconds)	relative to MOVE	bandwidth (MB/s)
DO...LOOP	~2350	131%	850
FASTMOVE	~1800	100%	1110

The theoretical maximum total bandwidth of the memory-system of the Raspberry 3B+ is ~1500 MB/s. As the PIXEL graphical system occupies around 350 MB/s of the bandwidth, this means that we are pretty close to the optimum.
If you repeat the measurements a couple of times you will see that the elapsed time is not very stable. Especially the FASTMOVE version varies quit a bit. The high occupation of bandwidth is probably the cause.

Please note that using:

```
create tstdest   1000000 allot
create tstsource 1000000 allot
```

to define the two memory-blocks would result in slower code as this creates non-inlinable words to put an address on stack. Normally not noticeable, but clearly so when you do 250.000 loops with a **DO...LOOP**, where these two words are put on stack every loop.

DO...LOOP can be combined with **AFT/THEN**. See **FOR...NEXT** for an example. This is fun to try, but non-conformant with the ANSI-standard. And so hinders exchange of source-code with other Forth-versions.

FOR...NEXT

On small systems the **FOR...NEXT** concept is faster than **DO...LOOP**, uses less cells on the return-stack and is easier and smaller to implement. All important advantages.

Reality-check...

On larger CPUs, like the Cortex-a53 in the Raspberry 3b+, testing has shown that the speed-advantage is only true for empty loops. And saving a few cells on the return-stack or a few words of memory is irrelevant if you have hundreds of MegaBytes to work with.

FOR...NEXT - choices to be made

As there is no speed advantage to be gained, wabiForth provides the FOR...NEXT loop mainly for compatibility-reasons. Preferably with as many different Forth-implementations as is possible.

This is not an easy task; **FOR...NEXT** is not part of the ANSI-standard. Standardising **FOR...NEXT** was deemed impossible by the committee, as the various existing implementations differed too much.

There are (at least) two ways of designing a **FOR...NEXT** loop:

1. **PRE loop subtraction**
   At the start of the loop, subtract 1 from the index and check if the end of the index is reached, if not perform a loop. This version does not perform a loop if the input is zero. The first index available is the input-value minus 1.

2. **POST loop subtraction**
   First perform a loop, then subtract 1 from the index and check if another loop has to be done. This version always performs at least 1 loop. The first index available is the input-value and this version does 1 loop more than the input-value.

Please note that for programming reasons the variants can be programmed in different ways but with the same functionality.

Other differences between different Forth-implementations are related to:
- the availability of index **'J'** for a **FOR...NEXT** loop contained in a **DO...LOOP** and visa versa
- combining of **FOR...NEXT** with **WHILE/ELSE/THEN**
- the availability of the words **AFT/THEN**
- the availability of **LEAVE**, **EXIT** and **UNLOOP**
- use of signed positive numbers or unsigned numbers for the index

Take all these differences into account and it is easy to see that the humble **FOR...NEXT** loop has greater complexity than obvious at first sight, and it is easy to understand why a universal standard solution evaded the ANSI-committee.

The wabiForth FOR...NEXT implementation

WabiForth supports both variants of **FOR...NEXT: FOR_pre** and **FOR_post.**

The following features are available in wabiForth for **FOR...NEXT:**
- interactive choice of the two variants during compilation
- The index **'J' is** available in nested loops. This is true for **DO...LOOP** and **FOR...NEXT** in any combination
- **UNLOOP** and **EXIT** can be used

In addition:
- **AFT/THEN** and **LEAVE** are available in the **FOR_post** variant
- **WHILE/ELSE/THEN** is available in the **FOR_pre** variant

Choosing the FOR...NEXT variant

Both variants are available in wabiForth. Both principles have advantages and disadvantages. During compilation two words govern which variant is compiled:

FOR_post (--): immediate - The **FOR...NEXT** loop functions as in eForth: The index starts at the input-value, and a loop is always done at least once. **LEAVE** and **AFT/THEN** can be used, **WHILE/ELSE/THEN** not.

FOR_pre (--): immediate - The **FOR...NEXT** loop functions as in noForth. The index starts counting at the input-value minus 1,

and the loop will not run on an input of zero. **WHILE/ELSE/THEN** is available. **LEAVE** and **AFT/THEN** are not.

Changing which **FOR...NEXT** variant has effect on the next **FOR...NEXT** loop(s) to be compiled. Existing loops, and loops still being compiled are not influenced.

Compatibility with other Forth-implementations

The two variants together should allow compatibility with most Forth-implementations.
The FOR_pre variant is fully compatible with noForth.
The FOR_post variant is, for instance, compatible with eForth and iForth.

Please note: LEAVE from **FOR...NEXT** is not always available in other Forth-implementations; do not use it if you want to be compatible with other implementations.

Example of the effect of switching the **FOR...NEXT** system:

```
FOR_pre
: for_v1 cr 5 for i . next ;

FOR_post
: for_v2 cr 5 for i . next ;

for_v1 ( prints: 4 3 2 1 0 )
for_v2 ( prints: 5 4 3 2 1 0 )
```

Example of **UNLOOP and EXIT** in **FOR...NEXT**

```
: tstf 10 for
    i dup . 5 = if
       unloop exit  \ exits the definition!
    then
  next ." end" ;     \ end is never printed!

tstf ( prints: 10 9 8 7 6 5 )
```

Example of **LEAVE** in **FOR...NEXT**

```
FOR_post
: tstl 10 for
    i dup . 5 = if
       leave        \ exits the loop
    then
  next ." end" ;     \ end is printed

tstl ( prints: 10 9 8 7 6 5 end )
```

Use of AFT & THEN

As far as I can find, the word **AFT** was introduced by eForth. In wabiForth it is included for compatibility reasons. It is one of those concepts which grows slowly on you, or not at all. It *is* a powerful word. In the book: "eForth Overview" by dr. C.H. Ting there are several interesting examples of the use of the word.

AFT (--): immediate - exchanges a destination from R-stack with a destination pointing to just after **AFT**. And adds a new destination on R-stack pointing to **AFT** itself.

This manipulation of destinations allows the program to do something different during the first **FOR...NEXT** loop compared to the other loops.

Like this:

AFT and THEN in FOR...NEXT:

```
FOR
  {only executes during first loop}
AFT
  {executes during all other loops}
THEN
  {executes during all loops}
NEXT
```

Please note: in wabiForth, **AFT/THEN** can also be used in the **DO...LOOP**. But ecod, a useful purpose eludes me...

Example with FOR...AFT...THEN...NEXT

This short example defines a word '**.A**' (DOT_A) which prints as many A's as the value on stack. If the value on stack is zero, it will print zero A's, like it should. And that without any check if there is a zero as input.

FOR_post

```
: .A ( n -- ) for aft ." A" then next ;
```

Example with FOR...NEXT and WHILE/ELSE/THEN

This example shows the use of **WHILE/ELSE/THEN** in **FOR...NEXT**. **KEY_TIMEOUT** waits for a keystroke and returns the keystroke. If it takes too long an error-message is returned.

Because of the speed of wabiForth a large input-value (100000000) is needed. This looks silly and also is energy-inefficient, as the processor is working very hard while waiting. In the **assembly-examples** chapter there is an example showing how, by adding one opcode, a more efficient wait-loop can be implemented.

```
FOR_pre                \ FOR_post does not support this
: KEY_TIMEOUT
```

```
100000000 for
   key? 0= while
next
    ." time-out"
else
   key emit unloop
then ;
```

MOVE & FILL

MOVE, **CMOVE**, **CMOVE>**, and **FILL** function as described in the ANSI standard.

In addition the following words have been defined:

FASTMOVE (orig, dest, no_of_words --): Moves 32b words from origin to destination. As the name implies, this word is faster than **MOVE** (~22% faster). Contrary to **MOVE**, the origin and destination must be aligned for **FASTMOVE**.

FASTFILL, (addr, no_of_words, n --): Fills the memory starting at address with the 32b word n, no_of_words times. As the name implies, **FASTFILL** is faster than **FILL**. Address can be non-aligned but then the speed advantage is usually lost.

Store & fetch

Wabiforth provides all the standard words to store and fetch data from memory and variable. The standard words **!, @, 2!, 2@, C!, C@, +!** function as defined in the ANSI-standard.

In addition wabiForth supports some additional words.

H@ (addr -- 16b): fetches a 16bit value from memory.

H! (16b addr --): stores a 16b value in memory

SH@ (addr -- 16b): fetches a 16bit value from memory and extends the sign-bit to 32b.

SC@ (addr -- 16b): fetches a 8bit value from memory and extends the sign-bit to 32b.

C+! (n addr --): raises the 8it value at address with 8bit value n - idea: W. Baden

M+2! (n addr --): raises the double at address with the single value n

D+2! (d addr --): raises the double value at address with the double value d

C@+ (addr -- addr+1, c): fetches a 8bit value from address and raises the address with 1. This is equal to COUNT on most small systems.

SC@+ (addr -- addr+1, sc): fetches a 8bit value from address, extends the sign-bit to 32bit and raises the address with 1.

H@+ (addr -- addr+2, h): fetches a 16bit value from address and raises the address with 1.

SH@+ (addr -- addr+2, sh): fetches a 16bit value from address, extends the sign-bit to 32bit and raises the address with 1.

@+ (addr -- addr+4, n): fetches a value from address and raises the address with 4. This is the same as COUNT in wabiForth. Using **@+** to go thru an array is preferable as it makes specifically clear what is being done.

User Input

User input is via the serial line at GPIO 14 and 15, and technically is handled by the WAVE-loop running on core2. This loop polls the UART for a character 44100 per second. Received characters are put into a circular buffer with a size of 1 MB.
The effect is a system which is very resilient against buffer overruns. You can find definitions to manage the input-buffer below.

This entry-system forms the basis for the normal Forth user-input words.

Adherence to the ANSI standard:
KEY, KEY?, EXPECT, >IN, and **TIB** work as described in the ANSI-standard. **ACCEPT** works as described in the standard but has as additional feature the handling of 'esc'.

ACCEPT (addr len_max -- len_string): Accepts len_max characters input from the keyboard. Entry ends by a 'return' or an 'esc'. On pressing 'esc' entry ends immediately and a 1 character string is returned with the value 27 as the first character.

The following additional words are available

BACKSPACE (--): executes the sequence: 8 EMIT 32 EMIT 8 EMIT

The following words are available to reset and get the status of the UART entry-buffer.

UARTCLEAR (--): fills the UART entry-buffer with zeroes and resets the position of the read and write pointers

CHARSINBUF (-- u): returns the number of characters left in the input-buffer

UARTRWRITE (-- addr): variable - contains the position of the write-pointer of the UART entry-buffer. This variable is used by the entry-system and changing the value probably results in strange behaviour

UARTRREAD (-- addr): variable - contains the position of the read-pointer of the UART entry-buffer. This variable is used by the entry-system and changing the value likely results in strange behaviour

String handling

String handling is an important area of WabiForth. Strings in wabiForth can be large, they have a 32 bit index and thus can use the whole memory. And the handling of strings in wabiForth is fast, all primitives are written in optimised assembly.

The words **C@, C!, FILL, C, (c-comma), MOVE, CMOVE, CMOVE>, FILL, ." (dot-quote), S" (s-quote), SEARCH, COMPARE, /STRING (slash-string), -TRAILING, CHAR, [CHAR], CHAR+** and **CHARS** function as specified in the ANSI-standard.

SLITERAL interpreting: **(addr len ... <ccc> -) -** exec: **(-- addr len)**
SLITERAL in wabiForth presently does not follow the ANSI standard. In wabiForth it is used during interpretation. It is placed after a string defined with S"**...**" and is followed by a name. Calling the name during execution will put the address and length of the original string on stack, allowing direct typing or other string manipulations.
The ANSI standard explicitly forbids changing this string, but wabiForth does not guard against this. Complying with the standard is a responsibility of the programmer.

In addition the following words have been defined:

>CAPS (char -- CHAR): converts a character in the range 'a'-'z' to 'A'-'Z'

>LOWER (char -- CHAR): converts a character in the range 'A'-'Z' to 'a'-'z'

$>UPPER (addr len --): converts the string defined by addr and len to all capitals characters.

$>LOWER (addr len --): converts the string defined by addr and len to all lower case characters.

STRING<> (addr1 len1 addr2 len2 -- f/t): compares 2 strings and gives a true if the strings are *not* the same.

Here an example of the use of SLITERAL and UPPER.

```
s" a string without capitals" sliteral mystring
: show mystring type ;
: makecaps mystring $>upper ; \ not ANSI compliant

show
makecaps
show
```

You will see that the string is first printed in lower case and then in upper case. The string is changed permanently. Using '**$>lower mystring**' will change it back to all lower characters.

Memory allocation

WabiForth largely follows the ANSI-standard with regard to memory allocation. The main difference are the words **FREE** and **RESIZE.** In wabiForth these two words only work on the last block defined. In wabiForth there is no difference between a buffer and a block of memory allocated.

Both **FREE** and **RESIZE** are protected by a CRC-code. The words will only take effect if you give the exact start-address of a block as input. Using a wrong address will result in an error-code on stack. A programmer can use the words as often as needed without fear of a memory leak.

ALLOCATE (u -- addr t/f) Allocates a block of memory at least u bytes long. The block defined will always be 64 byte aligned, to optimise cache-allocation. If the allocation was successful, a valid address and 'false' are returned. If there was not enough space, a true-flag will be returned with an invalid address.
ALLOCATE works from high to low memory, and all empty memory space is available.

FREE (addr -- f/t) releases a block of memory previously defined by **ALLOCATE** or changed by **RESIZE. FREE** can be used on any blocks defined, but blocks more recently allocated will also be FREEd.

RESIZE (addr - f/t) in wabiForth resizes a block of memory and moves the content to the new starting-position. If a block is reduced in size, data falling outside of the new size will be deleted without further warning.
Normally only the lowest (ie. most recently defined) block in memory can be resized. If you resize a block of memory which is not the lowest (ie most recent), it will be the lowest after resizing. Words below the word being resized will be deleted without warning.

#To Be Done:

BLCKSIZE? (addr - max_size) Returns the maximum size of the block at addr. If 'TRUE' is returned an error occurred.

BLCKFILL (u addr - f/t) Fill the memory block at addr with U. Return a flag with '**FALSE**' signifying succes and '**TRUE**' signifying an error.

BLCKCOPY (addr u u addr u), BLCK

Example:

```
: CHECK? abort" problem with memory-block" ;
```

CHECK? checks if a memory-block action was successful and aborts on an error.

```
100000 ALLOCATE check? value MYBLOCK
```

Allocates a block of 1 million bytes and puts the address in value MYBLOCK.

20000000 myblock resize check? to myblock

Resizes the memory-block MYBLOCK from 1 to 20 million bytes

500000 myblock resize check? to myblock

Resize the memory-block again, but now to a smaller size

myblock FREE check?

Frees the memory occupied by the memory-block MYBLOCK

Wordlists (a.k.a. vocabularies)

ANSI introduced the more generic term '**WORDLIST**' for vocabularies.

WabiForth supports the following standard words: **WORDLIST, FORTH-WORDLIST, SEARCH-WORDLIST, FORTH, DEFINITIONS, ALSO, ONLY, PREVIOUS, ORDER, SET-ORDER, GET-ORDER, SET-CURRENT** and **GET-CURRENT.**

These words follow the ANSI-standard. Implementation-specific details are the following:

PREVIOUS: When an attempt is made to remove a word-list from an empty search-order list, PREVIOUS will be ignored and no error-condition will occur.

ONLY: According to the ANSI-standard the minimum search-order is implementation dependent. In wabiForth the minimum search-order includes all system-defined words. This is done by ensuring that FORTH is always searched at least once.

In addition the following definitions are available.

VOCABULARY: (<name> --) The word vocabulary defines a new vocabulary with the name <name> which follows the word VOCABULARY.

.VOCS: (--) Prints an overview of all vocabularies defined. Also prints vocabularies which are not in the search-order.

Example: Enter the following:

```
order
vocabulary NEWVOC
also newvoc
definitions
order
```

And you will see the following:

```
(0 0 0) order
definitions :
        FORTH
search order:
        FORTH ok
(0 0 0) vocabulary NEWVOC ok
(0 0 0) also newvoc ok
(0 0 0) definitions ok
(0 0 0) order
definitions :
        NEWVOC
search order:
        NEWVOC
        FORTH ok
```

ORDER shows the present search-order. Than **VOCABULARY** is used to make a new vocabulary with the name **NEWVOC. ALSO NEWVOC** adds **NEWVOC** to the top of the search-order list. **DEFINITIONS** ensures that new definition are added to the new vocabulary. Finally

ORDER shows the final situation: 2 vocabularies and new definitions are compiled in **NEWVOC.**

.VOCS will show the two available vocabularies, including their wordlist-identifiers.

```
  wid name
  --- -------
    1 NEWVOC
    0 FORTH
  --- -------
```

Compilation/Execution control

This section of wabiForth was developed with the kind and patient support of Albert Nijhof and Willem Ouwerkerk.

The words **[IF]**, **[ELSE]**, **[THEN]**, **REFILL**, and **PARSE** function as specified in the standard. In addition **[IF** is available.

[IF: [IF allows for conditional compilation based on categories.

[IF_VARIANT: is a VALUE which contains the categorie to be executed or compiled.

As the example shows it can be used while interpreting. This example shows the effect of different category selections

```
char A to [if_variant
[IF AC]    1 [ELSE] 0 [THEN] .
[IF CA]    1 [ELSE] 0 [THEN] .
[IF B]     1 [ELSE] 0 [THEN] .
[IF BCEFD] 1 [ELSE] 0 [THEN] .
[IF 13%A]  1 [ELSE] 0 [THEN] .
[IF ]      1 [ELSE] 0 [THEN] .
```

And it can be used for conditional compilation:

```
char a to [if_variant
: test1 [if BCEFD] 1 [else] 0 [then] . ;

char c to [if_variant
: test2 [if BCEFD] 1 [else] 0 [then] . ;

test1 -> 0
test2 -> 1
```

As you can see in the examples above, the categories used are NOT case-sensitive.

CATCH THROW

Wabiforth implements **CATCH** and **THROW** as described in the ANSI standard. The examples in the ANSI documentation function as described.

Please note:
The implementation of both CATCH and THROW is experimental and might be changed or even removed in the future if long-term stability of wabiForth is not as required due to CATCH and THROW.

Contrary to some other Forth-implementations, CATCH/THROW is not used for an ABORT. And there is no last resort CATCH defined in QUIT.

Stacks

The ANSI standard contains 2 words giving information on a stack: **DEPTH & FDEPTH.** But there are several other related or common words with which the programmer can get information on the stacks and their status. Some of these are even mentioned and used in the ANSI standard. For instance **SP@** and **SP!** which are used in the example sources for **CATCH** and **THROW** of the ANSI-standard.

This is the set of such words in wabiForth:

DEPTH (-- n): puts depth of the data stack on stack as it was before the depth was put on stack

FDEPTH (-- n): puts the depth of the floating point stack on stack

CPUDEPTH (-- n): puts depth of the CPU return-stack on stack. WabiForth has separated stacks for the return: the user-stack and the CPU-stack.

USDEPTH (-- n): puts depth of the user return on stack. WabiForth has separated stacks for the return available to the user and for the CPU. USDEPTH returns the depth for the user-stack.

SP@ (-- addr): fetches the address pointed to by the stack pointer

SP! (addr --): puts the address on stack in the data stack pointer

RP@ (-- addr): fetches the address pointed to by the return stack pointer

RP! (addr --): puts the address on stack in the return stack pointer

FP@ (-- addr): fetches the address pointed to by the floats stack pointer

FP! (addr --): puts the address on stack in the floats stack pointer

R13@ (-- addr): puts the content of CPU register 13 on stack. In the chapter '**Overview of Dictionary**' R13@ is used in an example.

R13! (addr --): puts the address from the stack in CPU register 13

R14@ (-- addr): fetches the address contained in register 14 of the CPU. ***This word can only be used in interpreting mode!*** Register 14 is the link-register of the ARM-processor.

[R14@] (-- addr): fetches the address contained in register 14 of the CPU. ***This word can only be used within a definition!*** Register 14 is the the link-register of the ARM-processor.

R15@ (-- addr): fetches the address contained in register 15 of the CPU. Regsiter 15 is the program counter of the ARM-processor.

GOTOADDR (addr --): continues processing at the address on stack, this word directly forces the ARM-processor to jump to the address on stack. **GOTOADDR** influences both register 14 and register 15, which is why there are no R14! and R15!. GOTOADDR is a lot like the good old GOTO in the BASICS of the 80s. It is possible to write horrendous spaghetti code with it!

System constants, variables and values

The wabiSystem depends on a range of variables, constants and tables to organise a correct flow of actions and to keep track in general. Some of these are made available to the programmer. For instance if you are interested in the internals of the system. Or if you want to add new functionality.

LAST (-- addr): variable, contains the address of the link-field of the most recent word in the dictionary

TRANS (-- addr): constant, contains base-address of buffer for trans-area

TRANSNUM (-- addr): constant, contains base-address of buffer for nummer-conversion

WORD_LIST_COUNT (-- addr): variable, tracks overall number of wordlists defined - cannot be larger than 255

SEARCH_ORDER_TABLE (-- addr): constant, start-address of the table with the search_order lists

FREENOCACHEMEM (-- addr): constant, start-address of the unoccupied part of a uncached memory-block.

NOCACHEDATA (-- addr): variable, pointer into no_cached memory block

FREECOHERENTMEM (-- addr): constant, contains start-address of the unoccupied part of a coherent memory-block

COHERENTDATA (-- addr): variable, pointer into coherent memory block

FREEUNSECUREMEM (-- addr): constant, start-address of the free/ available part of a unsecure memory-block. Unsecure is a term related to the ARM-processor.

UNSECUREDATA (-- addr): variable, pointer into available unsecure memory.

UARTREC (-- addr): constant, start of UART receive buffer

UARTRWRITE (-- addr): variable located in coherent memory-block, contains the location within the UART receive buffer where the next character received will be stored in the buffer.

UARTRREAD (-- addr): variable located in the coherent memory-block, contains the location within the UART receive buffer where the next character will be read from the buffer.

UARTRECLEN (-- addr): constant, contains length of the UART receive buffer

Time and time-measurement related words

WabiFORTH contains two 64bit counters. One, the micro-second counter counts up with a frequency of 1 MHz. The other is the CPU-cycle counter, which counts up with a frequency of 1.5 GHz (1.4 Ghz for Willem Ouwerkerk). Both start counting from zero after a full reboot of the system.
The micro-second counter is meant as a general purpose time-measurement tool. The CPU-cycle counter is especially useful when optimising the performance of algorithms, as it counts the actual number of CPU-cycles needed to execute an individual routine.

THE MICRO-SECOND COUNTER

is accessed with the word **DMCS**. **DMCS** returns the time in microseconds since the last reboot of the system, as a double. The underlying counter in fact counts at 5.6448 Mhz to get better rounding-properties for **DMCS**. In theory it takes more than 100.000 year before this underlying counter wraps around to zero. In reality this counter is implemented in hardware with 56 bits. Which gives a wrap_to_zero time of something like 400 years. Still not a problem in normal situations.

A related word is **MCS**, which does the same as **DMCS** but returns a 32bit word.

DMCS (-- d): returns the elapsed time in microseconds since the last reboot, as a double. It does so by dividing the underlying counter with 5.6448 to get at microseconds. **DMCS** is a primitive word and takes ~17 cycles to put an answer on the stack.

MCS (-- u): returns the elapsed time in microseconds since the last reboot, as an unsigned single. **MCS** wraps around to zero after 4295 seconds, so care should be taken that this wrap-around does not impede on interval calculations.

MS? (-- u): returns the elapsed time in milliseconds since the last reboot, as an unsigned single. **MS?** wraps to zero after 49.7 days.

Please note the question mark! **MS** is a ANSI-standard word which waits for a given number of milliseconds.

MS (n -): wait n milliseconds.

CENTIS (-- u): returns the elapsed time in centi-seconds since the last reboot, as an unsigned single. CENTIS wraps to zero after 497 days.

MCS, MS? and CENTIS are inlinable primitives, and take ~12 cycles to put an answer on the stack.

SEC (-- u): returns the time in seconds since the restart of the system, as an unsigned single. **SEC** wraps around to zero after ~17 years. SEC is a primitive and takes ~16 cycles to put a result on the stack.

THE CPU-CYCLE COUNTER

is accessed with the word **CPUCYCLES.** It starts counting at 0x0 after a RESET or REBOOT. The counter counts the actual clock-cycles used by a CPU-core. If a core is temporarily halted, for instance using WFE or WFI, this counter also stops counting. If the frequency of the CPU changes, this clock will count at that changed frequency.

It is important to realise that the measured numbered of cycles will vary greatly. This variability reflects reality. **CPUCYCLES** measures the actual number of cycles taken by whatever is being measured. The state of the different caches, the state of the pipelines and other aspects influence the number of cycles actually required.
If you want to measure the fastest possible time of a routine use '**((**' and '**))**'. If, on the other hand, you are interested in the actual time taken, use **CPUCYCLES.**

CPUCYCLES (-- d): puts the elapsed number of CPU-cycles since the last reboot on stack as a double. It wraps around to 0x0 in about 389 years.

Executing **CPUCYCLES** twice directly after each other will show a difference of (usually) 6 cycles. These 6 cycles are the time required to put the values on the datastack. Subtracting the 6 cycles from a measurement will give you the time time taken by whatever is between two **CPUCYCLES.**

Examples:

```
: MINIMUM cpucycles cpucycles 2swap d- d. ;
: SHOWMIN 10 0 do cr minimum loop ;
```

Executing **SHOWMIN** shows ten times the difference from two consecutively executed **CPUCYCLES.** The shortest difference possible is 6, but values up to 600 or so can be returned. By running **MINIMUM** ten times you can, usually, see the effect of the cache-system and the branch-prediction system.

The Cortex-a53 in the Raspberry 3b+ is one of the few CPUs presently available where the actual number of CPU-cycles required for a given task is non-deterministic. More specifc: there is no way of determining how long a specific task will take on a Cortex-a53. Which is interesting to say the least.

```
: HOWLONG cpucycles 1 1 + drop cpucycles 2swap d- d. ;
```

Executing **HOWLONG** measures the number of cycles it takes to execute **CPUCYCLES** twice plus the cycles needed for '1 1 + drop'. The result printed can be anything from between 9 and around 350 or so. The shortest number of cycles possible is 9. As the 2 **CPUCYCLES** use 6 cycles, the rest, 3 cycles, is the fastest than '1 1 + drop' can be executed.
Any larger measurement than 9 is the result of opcodes not yet being available in the cache-system of the CPU. In that case the execution takes longer. Repeating a measurement is always recommended.

OTHER WORDS FOR TIME-MEASUREMENTS:

.UPTIME: (--): prints how long the system has been running since the last reboot or (re)start. It prints in the format 00h00m00s. WabiForth is normally a very stable system, uptimes of thousands of hours are entirely possible. **.UPTIME** can track the uptime up to ~400 years.

Interval measurements

For measuring an time-interval the following words are available:

T[(-- dmcs): puts the present time in mcs as a double on stack

]T. (dmcs --): calculates the difference in mcs with the last T[and prints it in a formatted way

For instance:

```
: TST 100000 0 do i drop loop ;
t[ tst ]t.
```

prints "340 mcs" or something like that. In other words: putting I on stack and dropping it again one hundred thousand times takes around only 340 micro-seconds.

C[(-- dcpucycles): puts the present number of cpucycles as a double on stack

]C. (dcpucycles --): calculates the difference in cycles with the last C[and prints it in a formatted way

For instance:

```
: TST 100000 0 do i drop loop ;
c[ tst ]c.
```

prints "503540 cycles" or something like that.

Measurements with **CPUCYCLES** have a high variability. That variability reflects the reality of a modern CPU, which is very fast on average but unpredictable for the performance of individual pieces of code at any given time.

Measurement of number of cycles

If you want to measure pretty exact how long the execution of a word or short snippet of Forth-code takes at its fastest you can use the words **((** and **))**.

The measurements are done by running the Forth-code between **((** and **))** a million times and measuring the elapsed time to calculate the number of cycles per loop. As the code is called a million times, the cache-system, the branch predictor and the pipe-line system all function optimally. This ensures that the maximum performance of the CPU is measured. Like measuring the

top-speed of a car. In normal software, especially when the rate of dirtying of cache-lines is high, the actual speed is slower, sometimes a lot slower.
Please note that the word or forth-code snippet being measured must be stack-neutral. And that **((** and **))** can only be used inside a compiled word.

(((-- mcs): Initiates the measurement and 1 million loops

)) (mcs --): takes the time from **((** from stack, calculates the number of cycles used and prints this in a formatted way.

For instance:

```
: TST 10 0 do loop ;
: HOWMANYCYCLES (( tst )) ;
```

Executing **HOWMANYCYCLES** prints something like:

```
30300 mcs -> 45.5 cycles
```

The 30300 mcs reported is the time (in micro-second) it takes to call the word **TST** a million times. The 45.5 cycles reported is the number of CPU-cycles it takes on average to call and execute the word **TST** 1 time.
In other words, calling TST one time takes on average ~30 nanoseconds. To make clear how short a time that is: light travels just 9 meters in 30 nanoseconds. In that very short time wabiForth jumps to a subroutine, does 10 loops and returns. Which I find truly awe-inspiring!

Defining a timer

The following code-example defines a timer called TIMERA which meaures microseconds. As many individual timers as needed can be set up this way.

```
2variable timera
: SETTIMERA ( -- ) dmcs timera 2! ;
: GETTIMERA ( -- d ) dmcs timera 2@ d- ;
```

SETTIMERA (--): resets timer A to zero

GETTIMERA (-- d): returns the time in microseconds since timera was last reset, as a double.

The following code shows how this timer can be used. It measures how long it takes to do 10000 empty loops using do...loop:

```
: HOWLONG settimera 10000 0 do loop gettimera d. ;
```

At the time of writing it takes around 14 microseconds (i.e. 14 millionth of a second) to do ten thousand empty do...loops with wabiFORTH.

Pausing execution

Pausing execution for a certain time can be done using the following words:

WAITMCS (n --): waits for n microseconds before continuing with execution. Pausing is done by measuring the elapsed time, and as such is reasonable accurate.

WAIT (--): waits for for 1 second before continuing execution

BLINK (--): waits for 0.1 second before continuing execution

10MS (--): waits for 10 miliseconds before continuing execution

1MS (--): waits for 1 milisecond before continuing execution

MS (n --): waits n miliseconds before continuing execution

GPIO

The Raspberry Pi 3B+ board contains a 40 pin connector for IO (input/output). Twenty-six of these are IO pins available to the user. The usual term for these is GPIO (**G**eneral **P**urpose **I**n/**O**utput), as they have more functions than just IO.
For the most common actions wabiForth offers specific words. Full control of the GPIOs is possible via the CPU-registers.

3.3v	1	2	5v		
gpio2	3	4	5v		
gpio3	5	6	0v		
gpio4	7	8	gpio14		
0v	9	10	gpio15		
gpio17	11	12	gpio18		
gpio27	13	14	0v		
gpio22	15	16	gpio23		
3.3v	17	18	gpio24		
gpio10	19	20	0v		
gpio9	21	22	gpio25		
gpio11	23	24	gpio8		
0v	25	26	gpio7		
HAT id	27	28	HAT id		
gpio5	29	30	0v		
gpio6	31	32	gpio12		
gpio13	33	34	0v		
gpio19	35	36	gpio16		
gpio26	37	38	gpio20		
0v	39	40	gpio21		

The picture on the left shows you that the GPIO-numbers are not the same as the pin-numbers. All wabiForth words always use the GPIO-numbers.

Pin 27 and 28 are used by the Raspberry for HAT-identification (HAT=**H**ardware **A**ttached on **T**op), and are not available to the user; even if no HAT is present.

All other GPIO's can be an in- or output. In addition most GPIOs support other functions. On the next page there is an overview.

Pins 2 and 4 are the 5.0v pins. Connecting these with any other pin, no matter how short, might (and probably will) destroy the CPU!

You can actually use these 2 pins to connect an external 5.0v power-supply. This way of supplying power is called back-feeding and it works fine. It is wise to cover the 5.0v pins with something if they are not in use.

Use of GPIO-pins in wabiForth: wabiForth has default settings for 6 GPIO's. The UART0 on pin 8 and 10 is critical for communication with wabiForth, the functionality should not be changed.
The other pins (I2C and PWM) can be changed to other functions.

pin 8	gpio 14	UART0 - TX
pin 10	gpio 15	UART0 - RCV
pin 3	gpio 2	I2C - SDA1
pin 5	gpio 3	I2C - SCL1
pin 32	gpio 12	PWM - output synthesizer
pin 33	gpio 13	PWM - output synthesizer

WabiForth has words for setting **ALT functions** of a GPIO, for setting and reading a GPIO and for the pull-up and pull-down resistors. Interrupts are not covered by wabiForth itself.

SETFUNCGPIO (n gpio# --): sets the alternate function of port gpio# to function n. Valid inputs are between 0 and 7. When 0 is the input, a pin as configured input, 1 configures a gpio as output. This is true for all GPIO's. The other options are pin-specific. See the ARM-documentation for full details on the other special functions.

Please be careful as the special functions-values have a strange order. These are the inputs for the different functions and ALT functions:

value for input	function
0	input
1	output
4	ALT0
5	ALT1
6	ALT2
7	ALT3
3	ALT4
2	ALT5

If you read the Raspberry manual, the **alternate functions** of the GPIO pins seem daunting at first glans. But most alternate functions are not of any practical use for most programmers.
Below you'll find an overview of the relevant ALT functions. With these you can interface to almost everything.

Simplified alternate functions Raspberry Pi 3B+

ALT5	ALT4	ALT3	ALT0	GPIO	pin	pin	GPIO	ALT0	ALT3	ALT4	ALT5
				3.3v	01	02	5.0v				
			SDA1	gpio2	03	04	5.0v				
			SCL1	gpio3	05	06	0v				
			GPCLK0	gpio4	07	08	gpio14	TXD0			TXD1
				0v	09	10	gpio15	RXD0			RXD1
RTS1	SPI1_CE1	RTS0		gpio17	11	12	gpio18	PCM_CLK		SPI1_CE0	PWM0
				gpio27	13	14	0v				
				gpi022	15	16	gpio23				
				3.3v	17	18	gpio24				
			SPI0_MOSI	gpio10	19	20	0v				
			SPI0_MISO	gpio9	21	22	gpio25				
			SPI0_SCLK	gpio11	23	24	gpio8	SPI0_CE0			
				0v	25	26	gpio7	SPI0_CE1			
				HAT id	27	28	HAT id				
			GPCLK1	gpio5	29	30	0v				
			GPCLK2	gpio6	31	32	gpio12	PWM0			
			PWM1	gpio13	33	34	0v				
PWM1	SPI1_MISO		PCM_FS	gpio19	35	36	gpio16		CTS0	SPI1_CE2	CTS1
				gpio26	37	38	gpio20	PCM_DIN		SPI1_MOSI	GPCLK0
				0v	39	40	gpio21	PCM_DOUT		SPI1_SCLK	GPCLK1

SETGPOUT (true/false gpio# --): sets port gpio# to 'on' or 'off'. The valid range of port-numbers is 0-53. Other values are ignored. The maximum frequency which can be reached is ~3.2 MHz.

This word only has effect for gpio pins configured as output. But the value is stored. If you write 'true' as output to a pin, the pin will go high as soon as it is configured as output.

Please note that the ARM-processor can only source around 2mA per pin! This is much lower than for instance an Arduino. So in most cases you will need a buffer to drive a LED or other effectuators.

GETGPIN (gpio# -- 0/1): gets the input-value at port gpio#. The maximum frequency with which ports can be read is ~4.5 MHz.

GETGPIN gets the actual value of the pin, also if the pin is configured as output. This can be handy when debugging a routine.
The inputs are all sampled, that is the Raspberry does several reads of a port and then a majority-vote determines what level to return. This is far more reliable than just reading a level, but takes longer.

SETPULLUD (action gpio# --): controls the use of pull-up/pull-down resistors setting the GPIO specified by gpio# to action. The pull-up/down feature only has effect for gpio pins configured as input.

The following table shows what input gives which action.

value for input	input
0	no resistors
1	pull-down
2	pull-up

It might be interesting to note that the pull-up/pull-down resistors can be used as high-impedance outputs. In that case the pin needs to be configured as INPUT. And than by setting the pull-up/down resistors to high or low, the pin in fact generates high-impedance output.

WAVE - sound generator

** THIS CHAPTER IS UNDER DEVELOPMENT **

As the sound generator has a lot of options, describing these clearly and in detail is a lot of work.

** THIS CHAPTER IS UNDER DEVELOPMENT **

The Wabi-system contains a flexible sound-generation system. Core 2 of the CPU is dedicated permanently to this. The system generates digital sound at 44.1 KHz and has 2 channels. These can either be used as left and right channel, or as 1 for sounds with a low pitch (ie. the 'bass'-channel), and 1 for sounds with a higher pitch. This depends on the implementation of the reconstruction-filters connected to the outputs of the Raspberry.

On the original home computers, sound-generators had a fixed number of oscillators, 1 or more noise-generators and a fixed number of registers to define frequencies, wave-forms, inter-connections and other such options.

The wabi-system functions in a different way. There is no fixed number of oscillators or other elements. The user has the possibility to define whatever number of oscillator and other elements is needed. Obviously within the technical limits of the ARM processor. As a rule of thumb, around 1500 elemenst can be defined at the same time. Generating a sound using 100 oscillators is perfectly fine.

The way WAVE functions is that the user defines sound-elements, and links these elements together. The defined elements then generate or manipulate the sound.
A sound-element is created by creating a control-block. Such a control-block is nothing more than 8 to 12 words of data describing what an individual sound-element should do. So for instance for a sinus-generator there is a word specifying what frequency the generator should generate.
All the control-blocks are put after each other in a list (called the WAVE-list). The WAVE-sound generator uses this information to generate the specified sound/sounds.

You surely will know the phrase "sheer endless possibilities". Well that is especially true here. A guesstimate of the number of possible different settings for WAVE is a 1 followed by 21250 zeroes (give or take a zero...).

The WAVE sound-generator is a prototype and testing will take a lot of time. But the system is already usable. And what is clear is that it is an exiting tool with a broad spectrum of functionality. As with any new tool, changes and additions can happen in the future.

A quick-start example is probably the easiest way to show the principles of WAVE. The code:

```
resetwv
440 ~sine constant osc1
osc1 osc1 ~output drop
startwv
```

will sound a 440 HZ sinus on both channels. If you want to stop the sound use STOPWV. STARTWV will start it again.

Five words are used here:

RESETWV (--): resets the WAVE generator, empties the WAVE-list and stops all output. Is used before a new setup of WAVE is done.

STARTWV (--): starts the WAVE generator. It is usually done after setup, but can also be done earlier to hear the effect of individual setup-steps.

STOPWV: (--): halts the WAVE generator but does not reset it. Executing STARTWV will restart it from where it was stopped.

~**SINE**: (F -- *out): adds a sine-oscillator to the WAVE-list. F specifies the wanted frequency. *OUT is the address where the output of the oscillator is available. With ~SINE frequencies between 1 Hz - 20KHz can be generated. By putting the address (ie.: *OUT) in a named constant, this output-address is available for other elements to link to.

~**OUTPUT** (ch0 ch1 -- *out): adds an output-element to the WAVE-list. At least 1 output element is needed to hear output. But the system itself will happily generate wave-data without an output module. You just don't hear anything.

The oscillators:

There are presently 11 different oscillators available: 3 sawtooth oscillators, 2 block-wave oscillators, a sine-oscillator, 2 arbitrary wave generators and finally 3 noise-generators. With the exception of 2 noise-generators, all these generators are frequency based. That is, their main defining characteristic, next to the wave-form, is a base-frequency.

The 9 frequency-based oscillators have some properties in common: Each has a base frequency. They have one output each. The phase of an oscillator can be controlled realtime. The frequency can be fixed or be defined by an external source, for instance a sequencer or from wabiForth. And finally all have technically a very wide range of frequencies which can be generated. The technical upper range is in the MHz. And on the opposite side, the maximum period a wave can have, is around 27 hours.

In addition, some oscillators have specific functionality which are explained later.

The sawtooth oscillators:

There are 3 sawtooth oscillators. The base saw, the reverse saw and the universal saw.

The base sawtooth produces a sawtooth formed wave, where the wave slowly rises from minimum to maximum and then rapidly descents to minimum again. In other words, the top of the wave is at the end of the wave.

The reverse sawtooth also produces a sawtooth formed wave but with a reversed profile: it rapidly ascend to maximum and then slowly descends to minimum. In other words, the top of the wave is at the start.

The universal sawtooth oscillator can produce a wave-form with the top at a user-specified position. This allows the sound to be more mellow, or sharper, or anything in-between.

~BSAW (F -- *out): Creates a base sawtooth oscillator, oscillating at frequency F

~RSAW (F -- *out): Creates a reversed sawtooth oscillator, oscillating at frequency F

~USAW (F top -- *out): Creates a sawtooth oscillator, oscillating at frequency F and where the top is between 0 and 100% of the width of the wave. Top is a signed 32b number where lowest negative value indicates a top at the start of the wave and highest positive value indicates a top at the end of the wave. An easy way of getting correct numbers is by using '**~%**'. It will calculate the position using a percentage -> 50 ~% will put the top in the middle.

For example:

```
440 50 ~% ~usaw constant saw1 \ create sawtooth-oscillator
saw1 saw1 ~output drop        \ create output element
```

produces a sawtooth oscillator with the top at 50% of the width of the wave, and an output element with the same saw-wave oscillator as input in both of its inputs.

The block-wave oscillators:

There are two block wave oscillators:

The base block oscillator produces a block-form wave with a fixed 50/50% ratio. It produces a lot of nasty harmonics, and sounds rather unpleasant at any frequency and even worse at low frequencies. But the sound is very recognisable, as it was used a lot on early homecomputers. So if you are working on a vintage game, this is the oscillator you would consider first. Listen to a few beeps of this oscillator and the early eighties spring back in mind forcefully.

The universal block wave oscillator produces a wave with a user-definable ratio. Any ratio between 0 and 100% can be chosen by the user.

~BBLOCK (F -- *out): Creates a base block oscillator with a fixed 50/50 ratio and oscillating at frequency F

~UBLOCK (F ratio -- *out): Creates a universal block oscillator with a user-definable ratio and oscillating at frequency F. The ratio is signed integer number between minimum and maximum integer value. Zero gives a ratio of 50%. The '**~%**' word can be used (see '~USAW')

The sine oscillator:

There is 1 sine generator as there is only 1 sine. It produces a very nice mellow sound, but some find it a bit boring.

~SINUS (F -- *out): Creates a base sine oscillator, oscillating at frequency F

The noise generators:

There are three noise generators. They produce slightly different sounding noise. In reality the difference between the sound produced is hardly audible.

~FNOISE (F -- *out): produces a noise which changes value randomly at the given frequency F. Some base-frequency might be audible. This generator has the same common features as the above mentioned oscillators. Thus the frequency can also be controlled by an external source and phase-control is available. Why you would need phase-control for a noise-generator is unclear, but it is there for you to play with if you feel the need to do so.

~BNOISE (cycle -- *out): produces a basic noise which randomly changes it's output from max-integer to min-integer or vv. at the specified interval of cycles. The sound of this generator is pretty harsh.

~SNOISE (cycle -- *out): produces a slightly softer noise which changes it's output to a random value at the specified interval of cycles. This generator is comparable to the ~FNOISE generator, but with a different time-base.

PIXEL

This chapter and PIXEL are under development

Overview:

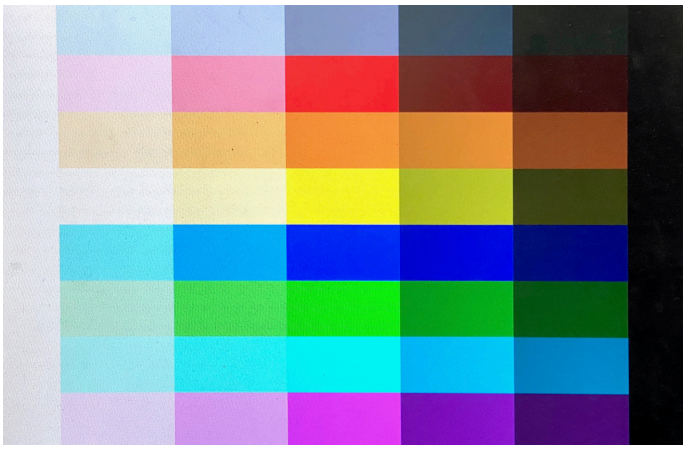
The wabiForth system contains an experimental graphical system called PIXEL. The setup of PIXEL tries to follow the way the graphical systems of the home-computers of the mid-eighties worked.
All the management task for PIXEL are provided by a dedicated core of the ARM-processor, the Raspberry-GPU is not used. The big advantage is that the user has full control over every pixel being drawn. **PEEKing** and **POKEing** of pixels is very much a feature of wabiForth. Developing the latest 3D-shooter is probably not feasible.

Colour-system:

WabiForth uses 24 of the available 32 bits for defining colours, in the normal RGB order. The 4th byte is the Alpha-byte which on a Raspberry does not function. In wabiForth it can be used to store information on the screen.

Predefined colours:
The following colours have been pre-defined as CONSTANTS:

black	**white**			
gray	**dgray**	**vdgray**	**lgray**	**vlgray**
red	**dred**	**vdred**	**lred**	**vlred**
orange	**dorange**	**vdorange**	**lorange**	**vlorange**
yellow	**dyellow**	**vdyellow**	**lyellow**	**vlyellow**
blue	**dblue**	**vdblue**	**lblue**	**vlblue**
green	**dgreen**	**vdgreen**	**lgreen**	**vlgreen**
cyan	**dcyan**	**vdcyan**	**lcyan**	**vlcyan**
magenta	**dmagenta**	**vdmagenta**	**lmagenta**	**vlmagenta**



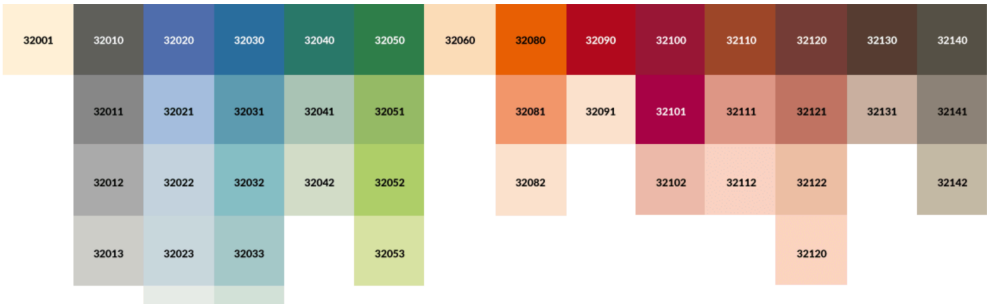
The special color green, defined by my daughter Lize:
greenlo

The Corbusier colour palettes:

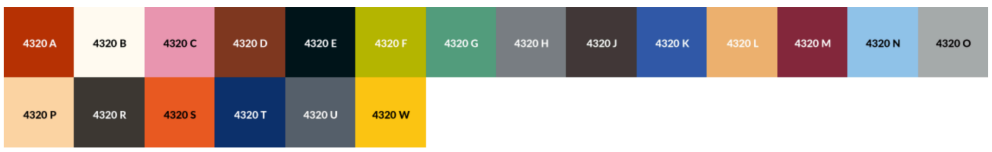
The 1931 and 1959 colour palettes of Le Corbusier have also been pre-defined as CONSTANTS. You need a good monitor to see them as they are intended. All colours match when combined.

32001_blanc
32010_grisfon31
32011_gris31
32012_grismoy
32013_grisclr31
32020_bleuoutr31
32021_outrmoy
32022_outrclr
32023_outrpal
32024_outrgris
32030_bleuceru31
32031_ceruvif
32032_cerumoy
32033_ceruclr
32034_cerupal
32041_vertanglclr
32042_vertanglpal
32050_vertfon
32051_vert31
32052_vertclr
32053_vertjauneclr
32060_ocre
32080_orange
32081_orangeclr
32082_orangepal
32090_rouverm31
32091_rosepal
32100_roucar
32101_rourub
32102_roseclr
32110_ocrerou
32111_ocreroumoy
32112_ocrerouclr
32120_tersnnbru31
32121_tersnnbrq
32122_tersnnclre31
32123_tersnnpal
32130_terombbru31
32131_ombbruclre
32140_ombnat31
32141_ombnatmoy
32142_ombnatclre
4320A_rouverm59
4320B_blancivo
4320C_rosevif
4320D_tersnnbru59
4320E_noirivo
4320F_vertolvvif
4320G_vert59
4320H_gris59
4320J_terdombbru59
4320K_bleuoutr59
4320L_ocrejauneclr
4320M_lerubis
4320N_bleuceru59
4320O_grisclr59
4320P_tersnnclre59
4320R_ombnat59
4320S_orangevif
4320T_bleuoutrfon
4320U_grisfon59
4320W_jaune

The colour palette of 1931



The colour palette of 1959



The PIXEL window-system

WabiForth contains a basic windowing-system. Compared to modern standards, and even compared to Windows 3.1, it is very primitive. But it is fun to play with and it has a couple of unique features.

A maximum of 8 windows can be defined in wabiForth. The order of visible windows is fixed, window 0 is the lowest and the other windows are visible in order of numbering.

Window 0 is special. It is always visible and it always has the same size as the screen and thus fills the background completely. Any other visible window floats on top of window 0.
A number of routines are specifically made for window 0 to make use of the fact that size and memory location of window 0 are fixed, and thus using window 0 can be faster.
It is intended that window 0 will be the standard output for system and error-messages but this is still under development.

Words available for PIXEL:

Drawing:

WINPIXEL: (x y win# --) draws a pixel at x,y of window win# using the INK setting of win#. If x,y are out of bounds, nothing is changed. If x,y point to a section of a window outside of the screen, the pixel is drawn in the window.

WIN0PIXEL: (x y --) draws a pixel at x,y in window 0 using the INK-setting of window 0. If x or y are out of bounds, no changes are made. This routine is upto ~3* faster than the general pixel-draw routine WINPIXEL.

WINLINE: (x0 y0 x1 y2 win# --) draws a line from x0,y0 to x1,y1 on window win# using the INK setting of the window. The routine draws 1 pixel if the length of the line is zero. This is an implementation of the Bresenham algorithm.

HORLINE: (x y len win# --) draws a horizontal line starting at x,y with a length len using the INK setting. The routine does not draw anything if the len=0. Drawing a horizontal line is ~5 times faster than drawing a line with **WINLINE.**

DRAW4x4: (addr x y win# --) draws a 4x4 box using the pattern stored at addr. The pattern is stored as 16 32bit color constants. The ranges of X and Y are from 0-255 and from 0-192 respectively. This is an optimized assembly routine, and as such upto 20x faster than an equivalent in wabiForth.

DRAW2x2: (x y win# --) draws a 2x2 box using the INK setting of the window. Range x=0-511 y=0-383.

WIPE2x2: (x y win# --) draws a 2x2 box using the CANVAS setting of the window win#. Range x=0-511, range y = 0-383.

DRAWBOX: (x y lenx leny win# --) draws a filled rectangle on window win#, using the INK-setting of window win#. The rectanle starts at x,y and has a width of lenx and a height of leny. It is valid for the box to be partly or complety outside of the window.

Words to make and resize a window:

MAKEWIN: (maxx maxy win# --) creates a new window with size specified by maxx and maxy. A window can be resize afterwards at will but cannot be bigger than maxx and maxy. A new window uses the present values for the variables INK and CANVAS. A newly defined window is not visible. This allows for flicker-free changes to the content.

>WINSIZE: (x y win# --) sets the new size of a window. The size can not be larger than the maximum size set during definition of a new window. Changing the size of a window invalidates the content of a window but a WINCLEAR has to be performed by the programmer.

WINGETSIZE: (win# -- x y) for window win# gets the size in pixels.

Words related to the handling of windows:

WINCHAR: (char x y win# --) writes character char at position x,y on window win# using the INK and CANVAS settings for the window.

WINHOME: (win# --) puts the non-visible text-cursor at position 0,0 in window win#. The window is not cleared.

WINTAB: (position win# --) moves the text-cursor on the present line to position by printing spaces. If the cursor is already past the position, nothing happens. Normal the ANSI word TAB is used, which calls WINTAB.

WINCLEAR: (win# --) clears window win# using the CANVAS-setting of the window. Does not clear the border of a window.

WINCURSOR: (win# -- x y) for window win# returns the present position of the text-cursor.

WINVISIBLE: (n win# --) sets the visible flag to n. If ‘false’, window win# is not visible. If flag is greater than 0, window win# is visible.

A wabiForth specific feature is that that this flag counts down at the refresh-rate. This means that ANY window is only temporarily visible with the exception of window 0. Setting this flag to -1 results in the maximum period of visibility, which is around 10 months since the setting of the flag. For most applications this is long enough.

>WINRAND: (pixel win# --) sets the width of the border of a window. The maximum width is 32 pixels. The border is not part of the active window. For instance text never prints in the border and a WINCLEAR does not clear the border.

Wether you want to use a border is up to you and your taste. But a nice fat red border functions well as a warning!

>WINORIG: (x y win# --) sets the origin of a window. X and Y are the location of the upper left corner of a window, and changing them moves the window. It is valid for a window to be partly or completely outside the screen. This does not change the content or functionality of a window.

WINSCROLL: (p win# --) scrolls window win# up by p pixels, clears the area at the bottom with the canvas color of the window. If p is larger than the hight of the window, the window is just cleared.

>WINNOSCROLL: (scroll-disable-flag win# --) sets of clears the window scroll disable flag. The content of a window will not scroll up if the flag is set to 'true'. Default-value for a window is 'false'. Scrolling up normally happens when printing text in a window.

Words to set and get INK and CANVAS values:

>WININK: (ink win# --) sets the INK setting for window win#. Any 24 bit number can be used.

>WIN0INK: (ink --) sets the INK setting for window 0 to the given ink. Any 24 bit number can be used. See the PIXEL intro text for the details of INK. This word is ~3* faster then the general >WININK.

WINGETINK: (win# -- ink) gets the present INK setting for window win#

>WINCANVAS: (canvas win# --) sets the CANVAS (=backgrond color) setting for window#. CANVAS and INK have the same properties, they are 24bit numbers.

WINGETCANVAS: (win# -- canvas) gets the present CANVAS setting for window win#.

Words to control where to print:

WIN#>TASK#: (win# task# --) sets the window on which a task prints. Presently only task 0 is active. But task 0 can be made to print on any of the 8 windows. The effect of the word is to direct the output of words like EMIT to a given window. A task can only print to 1 window at a time. By default task 0 prints to window 0.

UART>TASK#: (flag task# --) sets or clears the UART print-disable flag for a given task#. Presently only task 0 is active. Setting the flag to true results in the disabling of printing for a given task. Disabling the UART-printing can save time if you print a lot to the screen.

Words related to the handling of the alpha-byte:

GETWIN0ALFA: (x y – (false) or (addr value true)) (notice the spelling of the word ALFA!) gets the value of alpha of the pixel at x,y of window 0. If the x or y are out of bounds a false is returned. Otherwise the address of the pixel, the alpha-value and a true are returned. The address allows for fast writing of a new alpha-value without recalculating the address.

For an explanation of alpha and why that is available see the intro text of PIXEL.

SETWIN0ALFA: (alpha x y --) sets the alpha-value of the pixel at x,y. If out of bounds, nothing is changed. The alpha-value is a 8 bit value.

Windows info-table

PIXEL contains all control-data for each window in an array of 8*132 bytes. For each of the 8 windows a list of values is stored. Not all items are presently functional.

Please take note that the info-table is mainly ment for getting data on a window. For changing of values usually it is better to use the specific words for that. Changing values directly might have unexpected results.
For instance changing the size of a window should be done with **>WINSIZE** and not by writing the new x and y values directly in this table. The same is true for **>WINRAND.**

variable	offset	comments
pxmaxx	0	maximum width window - window can be smaller but never larger
pxmaxy	4	maximum height window - window can be smaller but never larger
pxorigx	8	location window relative to screen - can be outside of screen
pxorigy	12	ditto
pxsizex	16	current size window in x-direction
pxsizey	20	ditto
pxcurx	24	position grafical cursor in x-direction
pxcury	28	ditto
pxbuffer	32	pointer to memory-buffer of window - filled during the creation of a window
pxsizebuf	36	size of the memory-buffer - filled during the creation of a window
pxbackgr	40	not in use
pxpitch	44	number of bytes per line of pixels of a window - allows fast fill of pxwopitch
pxmask	48	not in use
pxsizemask	52	not in use
pxtext	56	not in use
pxsizetxt	60	not in use
pxcurstxt	64	not in use
pxrand	68	size border around window - can be from 0 (default) to 32 pixels wide - the border is an area which will not be overwritten by grafical commands

pxink	72	ink for this window – start with of–white for win0 – other windows start with the value contained in the variable INK
pxcanvas	76	canvas color for this window – win0 starts with very dark cyan – other windows start with the value contained in the variable CANVAS
pxfont	80	pointer to default–font for a window – presently always point to the font NeoForth and has no actual function
pxspacingx	84	horizontal spacing between chars – presently defaults to 2 and has no actual function
pxspacingy	88	vertical spacing between chars – presently defaults to 2 and has no actual function
pxcharx	92	number of chars per line – calculated during the creation of a window
pxchary	96	number of lines in window – calculated during the creation of a window
pxvarout	100	varOUT for a window – is x_pos of the textcursor
pxvarrow	104	which row is text cursor
px_wobase	108	address 1st pixel to draw –> this value is filled during creation of the window
px_wohix	112	size x window minus 2*border – limits of drawing –> redo: see px_wobase
px_wohiy	116	size y window minus 2*border – limits of drawing –> redo: see px_wobase
pxflags	120	not in use
pxnoscroll	124	flag=true–> no scrolling
pxvisible	128	window is visible if flag <> 0 –this value counts down to zero at the screen–refresh rate, once zero is reached the window will become invisible

Word available for using the windows info–table

WINADR> (win# –– addr) for a given window (0 – 7) returns the address where the relevant part of the info–table starts. Adding the offset from the table above gives the actual address of a value.

Random numbers

Generating high quality random numbers reliably is surprisingly complex, care has to be taken to avoid pitfalls. WabiFORTH has 2 different ways of generating random numbers. A very fast, deterministic generator of excellent quality, and a slow non-deterministic method which generates true random numbers. The two methods together should cover all realistic needs.

The fast method is based on the XOSHIRO-method as published by David Blackman and Sebastiano Vigna in 2018. The version used here is the 256 bit version with multiply-roll-multiply scrambling. It has a period of >1e77 and returns a random-number in 13-15c. This method is intended as a fast and reliable 'workhorse' function. It produces random values of the highest possible quality. Any combination of bits, both within a generated random value and from value to value, passes all known quality-tests for random-generators (incl. Diehard, Crush and BigCrush). At present there are no known biases in the generated sequence of random values (tested up to 1 Peta generated values). However there are some unlikely sequences of equal numbers known. Fortunately these are so rare that they cannot be measured statistically. To put in perspective how rare: the chance that you encounter the end of the universe is very much larger than the chance that you will ever in your life encounter an unlikely sequence. And if you would encounter such an unlikely sequence, you would not notice.
It is believed by the authors that the method can be used for any serious task. But for critical work, obviously you should test yourself if the method is suitable before using it. It should also be noted that the method is not intended for cryptographic work.
The XOSHIRO-method has been put into public domain by David Blackman and Sebastiano Vigna and can therefore be used legally for all tasks.

True random numbers are generated by the true random generator (TRNG) in the ARM-processor. This TRNG conforms to NIST SP800-90B, NIST SP800-22, FIPS 140-2, and BSI AIS-31. The output cannot be predicted, there is no periodicity, and it is impossible to reset, seed or reseed this generator.
It is however a slow method. A random number is generated every ~50000c. The true random generator has a buffer to store up to four random-numbers for faster access. In case a number is available in the buffer, it is available in around 1800c. WabiForth handles this automatically.
Typically the true random generator is used to seed the other generator periodically. This results in unpredictable random numbers while being much faster than only using the TRNG.

The main words are **RNDM, TRNG and FRNDM**

RNDM (n -- m): returns a random number between 0 and n minus 1 in ~15c.
As the XOSHIRO generator resides permanently inside the NEON-unit of the ARM-processor it has no influence on - and is not influenced in a bad way by - the state of the memory-caches. In other words, a high rate of cache-line dirtying does NOT slow the generator down, as would happen with memory based generators. This is especially beneficial in situations where data is read at random from a large block of memory.

TRNG (n -- m): returns a random number between 0 and n minus 1 in ~1800c. If the the buffer is empty than it takes ~50000c (!) to return a random number.

FRNDM (-- f): returns a 32b floating point number between 0 and 1. This generator is also based on the fast method described above, the generated 32b integer is afterwards converted to a float between 0 and 1.

Related words are **RNDM32 and TRNG32.**
These return a 32bit random number, instead of a word between 0 and the number on the stack minus 1.

RNDM32 (-- u) Uses the same generator as **RNDM,** but returns a 32bit random number in 13c.

TRNG32 (-- u) Uses the same generator as **TRNG,** but returns a 32bit random number.

RESEEDING THE RANDOM GENERATOR

The XOSHIRO algorithm uses 4 internal 64bit seeds to generate the random values. No other information is used by the algorithm. The 4 seeds can be set to any value using the word **RNDMTOSEED.** And if the present values of the seeds are needed, for instance to store them for later use, the word **GETSEEDX** copies the seeds to the stack.

RNDMTOSEED (d_s0 d_s1 d_s2 d_s3 --) Copies the 4 doubles from the data-stack to the seeds in the XOSHIRO generator. Any 4 value can be used as long as at least 1 of the 4 seeds is non-zero.
If the seeds have a very low number of set bits, for instance if only 1 of the 4 numbers is non-zero, the first 10 to 20 random values generated lack randomness.
A sequence of random values is exclusively defined by the 4 seeds.

RNDMGETSEED (-- d_s0 d_s1 d_s2 d_s3) Copies the 4 64-bit seeds to the data-stack.

The random generator can be reset to the initial state, as found after a reboot of the system, or randomized to a new starting point.

RNDMRESET (--) resets the seed for the random generator to the situation as found after a reboot of the system.

RANDOMIZE (--) sets the seeds of the random generator to random values. There is no way of predicting what the values are as the internal true random generator is the basis for this function.

JUMP GENERATION

In certain use-cases it is important to be able to define 2 or more subsequences of random values which with 100% certainty do not overlap. As the XOSHIRO algorithm is a linear method it is possible to define a jump function. A jump function calculates what the value of the seeds would be after a certain number of

random values have been generated (ie the name jump-function). And that without actually generating these numbers.
The jump-function in wabiForth calculates how the seeds would be after the generation of 2^128 values. This allows for the creation of up to 2^128 non-overlapping subsequences. The present maximum rate of random value creation is around 100 million/s. Generating 2^128 samples at this rate would take around 1e23 year. The jump function saves time by performing this calculation in 14 mcs.

There is no jump function for the True Random Generator as sequences from this generator cannot overlap and never repeat.

JUMPSEEDX (d_s0 d_s1 d_s2 d_s3 -- d_s0 d_s1 d_s2 d_s3) calculates, based on the four 64 bit seeds as input, what the value of these seeds would be after generating 2^128 random samples. This allows for the definition of non-overlapping sequences of 2^128 samples each.

Other words related to randomisation in wabiFORTH:

INITTRNG (--) internal word which starts the true random generator (TRNG) during start-up of the system. It is safe, but useless, to call this word. **INITTRNG** does NOT reseed or reset the TRNG, it just starts it. There is no way of halting the TRNG-generator.

Cyclic Redundancy Check (CRC)

The wabi system has several words which allow for fast and efficient generation of CRCs using specific hardware-features of the ARMv8 processor. Generating CRCs this way can be up to 20 times faster than using software alone. WabiForth uses **0x04C11DB7** (=ISO/IEEE) as polynomial for the generation of the CRC.

The following words are available:

CRC8B: (CRC_previous, byte-- CRC_updated) Based on the previous CRC and a byte as input generates a new 32bit CRC. Using this word on the 4 bytes of a word generates the same CRC as using CRC32B on that word.

CRC32B: (CRC_previous n -- CRC_updated) Based on the previous CRC value and a word as input generates a new 32bit CRC.

DATACRC32: (CRC_previous, addr_c, len -- CRC_updated) Based on a previous 32bit CRC and a data-set starting at addr and with a length of len, generates a 32bit CRC.
This word uses the data as is. This means that a string starting with a capital character results in a different CRC than the same string starting with a small character. If the generated CRC has to be independent of the case of the characters used, use **$CRC32** instead of **DATACRC32.**

$CRC32: (CRC_previous, addr_c, len -- CRC_updated) Based on a previous 32bit CRC and a string starting at addr and with a length of len, generates a case-independent 32bit CRC. **$CRC32** is 4-5 times slower than **DATACRC32.** This word works by converting all lower-case characters to upper-case characters during the generation of the CRC.

CRCXMODEM (byte_in --) Generates a 16 bit CRC as used by te XMODEM-protocol.
Before generating a CRC for one XMODEM block, the variable **CRCXM** must be set to zero. After this the individual bytes of the XMODEM block are fed to this word. After all bytes of 1 block for the XMODEM protocol have been included, the variable **CRCXM** contains the 16bit CRC.

CRCXM: variable used in combination with **CRCXMODEM.**

Optimisation in wabiFORTH

During the development of wabiFORTH, performance was an important factor, and several design-decisions result in higher running speed. Some of these can be influenced by the user during programming. Others are fixed.

Optimisations implemented in wabiFORTH

Subroutine threading: WabiFORTH is subroutine threaded. This means that at all times wabiFORTH, and any Forth program compiled by the user, is running as if it is a program written in ARMv8 assembly. This is the fastest way of running a Forth-program for the ARM Cortex-a53 processor.

Separate CPU- and user-return-stack: the return-stack used by the CPU and the return-stack available to the user are separated in wabiFORTH. The main reason is the efficiency of the branch-predictor. This branch-predictor is essential for the performance of the CPU. But it only functions well if the return-stack for the CPU is not changed by a user-program.
An added bonus is that the separate return-stack makes wabiForth more stable and resilient against mistakes. For instance if you pop a value from the return-stack by mistake, wabiForth will keep running without a problem.
For the programmer there are no consequences of this separation.

Use of assembly and full use of the 'leaf' concept of ARM. The 'leaf' concept of ARM is: routines which do not call other routines (called leaf-routine in ARM-terminology) do not need to save and restore their return-address on stack when called. And by dedicating two CPU-registers exclusively to leaf-routines, they also do not have to save and restore registers when called. This allows for extremely fast calling and returning from these routines.
Wherever possible, wabiFORTH primitives are written in optimised assembly using this leaf-principle. For instance all comparisons, most mathematical functions and all stack-manipulation words are written as leaf-routines.
More complex Forth-primitives usually do not benefit from or even cannot use the principle, but these are also written in assembly when useful to do so. Forth-words where speed is less relevant are defined in wabiFORTH to save space and thus raise cache-efficiency.

Dedicated memory blocks for several time-sensitive functions. Faster routines and algorithms can be developed by fixing where certain data is stored in memory. WabiFORTH extensively uses this option. This also optimises the use of the data-caches, by ensuring that critical datablocks always start at cache-line borders.

Dedicated CPU-registers for the DO...LOOP. Dedicated CPU-registers ensure the fastest possible DO...LOOP. This allows the DO...LOOP to do a loop in 2 CPU cycles. And the index '**I**' can be put on the stack in 1-2 CPU cycles.

Inlining of Forth-primitives. The use of the leaf-principle also allows for efficient inlining of Forth-primitives, resulting in an additional increase of speed. Over half of the words in the dictionary can be inlined.
Inlining is a very effective way of avoiding time-consuming calls to subroutines. And the calls that are still made are mostly to words where inlining does not bring speed-advantages. WabiForth inlines a lot. Most programs mainly consist of ARM-opcodes with only a few calls dispersed in-between the opcodes.

Values, variables, constants and literals are always inlined by wabiFORTH. There simply is no benefit of not inlining these. The inlining of other words can be controlled by the user (see later).

Pruning of unnecessary opcodes. If during compilation wabiFORTH notices that certain opcodes are unnecessary, it will prune the superfluous ones. An example is if an opcode for DROP follows an opcode for DUP. In that case both can be deleted. A few other combinations are also pruned or optimised.
Another example is when a number is put on the stack. If the number is positive and smaller as 2^16, two opcodes are generated, otherwise three opcodes are generated.
The pruning-routine ensures that pruning never crosses destinations of jumps, as that could lead to errors and possibly crashes. It also ensures that compounding and pruning do not clash.
It is maybe interesting to note that pruning is done at opcode-level, not at wabiForth word-level. The word DROP followed by the word DUP is not compounded away. But the resulting opcodes might be pruned away.

Hash-Table based FIND: The searching of words in the dictionary is based on a hashing technique. This speeds up the searching for words. For large dictionaries FINDing a word can be more than a 1000 times faster. This is relevant for the speed of compilation and for the speed of interpretation of source-code. The hash-table functionality is also available to the programmer for use within a program. For this see the separate chapter.

Compounding of words: There are many combinations of FORTH words where the assembly-code for the combination of two words is faster than the assembly-code for the two words separately. A few of these combinations are so common that they are part of the ANSI standard. An example is '**1+**'. The only reason why it exists is that it is faster than a separate '**1**' and '**+**'.

One of the best examples which is not an ANSI standard is '**OVER +**'. It takes 4 CPU cycles execution-time when compiled as two separate words, but only 1 CPU cycle when coded as a combination of both words.
There are hundreds, possibly even thousands of these time-saving combinations possible. It would be difficult for a programmer to remember which words-combinations are available as compound words in the dictionary. Especially as the combinations are partly hardware specific.
To ensure that the programmer can program in a care-free way using standard ANSI Forth, whilst at the same time ensuring best possible performance, wabiFORTH compounds words where possible. As an example: if you put '**OVER +**' in the source code, wabiForth will compound these two words into the much faster word '**OVER+**'.

The compounding feature is based on the ANSI-standard. Combinations of words where a ANSI standard exists, like '**1 +**' are not compounded. WabiForth assumes that if you do not use the ANSI-standard word, this must be on purpose.

WabiForth will also not correct programming-errors like a **DUP** followed by a **DROP**. WabiForth assumes that if you do something seemingly so illogical, that undoubtedly you must have a good reason for doing it.

Some final remarks: additional combinations are added to wabiForth in a haphazard way. Compounding functions well, but as it is a complex function, it will take some time before most wrinkles (read: 'bugs') are ironed out.

User-options of wabiForth optimization:

User options Inlining: Normal behaviour for wabiFORTH is to inline wabiFORTH-primitives, provided this inlining is beneficial for speed. This behaviour can be influenced by the user by the use of the immediate words **INLINE** and **NOINLINE.** These words can be used interactively during programming. This allows the user to test whether inlining has benefits for a program or not.
Inlining is switched 'on' as default after a reset or start of the system.
The overall speed advantage of inlining is quit large. In some cases code can run up to 250% faster.
Inlining is fully transparent to the user. Apart from the fact that inlining increases the size of a program by about 50-100%, as far as presently is known there are no disadvantages or risks associated with inlining.

Inlining can temporarily be halted with **HALTINLINE. HALTINLINE** remembers the previous inline-settings, so that inlining can later be set back to the settings as they were before executing **HALTINLINE** by using **INLINEASBEFORE.**

- **INLINE:** (–) starts inlining from that point onwards. Immediate word
- **NOINLINE:** (–) stops inlining from that point onwards. Immediate word
- **HALTINLINE:** (–) stops inlining and remembers the previous inlining-settings
- **INLINEASBEFORE:** (–) reinstates the previous inlining-setting before HALTINLINE was executed

User options Pruning: Pruning can be influenced by the user with the immediate words **PRUNE** and **NOPRUNE.** They can be used interactively during programming.
The overall speed advantage of pruning is usually negligible. But in rare cases pruning can result in an increase of speed of 10% or more for individual words. Pruning is switched on as default after a reset or start of the system.
Pruning is fully transparent to the user. As far as presently is known there are no disadvantages or risks associated with pruning.

User options Compounding of words: Compounding is switched on as a default. It can be influenced by the user with the immediate words **COMPOUND** and **NOCOMPOUND.** These words can be used interactively during programming.
The overall speed advantage of compounding is usually small but noticeable. Expect anything between 0 (if your are unlucky) and 30% (if you are lucky) speed advantage, depending on the sort of program you are writing and your programming-style.

Executing **.COMPOUNDTABLE** will give an overview of which combinations of words are combined into more efficient alternative definitions. The alternative words used for compounding are hidden en so not visible when listing words with **WORDS.** But they can be used in your programs if you want to.

.COMPOUNDTABLE: (--) lists the complete compound-table.

Hash-table functionality

Hash-table functionality: Wabi-Forth uses a hash-table, also known as key-to-address-transformation, to speed up searching for definitions in the dictionary. In Forth, the word **FIND** is the standard word to find a word/definition in the dictionary. Classically **FIND** does this by checking one definition after the other till the correct one is found, or till it can be confirmed that the word does not exist in the dictionary.
The disadvantage of this way of trying to find a definition is that it is slow. And it gets slower when the dictionary grows. In tests done with a library of 2000 words, using the classic way of trying to **FIND** a word, wabi-Forth on average needs ~30000 machine cycles to find a definition. Confirming that a given word does not exist in the dictionary takes ~60000 machine cycles.

A usual way of speeding up **FIND** is by splitting up the dictionary into several smaller dictionaries. This makes FIND faster, depending on into how many parts the dictionary is split. But **FIND** still gets slower as the dictionary grows.

The better way of speeding up the **FIND** is by using a hash-table. This is totally transparent to the user, it is just there and functions. Finding a 5 character word in a 2000 word dictionary with support of a hash table is on average around a 150 times faster than the classic way. And as the dictionary grows, the speed advantage grows as well. Finding the same 5 character word in a 10000 word dictionary is already around 700 times faster. Finding a definition fast is important, but speedily confirming that a word is not available in the dictionary is even more important. And here a hash-table really shines. The best case is confirming that a 1 character definition does not exist in the dictionary. In tests with a 2000 word dictionary, such a short definition can be confirmed to not exist in 60 ns. Which is around 300 times faster than using the classic way. The average improvement is closer to 200 times. Again when the dictionary grows, the speed-advantage get bigger.

Technical setup of hash-table: The hash-table in the wabiForth system is an advanced implementation of a separate entry-pool of 367000 cells and link-pool containing ~2000000 cells. Each cell contains 3 words. The first word is the link-field, the second is the crc-field and the 3rd word contains the message. A new definition is always hashed into the entry-pool. If on putting a new definition into the entry-pool it is found that a cell is already in use by another definition, this original cell will be moved into the link-pool and the new definition is put into the entry-pool. The link-field ensures that the original cell can still be found.

User functionality: A special feature of the wabi-Forth hash-table is that it can be switched off for the use by FIND and that it is then available for use by the programmer. This gives the best of two worlds: during the compilation of a program, the hash-table speeds up the compilation. And once compilation is ready, the hash-table is available to speed up user-programs if so wanted.

Using a hash table is an advanced topic which cannot be discussed here. But the following words are available for the user who wants to give it a try.

USEHTB: (--) resets and starts the use of the hash-table by the wabi-system.

NOHTB: (--) stops the use of the hash-table by wabi-system, and makes the hash table available for use by the user.

HTB_CLEAR: (--) clears the existing hash-table and switches the wabi-Forth system to the classic and slow FIND. The hash-table is then available to the programmer.

HTB_REDO: (--) re-fills the hash-table with the definitions from the dictionary. And switches the wabi-Forth system back to hash-table based fast FIND.

HASHWORD: (message, addr, len --) puts the 32bit message in the hash-table based on a hash of the string/data at the address and the length. What meaning the message has is up to the programmer. It could be a link to a field in a table, or point to a memory-block. Or whatever you need. If wabiForth is using the hashtable, it stores the XT of a word in the message-field.

HTBFIND: (addr_of_string -- message) finds the message based on the hashing of the counted string at addr_of_string.

Measuring the benefits of the hash-table and compiling:

The **benefit of the hash-table** can be big. The following example shows how big:

```
s" y" sliteral tsty
: tst0 (( tsty uncount find 2drop )) ;
```

If you run tst0 twice, once with the hash-table and without, like this:

```
tst0
nohtb
tst0
usehtb
```

You will get the time it takes to confirm that the string "y" is not part of the dictionary, based on trying 1 million times. With the hash-table it is around 363* faster than without the hash-table.

```
69514 mcs -> 104 cycles
25221581 mcs -> 37832 cycles
```

A longer string takes a bit longer, but this is still very much faster than without. Try it!

```
s" Doemaarwat" sliteral tsf10
: tst1 (( tsf10 uncount find 2drop )) ;
```

The following example also clearly demonstrates the benefit of a hash-table. It measures the shortest possible time in total to:

- put 2 numbers on stack
- add these numbers
- drop the result

TST0 uses normal compiled and optimised wabiForth, TST1 interprets the needed steps with the word EVALUATE. TST1 is run twice, once with hash-table on, and once with the hash-table off.

```
: tst0 (( 2 2 + drop )) ;
: tst1 (( s" 2 2 + drop" evaluate )) ;

tst0
tst1
nohtb
tst1 ( << this takes a while, grab a quick coffee... )
usehtb
```

You will see that the difference in running time is large. In fact, the running time of the second run of TST1 is more than a minute. The slowest way is 34.000 times slower than the fastest way. Using the hash-table is ~70* faster than not using the hash-table with around 2000 words in the dictionary.

Real time clock module

The RTC related words only function if a RTC-module is connected to the Raspberry. WabiFORTH contains a driver for such real-time clock. The clock can be read and controlled using the following words:

Date and time related functions

RTCSET: (hh mm ss dd mm yy --) sets the RTC-clock according to the data on the stack. Please be aware that the system, in accordance with a longstanding FORTH-tradition, does not perform any checks on the data. Making sure that the data is valid is your responsibility.

RTCGETDATA (--) fills the following values with data from the real time clock.

RTCYEAR (-- year) returns the year in the format yyyy

RTCMONTH (-- month) returns the month

RTCDAY (-- day) returns the day of the month

RTCWEEKDAY (-- weekday) returns the weekday as a number, where 1=Sunday, 2=Monday etc

RTCHOURS (-- hours) returns the hours in 24 hour format

RTCMINUTES (-- minutes) returns the minutes

RTCSECONDS (-- seconds) returns the seconds

The following printing-words are available:

.WEEKDAY (1-7 --) prints the name of a day as a 3 character abbreviation, where 1=sun, 2=mon etc.

.MONTHNAME (1-12 --) prints the name of a months as a 3 character abbreviation, where 1=jan, 2=feb etc.

.DATE (--) prints the date in the format dd-mm-yyy

.DATECAMEL (--) prints the date in the format DDmmmYYYY

.TIME (--) prints the time in the format hh:mm:ss

.RTC (--) prints date, weekday and time

These printing words all always execute RTCGETDATA (see above) and so print the most up-to-date time and dates.

OTHER DATE-RELATED WORDS

- **GETWEEKDAY** (dd, mm, yyyy -- 1-7) returns the day of the week for a given date in the Gregorian calendar (which starts at October the 15th 1582). Where 1=Sunday, 2=Monday etc.

- **LEAPYEAR?** (yyyy -- flag) returns 'true' if the year on the stack is a leap-year, otherwise returns a 'false'.

BATTERY PROTECTED MEMORY OF THE RTC-MODULE

The real time clock module contains 64 bytes of battery protected RAM, and these are available to the user. These 64 bytes are at locations 32 to 95. Positions 32 to 39 also double as power-down memory. If the RTC-module looses external power, the date and time where the battery took over the power-supply of the RTC-module is stored in these positions. This allows to monitor an external power-supply.

The following words are available to use and manage the 64 bytes of RAM:

RTCCLEARRAM: (--) clears all 64 bytes of RAM, including the Power-down bytes.

RTCDUMPRAM: (--) prints the 64 bytes of RAM in table form.

RTCRAM!: (char, 32-95 --) stores char n at position 32 to 95. When a position outside of the range is specified, no warning is given and no value is stored.

RTCRAM@: (32-95 -- char | -1) fetches an unsigned char from position 32 to 95. If a position outside this range is specified than -1 is returned.

Bit handling

In general Forth makes it easy to handle individual bits with words like **AND** and **OR.** But especially when handling bits in memory, assembly is faster. Therefor wabiForth contains a set of words for this. The main functions are to set or clear a specific bit at a given address, or get the value of a bit from a given address with the words **BIT@ and BIT!.** They are both optimised primitives.

The word **FLAG@** gets the value of a bit from a given memory address and than converts this value of the bit (0 or 1) to a flag-value (0 or -1). Very easy to do in Forth, with a **IF...THEN,** but faster in assembly. There is no need for a **FLAG!** as **BIT!** handles both bits and flags correctly.

The word **NEGFLAG@** does the same as **FLAG@** but with a negative logic. So if the bit is zero, the flag is true, and if the bit is 1, the flag returned is false,

The words **1BIT!** and **0BIT!** do the same as **BIT!** but 3 cycles faster.

Comparison execution time:
Using the following code-snippets you get an impression of how much faster the optimised primitives are.

```
: forthbit@ ( bit# addr -- 0/1 )
  @                     \ ( bit# value )
  swap                  \ ( value bit# )
  rshift                \ ( value>>bit# )
  1 and ;               \ ( 0/1 )

: timeforth (( 23 1000000 forthbit@ drop )) ;
: timeassembly (( 23 1000000 bit@ drop )) ;

timeforth
timeassembly
```

If you run the code you will see that a forth-based BIT@ takes ~15 cycles and the optimised BIT@ takes ~10 cycles. But 2 numbers and a DROP are also part of the loop. These together take 3 cycles to execute. Resulting in a final time for BIT@ of 7 cycles and 12 cycles for the forth-based WFBIT@. So the optimised primitive is faster, but not by thàt much. Still, every bit helps.
For FLAG@ the difference is a bit larger, 17 cycles in wabiForth and 7 as primitive. Avoiding a IF...THEN saves some extra cycles.

BIT-arrays
The words **BIT()@** and **BIT()!** support the use of an array of bit-flags spanning across a range of memory. With these words the specified bit to fetch or store can be any number. For instance, if you specified bit 32 with these words, it would address the first bit in the address 4 bytes higher than the base-address. Bit 33 is the second bit in the base address+4 bytes. Bit 64 is the first bit of the value at base address+8 bytes.

The full range of a single number can be used. So a bit-array can contain up to 2^32 bits, corresponding to a bit-array of 128 MB of memory.

The words **FLAG()@, NEGFLAG()@, 1BIT()!** and **0BIT()!** are also available (see the corresponding words above).

FIELD insert

Especially when setting up registers for peripherals it is very common that the programmer has to set/clear certain bits in a register without changing the other bits of the register. If it is an individual bit then the above mentioned words can be used. If it concerns a field of bits within an address, the word **FIELDINS** is very handy.
FIELDINS inserts a value in a bitfield in a value contained in an address.

For example:

```
17 5 8 myaddr FIELDINS
```

will insert the value 17 in the field starting at bit 8 with a length of 5 bits in the memory-location **myaddr**. The other bits in **myaddr** are not changed.

BIT@ (bit# addr -- 0/1) gets the value of the bit specified by bit# from the value contained in addr. Returns a 0 or 1.

FLAG@ (bit# addr -- false/true) as **BIT@** but returns FALSE or TRUE.

NEGFLAG@ (bit# addr -- true/false) as **FLAG@** but with the opposite logic

BIT! (0/non-zero bit# addr --) stores a 0 (=clears) or 1 (=sets) in the bit specified by bit# from the value contained in addr. Also works with FALSE/TRUE as input.

1BIT! (bit# addr --) sets the bit specified by bit# from the value contained in addr. Faster execution as '1 addr bit!'.

0BIT! (bit# addr --) clears the bit specified by bit# from the value contained in addr. Faster execution as '0 addr bit!'.

BIT()@ (u addr -- 0/1) Gets the value of the bit specified by n from the bit-array starting at base-address addr. The value n is a 32 bit unsigned value. The maximum size of a bit-array is thus 128MB.

FLAG()@ (u addr -- false/true) as **BIT()@,** but returns FALSE or TRUE.

NEGFLAG()@ (u addr -- true/false) as FLAG()@ but with the opposite logic.

BIT()! (0/non-zero u addr --) stores a 0 (=clears) or 1 (=sets) in the bit specified by n from the value contained in addr. The value n can be anything 32 bit unsigned value. Also works with FALSE/TRUE as input.

1BIT()! (u addr --) the same as 1BIT!, but for bit-arrays. A bit faster than '**1 u addr BIT()!**'

0BIT()! (u addr --) the same as 0BIT!, but for bit-arrays. A bit faster than '0 **u addr BIT()!**'

FIELDINS (u len start addr --) inserts the value U in the bit-field defined starting at bit **start** and with a length **len** at the memory-address **addr.** The other bits are left unchanged. Using a value of 0 resets the bits in the bit-field to 0.

Not so serious stuff

The ARMv8 processor has a long history. Some remnants of this long history are still present in the CPU. Features that were once useful but are now outdated. Of course wabiForth supports these features, if only out of respect for all the brilliant people who helped develop these processors.

Scratch-byte
In case the 1072693248 bytes of usable memory of the Raspberry 3b+ are not enough, the CPU contains 1 whole extra byte of scratch-storage in the UART-section. To make the use of this byte convenient, wabiForth contains 2 words: **C!SCRATCH** and **C@SCRATCH**. Using these you can store a byte for later use, or fetch it. The disadvantage of the scratch-storage is that the access-time is pretty long, namely around 220 cycles. Normal memory is at least 70 times faster. But if you really need one more byte, there it is...

C!SCRATCH: (c --) stores a character in the scratch-byte

C@SCRATCH: (-- c) fetches a character from the scratch-byte

ARM-v8 assembler

An integrated ARMv8 assembler was an important requirement for WabiForth. After a reboot or **'COLD'**, the assembler is available immediately. No need for messing around with vocabularies or loading of source-code.
Contrary to previous versions of wabiFort, the assembler cannot be 'forgotten' anymore. This 'forgetting' of the assembler offered the almost unparalleled benefit of freeing 0.0052% of memory, but alas, that is not available anymore.

The assembler is reasonably complete. Missing are NEON opcodes and literal pools. The NEON opcodes will be added in future, the literal pools not (but you can always use **PC**, as register).

And finally one has to use **MOV** for the shifts and rotate, but this does not limit functionality.

Any relevant additions to the assembler, especially functionality, will happen once more experience is gained with the present version. The focus now is on testing and debugging.

On the following pages the available assembler words are listed.

Why using the assembler

There are several reasons to use the assembler:

First: **the fun** you can have with an assembler. Being in total control, and being fully responsible for every aspect of a routine can be very satisfying.

Second: **speed.** WabiForth pushes the hardware pretty hard, with the assembler you can push it even harder.

Third reason: **help the optimiser.** The optimiser only handles a few common situations and is risk-averse to the absurd. For instance it will never optimise directly after an immediate word. Chances are pretty high that you can do better.

Fourth reason: **access to parts of the ARM-system** which are not covered by wabiForth functionality. For instance wabiForth has no support for the debug-functionality of the ARM-processor. But if you wish to do so, this is available using the assembler.

Forth words and opcodes can be mixed
Assembly code can be used in any of the defining words. A very important feature of the wabiForth assembler is that it is possible to mix Forth words and assembly code at will, and within any definition. And there is no limit to mixing.

The following, very silly, example (which you do not have to understand and which does nothing useful) shows respectively a literal (10000), an opcode (**MOV,**) a **DUP**, an opcode (**MVN,**) and a **STAR.** It puts -49 on top of stack. It shows the principle of mixing Forth words and opcodes. (it also shows 2 ways of creating values in assembly)

```
: silly ( -- -49 )
  10000                  \ << this puts 10000 on the stack
  [ top, 0 7 i#, mov, ]  \ << this changes the 10000 to 7
  dup                    \ << this duplicates the 7
  [ top, 0 6 i#, mvn, ]  \ << this changes one 7 in -7
  *                      \ << this multiplicates -7 and 7
;
```

Format and syntax:
The assembler follows the established Forth-way of assembling by having the registers followed by the opcode.

An example:
```
  ADD r1, r2, r3     \ add r2 and r3 with the result in r1
```

in the standard ARM-assembler notation.

Would become:
```
  r1, r2, r3, ADD,
```

in the wabi assembler.

Notice the comma's after every word in the wabi-assembler. They are there to avoid confusion between Forth definitions and assembler definitions.

There are a few exceptions to the 'comma'-convention:
- destination labels end with a colon (fi: **label0:**)
- ordinary numbers are written normally (fi: **23**)
- the word **LBL[** has no comma as it manages the label-system and does not assembly anything

Assembler words in wabiForth are not immediate
This means that you have to put them between brackets within a new definition.
There are pro's and con's to having the words compiling or immediate. The big advantage of normal compiling words is that it is much easier to develop macro's. No need to resort to endless POSTPONE statements.
The disadvantage is that you have to put brackets ('[' and ']') around blocks of assembly words.

ASSEMBLER WORDS

Normal definitions and primitives
Opcodes are either part of a normal wabiForth definition or part of a primitive. The distinction is important. Primitives must follow certain rules to function correctly.

CODE: defines a primitive
In Forth the colon, ‘:’, is for defining Forth-words and usually the word **CODE:** (or something similar) is used for definitions using assembly.
WabiForth is more flexible, assembly/opcodes can be mixed with Forth words at any time and place. The function of **code:** is not

to distinguish between Forth and assembly, but rather to define a primitive.

A primitive is a definition which does not call another definition. Because of this there is no need for the link-register (=r14) to be saved and restored. And that is the exact function of **CODE**: It starts a definition which does not save and restore the link-register. And this allows a primitive to be inlined if wanted.

In the chapter "**Snippets of code – examples of assembly**" there are examples of inlinable primitives.

Rules for a primitive

- primitives must not call any other definitions
- only the registers **V,** and **W,** can be used as scratch-registers within a primitive. Registers **r0, r1, r2, r3 and r4** cannot be used in a primitive. The registers with specific functions, **DTS,** and **TOP,** can be used within a primitive for their specific function.
- registers **V,** and **W,** cannot be used outside of a primitive.

WabiForth will only function correctly if these rules are followed. The bugs resulting from violating these rules can be dreadfully hard to find.

REGISTERS

The ARM processor in ARM32 mode has 16 registers. Register 5 to 15 each have two names. The standard ARM-name, like "r5,", and the name with which they are known internally in the wabi system. These names are synonyms and can be mixed.

```
r0,              (scratch for non-primitives)
r1,              (scratch for non-primitives)
r2,              (scratch for non-primitives)
r3,              (scratch for non-primitives)
r4,              (internal scratch, do not use)
r5,   =    top,  (top of stack register
r6,   =    fps,  (floating point stack pointer)
r7,   =    i,    (index of DO...LOOP and FOR...NEXT)
r8,   =    lim,  (limit of DO...LOOP)
r9,   =    dts,  (data stack pointer)
r10,  =    uss,  (user return stack pointer)
r11,  =    v,    (scratch, only use in a primitive)
r12,  =    w,    (scratch, only use in a primitive)
r13,  =    sp,   (CPU return-stack pointer)
r14,  =    lr,   (link register)
r15,  =    pc,   (program-counter)
```

CONDITIONAL EXECUTION

Use in front of any opcode which can be executed conditionally. See the ARM documentation for more information.

```
eq,
ne,
cs,   =    hs,
cc,   =    lo,
mi,
pl,
```

```
vs,
vc,
hi,
ls,
ge,
lt,
gt,
le,
al,  ( default, optional )
```

OPCODES

Data Processing Opcodes:

```
and,       ands,
eor,       eors,
sub,       subs,
rsb,       rsbs,
add,       adds,
adc,       adcs,
sbc,       sbcs,
rsc,       rscs,
orr,       orrs,
mov,       movs,
bic,       bics,
mvn,       mvns,

tst,
teq,
cmp,
cmn,
```

Operand 2:
This is a very flexible feature of the ARM-processor.

Operand 2 can either be:

- register
- register with it's content shifted immediately
- register with it's content shifted defined by another reg
- immediate, a 8 bit value rolled right with 0, 2, 4, etc

```
I#,
```

Please note: the following 8 words are ONLY for use in operand 2. Use MOV, if you want to shift or rotate values.

```
lsl#,
lsr#,
asr#,
ror#,
lslr,
lsrr,
asrr,
rorr,
```

16b immediate:

```
movt,
movw,
```

various:

```
sel,
clz,
usad8,
usada8,
```

```
        clrex,
        setendle,
        setendbe,

Multiply:
        mul,        muls,
        mls,
        mla,        mlas,
        smull,      smulls,
        smlal,      smlals,
        umull,      umulls,
        umlal,      umlals,
        umaal,

        smlabb,     smlabt,     smlatb,     smlatt,
        smlad,      smladx,
        smlalbb,    smlalbt,    smlaltb,    smlaltt,
        smlald,     smlaldx,
        smlawb,     smlawt,
        smlsd,      smlsdx,
        smmla,      smmlar,
        smmls,      smmlsr,
        smmul,      smmulr,
        smuad,      smuadx,
        smulbb,     smulbt,     smultb,     smultt,
        smulwb,     smulwt,
        smusd,      smusdx,
        smlsld,     smlsldx,


Load and Store:
        ldr,        str,
        ldrb,       strb,
        ldrh,       strh,
        ldrsh,
        ldrsb,
        ldrd,       strd,

        Load and Store acquire/exclusive
        lda,        stl,
        ldab,       stlb,
        ldah,       stlh,

        ldaex,      stlex,
        ldaexb,     stlexb,
        ldaexh,     stlexh,
        ldaexd,     stlexd,

        ldrex,      strex,
        ldrexb,     strexb,
        ldrexh,     strexh,
        ldrexd,     strexd,

        ldrt,       strt,
        ldrbt,      strbt,

        ldrht,      strht,
        ldrsht,
        ldrsbt,


Addressing modes:
        +],         -],
        +]!,        -]!,
        ]+!,        ]-!,
        i+],        i-],
```

```
        i+]!,       i-]!,
        ]i+!,       ]i-!,


Block Load and Store:
        ldmed,
        ldmib,
        ldmfd,
        ldmia,
        ldmea,
        ldmdb,
        ldmfa,
        ldmda,

        stmfa,
        stmib,
        stmea,
        stmia,
        stmfd,
        stmdb,
        stmed,
        stmda,


Register-block definition:
        {,
        r-r,

        },
        }!,
        }^,
        }!^,

Processor state:
        cps,
        cpsid,
        cpsie,

        processor modes
        user,
        fiq,
        irq,
        supervisor,
        monitor,
        abort,
        hyp,
        undefined,
        system,

        a,i,f flags
        <a>, <ai>, <af>, <aif>, <i>, <if>, <f>,

CoProcessor:
        mcr,
        mcrr,
        mrc,
        mrrc,

        CoProcessor registers:
        p14, p15,
        c0, c1, c2, c3, c4, c5, c6, c7, c8,
        c9, c10, c11, c12, c13, c14, c15,

Move to/from special registers:
        mrs,
        msr, ( only reg to banked_reg - no immediate )
```

special registers:
spsr, cpsr, apsr, (only for mrs, !!)
r8_usr, r9_usr, r10_usr, r11_usr, r12_usr, sp_usr, lr_usr,
r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, sp_fiq, lr_fiq,
lr_irq, sp_irq,
lr_svc, sp_svc,
lr_abt, sp_abt,
lr_und, sp_und,
lr_mon, sp_mon,
elr_hyp, sp_hyp,
spsr_fiq,
spsr_irq,
spsr_svc,
spsr_abt,
spsr_und,
spsr_mon,
spsr_hyp,

External debugging:
ldc, Rn, 8bIMM, {index_mode} ldc, -> 8b immediate
mandatory for every mode of opcode. Leave out p14 and c5.
stc, ditto

Division:
sdiv,
udiv,

Barriers:
csdb, (no options)
pssbb, (no options)

dmb,
dsb,
isb,

options for domain of memory barriers:
sy, (default, optional)
st,
ld,
ish,
ishst,
ishld,
nsh,
nshst,
nshld,
osh,
oshst,
oshld,

Hints:
sev,
sevl,
wfe,
wfi,
yield,
nop,

pldw, for literal variant use pc,
pld, ditto

```
       pli,       ditto


Exception handling:
       eret,
       bkpt,      16b value before opcode mandatory - no condition
       hvc,       ditto

       smc,        4b value before opcode mandatory - condition allowed
       svc,       24b value before opcode mandatory - condition allowed


       rfe,       rfe!,   \ rfe & refia are synonyms
       rfeia,     rfeia!,
       rfeda,     rfeda!,
       rfedb,     rfedb!,
       rfeib,     rfeib!,

       \ for all following: put processor mode before opcode
            -> fi: hyp, srsia,
       srs,       srs!,   \ srs and srsia are synonyms
       srsia,     srsia!,
       srsda,     srsda!,
       srsdb,     srsdb!,
       srsib,     srsib!,


Cyclic redundancy check:
       crc32w, =  crc32,
       crc32h,
       crc32b,
       crc32cw, = crc32c,
       crc32ch,
       crc32cb,


Bitfield manipulation:
       bfc,
       bfi,
       sbfx,
       ubfx,


Extent (and add): use ror#, to specify which byte - 0 is default
       sxtb,
       sxtab,
       sxth,
       sxtah,
       uxtb,
       uxtab,
       uxth,
       uxtah,

SIMD extent (and add):
       sxtb16,
       sxtab16,
       uxtb16,
       uxtab16,

Saturated aritmatic:
       qadd,
       qdadd,
       qsub,
       qdsub,
```

```
        ssat,
        usat,
        ssat16,
        usat16,
```

Packing and unpacking:

```
        pkhbt,
        pkhtb,
```

Parallel additions and subraction:

```
        sadd16, qadd16, shadd16, uadd16, uqadd16, uhadd16,
        sadd8,  qadd8,  shadd8,  uadd8,  uqadd8,  uhadd8,
        sasx,   qasx,   shasx,   uasx,   uqasx,   uhasx,
        ssax,   qsax,   shsax,   usax,   uqsax,   uhsax,
        ssub16, qsub16, shsub16, usub16, uqsub16, uhsub16,
        ssub8,  qsub8,  shsub8,  usub8,  uqsub8,  uhsub8,
```

Byte and Bit Reverse:

```
        rev,
        rev16,
        rbit,
        revsh,
```

Branch:

bx, (reg --) - branches to address in register
blx, (reg --) - linked branch to address in register - this is like a jump to subroutine to the address in register

Conditional Branch:

Some branches have two different names for the same function. They are put next to each other in the following list

```
        beq,
        bne,
        bcs,  = bhs,
        bcc,  = blo,
        bmi,
        bpl,
        bvs,
        bvc,
        bhi,
        bls,
        bge,
        blt,
        bgt,
        ble,
        b,    = bal,

        bleq,
        blne,
        blcs, = blhs,
        blcc, = bllo,
        blmi,
        blpl,
        blvs,
        blvc,
        blhi,
        blls,
        blge,
        bllt,
        blgt,
        blle,
        bl,   = blal,
```

LABELS

There are 16 destination labels. The word **LBL[** resets these. All 16 can be used within each new definition. Technically it is possible, to reuse the same destination-label within a definition. But that is an advanced topic, and with 16 destination-labels it should not be necessary.

LBL[(--): resets the label-addresses. Normally used once per definition, but if labels are not used inside a definition than the reset can be left out. A label-reset makes further resolution of previous jumps impossible, so make sure that each branch has the corresponding label. Multiple definitions can share labels. This makes it possible to branch from one definition to a label inside another.

```
label0:
label1:
label2:
label3:
label4:
label5:
label6:
label7:
label8:
label9:
label10:
label11:
label12:
label13:
label14:
label15:
```

There are 16 source-labels. One for each destination-label. Source-labels can be used as often as needed within an individual definition. They are put in front of a branch-opcode.

```
>lb0,
>lb1,
>lb2,
>lb3,
>lb4,
>lb5,
>lb6,
>lb7,
>lb8,
>lb9,
>lb10,
>lb11,
>lb12,
>lb13,
>lb14,
>lb15,
```

Lessons from one opcode...

The simplest definition with assembly imaginable is a definition containing 1 opcode. The example here only adds 15 to the top. This is enough to learn a few valuable lessons on assembly.

As base for comparison, the example in forth:

```
: f15+ 15 + ;
```

The same but using assembly inside a normal definition. Note: there is no need to uses **LBL[** as there are no labels.

```
: w15+ [ top, top, 0 15 i#, add, ] ;
```

And using assembly as a primitive.

```
code: c15+ [ top, top, 0 15 i#, add, ] ;
```

We can compare the three definitions with the words **((** and **))**. The 3 different version have different execution-times. A suitable test-setup would be the following. We put a zero on top of the stack, add 15 to it with one of the test-routines above, and then drop the result.

```
: tstf (( 0 f15+ drop )) ;
: tstw (( 0 w15+ drop )) ;
: tstc (( 0 c15+ drop )) ;
```

Running the three test-words gives the number of cycles execution-time for each of the three versions of our definition:

```
tstf 6700 mcs -> 10.0 cycles
tstw 6033 mcs ->  9.0 cycles
tstc 5366 mcs ->  8.0 cycles
```

You'll probably agree with me that the increase in speed is not impressive. Assembly is mostly used to enhance performance. And although the two assembly routines are faster, they are so only by a small margin

I would love to say that this is because wabiForth has such an excellent compiler, but alas, no. The real reason is that the CPU in de Raspberry has a very efficient branch-prediction system coupled with very effective pre-emptive branching. This system saves more cycles/time for a slightly longer routine (the wabiForth version) than for the shorter routine (the assembly versions). Resulting in only a small increase in efficiency due to the use of assembly.

But there is one more thing we can do. Make an **inlinable** primitive assembly routine. In this specific case we only have to add the word '**inlinable**', with a 1 in front of it, after the definition of our primitive.

Like this:

```
code: ic15+ [ top, top, 0 15 i#, add, ] ; 1 inlinable
```

We measure the time in the same way:

```
: tsti (( 0 ic15+ drop )) ;
```

The result is on the next page...

This is a much better result. The routine is now around 3 times faster than the original routine.

tsti 2011 mcs -> 3.0 cycles

Lessons learned:

1. The basic compiler of wabiForth coupled with the excellent Cortex-a53 processor do a good job in creating efficient and fast code.
2. Assembly is faster than Forth, but not by a lot when the routine is not developed with care.
3. At least for short assembly routines, developing inlinable primitives brings the biggest benefits with regard to speed.

Snippets of code – examples of assembly

Violà: a chapter with an eclectic mix of definitions using assembly. There is no special order and it for sure is not enough to learn ARM-assembly. But it shows how certain specific cases are programmed. More examples will be added in a haphazard way.

This example shows how **conditional execution** saves opcodes and branches. With 3 opcodes, ascii characters are converted to capitals:

```
: 2caps ( char -- CHAR ) [
  v, top, 0 97 i#, sub,
  v, 0 26 i#, cmp,
  top, top, 0 32 i#, lo, sub,   \ << conditional execution
] ;
```

This example shows 2 different versions of **ROT** in assembly. The 4 opcode version is the one used in wabiForth as it is slightly faster. The examples shows the use of **normal and immediate indexed addressing** for Load and Store opcodes. It also shows the use of **CODE:** and **INLINABLE.**

```
code: ROT ( m n o -- n o m ) [
  v, top, mov,
  w, dts, ldr,
  top, dts, 4 i+], ldr,
  v, dts, str,
  w, dts, 4 i+], str,
] ; 5 inlinable

code: ROT ( m n o -- n o m ) [
  v, top, mov,
  w, dts, ldr,
  top, dts, 4 i+], ldr,
  dts, {, v, w, }, stmia,
] ; 4 inlinable
```

Here is shown how **-ROT** is coded in assembly. It shows the use of **multiple load and store.**

```
code: -ROT ( m n o -- o m n ) [
  w, top, mov,
  dts, {, top, v, }, ldmia,
  dts, {, v, w, }, stmia,
] ; 3 inlinable
```

The following is an example of **an optimising word.** It combines the function of the words ROT and DROP into 1 word to reduce execution time. ROT uses 5 opcodes and DROP uses 1 opcode. But

combined into 1 word, ROTDROP, only 2 opcodes are needed. And this combined word is 3 cycles faster than ROT DROP separately. If CODE: and INLINABLE are not used, the word is slower than ROT and DROP. It also shows the use of the 'post-increment with an immediate' addressing-mode.

```
code: rotdrop [
  v, dts, 4 ]i+!, ldr,
  v, dts, str,
] ; 2 inlinable
```

Comparing a conditional branch vs conditional execution.

Both examples perform the same trivial action, namely to check if the top is equal to 23. But one uses a conditional branch and the other uses conditional execution. The latter is the shorter and faster option.

Example with **conditional branch** - ~4 cycles execution time

```
code: 23=
  lbl[ [              \ lvl[ resets the labels
  top, 0 23 i#, cmp,
  top, 0 movw,
  >lb1, bne,          \ branch on not-equal to label1:
  top, 0 0 i#, mvn,
  label1:
] ; 4 inlinable
```

Example with **conditional execution** - ~0-2 cycles execution time

Please note that under certain circumstances this code effectually runs in zero cycles. This happens when code runs while the CPU is still busy executing a slower opcode. Examples are opcodes like DIV, and writing or reading from memory. This is called "executing in the shadow" and is an important optimisation-technique available to the assembly-programmer.

```
code: fast23= [
  top, 0 23 i#, cmp,
  top, 0 0 i#, ne, mov,
  top, 0 0 i#, eq, mvn,
] ; 3 inlinable
```

Example using **MOV**, instead of **LSL**, (and comparable)

```
code: shift16left ( n -- m<<16 ) [
  top, top, 16 lsl#, mov,
] ;
```

Counting leading or trailing zeroes.

This example shows how to use the CLZ and RBIT opcodes to quickly count the number of leading or trailing zeroes within a number. RBIT reverses the order of the 32 bits of a number.

```
code: cntlz ( n - leading_zeroes )
  [ top, top, clz, ] ; 1 inlinable

code: cnttz ( n -- trailing zeroes )
  [ w, top, rbit,
    top, w, clz,
  ] ; 2 inlinable
```

Example with **2@** (**twofetch**) which does NOT need aligned addresses. The **2@** implemented in wabiForth can only read data from a word-aligned address. (see the chapter on Hardware specific wabiFORTH limitations). The following code implements a version of 2@ which is a bit slower but handles unaligned addresses correctly. It is faster than an implementation in wabiForth.

```
code: 2@nonaligned ( addr -- n )
[ v, top, 4 i+]!, ldr,
  w, top, ldr,
  top, v, mov,
  w, dts, 4 i-]!, str,
] ; 4 inlinable
```

Example of an assembly-macro. The following example shows how to make a simple macro. On the ARM-processor getting an 32b immediate into a reg takes two opcodes, MOVW and MOVT. This macro takes a 32b value from stack and generates the two opcodes directly.

```
: ldv32, ( reg n -- )     \ no square brackets!!!
  2dup
  16 lshift 16 rshift movw,
  16 rshift movt,
;
```

Please note that square brackets are not used in a macro!

On a ARMv8 processor **movw**, followed by **movt**, is the fastest way of getting an immediate into a register. On the Cortex-a53, the processor in the RPi 3b+, these two opcodes run in parallel, when put in the order MOVW directly followed by MOVT. And together take only 1 cycle execution-time.

LDV32, can be used as a normal opcode in the assembler. The following silly example shows how to put the number 123456789 into the top register.

```
: tld [ top, 123456789 ldv32, ] ;
```

If your enter:

```
  0 tld .
```

123456789 will be printed as **tld** changes the 0 into 123456789.

Calling other definitions from assembly. In wabiForth, assembly and wabiForth definitions can be mixed at will. This obviates the need to call a wabiForth word from within assembly, you can just mix them. However, if you really want to, it is possible to call a Forth-word from assembly directly.

The easiest way is by using the macro defined above (**ldv32,**) The following example shows how to call the word MYSQ, from assembly, with the help of the macro LDV32.

First lets define MYSQ, which simply squares the top of stack:

```
: mysq ( n -- n^2 )
    dup * ;
```

We can call this definition by getting the XT of MYSQ with TICK, putting the XT into the r0-register and than branch-linking to that address.

```
: sqcall ( n -- n^2 )
[ r0, ' mysq ldv32,        \ put xt of MYSQ in r0
  r0, blx,                 \ branch linked to address in r0
] ;                        \ << do not use inlinable
```

NOTE: this routine is not a primitive as it calls another Forth definition. And so **CODE:** and **INLINABLE** cannot be used!
Also note that there is no need to use **[']**, as assembling is done between brackets.

The macro **LDV32**, itself *can* be part of inlinable code. Remember that it creates two opcodes.

Example of the use of **shifts within operand 2.** The word implements a 1-seed random-generator according to Marsaglia.

First the function is implemented using wabiForth:

```
variable seed
2345 seed !

: forthrandom ( address_seed - rndm_val ) ( 38c )
  dup >r @
  dup 5 lshift xor
  dup 9 rshift xor
  dup 7 lshift xor
  dup r> ! ;
```

The same routine, but now assembly. The forth version is 31 opcodes long once compiled. The assembly-version is only 7 opcodes long and is 3.5 times faster.

```
variable seed
2345 seed !

code: asmrandom ( address_seed - rndm_val ) ( 10c )
[ w, top, ldr,
  w, w, w, 5 lsl#, eor,
  w, w, w, 9 lsr#, eor,
  w, w, w, 7 lsl#, eor,
  w, top, str,
  top, w, mov, ]
; 6 inlinable
```

This example splits a word on top of the stack up into 3 bytes. It shows the use of UBFX, ie unsigned bit-extract. This is a very useful opcode when handling lots of data.
The word SPLIT24 might for instance be useful when handling SPI-addresses, which are send to a peer as 3 bytes, hi, mid and low byte. The assembly routine SPLIT24 takes about 5 cycles. In wabiForth it takes 33 cycles. When speed is essential, and it usually is with SPI-related data-transfers, assembly routines can really help.

Coded using assembly:

```
code: SPLIT24 ( w -- c_lo c_mid c_hi )
[
  w, top, 0 8 ubfx,
  w, dts, 4 i-]!, str,
  w, top, 8 8 ubfx,
  w, dts, 4 i-]!, str,
  top, top, 16 8 ubfx, ]
; 5 inlinable

: tst1 (( 2345678 split24 3drop )) ; \ -> 7.5c
```

The same coded in wabiForth:

```
: split24w ( w -- c_lo c_mid b_hi )
  >r
  r@ 255 and
  r@ 8 rshift 255 and
  r> 16 rshift 255 and
;

: tst2 (( 2345678 split24w 3drop )) ; \ -> ???c
```

Indexing with a **shifted register added to an address** in a register:

```
  ldr r0, [r1, r2, lsl #2]
```

becomes

```
  r0, r1, r2, 2 lsl#, +], ldr,
```

This is useful when going thru an array. Put the base-address in r1, the index in r2 and the data gets loaded into r0.

Example of the **signed byte extend** opcode. It is used to make a primitive which gets the IMMEDIATE flag (a signed byte in the header) for a given XT and converts it to a valid 32b flag.

```
code: getimmediate ( xt -- flag ) [
  v, top, 35 i-], ldrb,
  top, v, sxtb, ]
; 2 inlinable
```

The code above shows the use of the sign_extent opcode. But it is not the optimal code. Using **ldrsb**, a signed byte can be loaded directly:

```
code: getimmediate ( xt -- flag ) [
    v, top, 35 i-], ldrsb, ]
  ; 1 inlinable
```

Adding and subtracting a value to/from a double number. The example shows the use of **ADDS**, and **SUBS**, (instead of **ADD**, and **SUB**,) to set the carry. Also note the two values in front of **i#,** to construct the values 0, 1 and 4. The wabiForth assembler uses the two values separately rather than combine into the one resulting value. Any other value which fits in the 4_8 immediate scheme can be used.

adding 1 to a double:

```
code: 1md+ ( d -- d+1 ) [
  w, dts, ldr,
  w, w, 0 1 i#, adds,
  w, dts, str,
  top, top, 0 0 i#, adc, ]
; 4 inlinable
```

subtracting 4 from a double:

```
code: 4md- ( d -- d-4 ) [
  w, dts, ldr,
  w, w, 0 4 i#, subs,
  w, dts, str,
  top, top, 0 0 i#, sbc, ]
; 4 inlinable
```

Reading data from a 64 bit system-register.
This example reads data from the **PMCCNTR** register. This is a 64 bit counter which counts the actual number of CPU-cycles. The data is put as a double on the stack. The function only needs 3 opcodes to get the double value, raise the stack-pointer by two cells and put the double value on the stack.

```
code: cycles ( -- no_of_cpu_cycles )
[ top, dts, 4 i-]!, str,        \ << this is in fact a dup
  p15, 0 w, top, c9, mrrc,
  w, dts, 4 i-]!, str, ]
; 3 inlinable
```

An energy-saving wait-loop using WFI,

This routine does the same as the example in the **DO...LOOP – FOR...NEXT** chapter. But by adding 1 opcode it becomes very energy-efficient.
The opcode **WFI,** waits for a next interrupt to occur in a system (any interrupt) and then simply continues with the next opcode.
The interesting feature of **WFI,** is that it switches off the CPU-core while waiting. WabiForth activates an interrupt 100 times per second and so 400 results in a wait time of 4 seconds (plus or minus 0,001 %).
Please note that it is irrelevant for the function of **WFI,** why an interrupt was activated. **WFI,** also doesn't influence or change the interrupt in any way.
WFI, is a very powerful concept. It is this opcode (combined with very smart software) which allows an Apple Watch with and ARM-processor, with 2 CPU-cores, running @ 1800 MHz, to run for 20 hours on a small battery. **WFI,** actually ensures that for most of the time the CPU isn't running at all.

```
FOR_PRE
: KEY_TIMEOUT
  400 for
    [ wfi, ]
    key? 0= while
  next
          ." error"
    else
         key emit unloop
    then ;
```

Primitives using high level code

This chapter shows a principle which sometimes can help you save a few CPU-cycles. And on top of that, it is fun to play with. Inlinable primitives normally are made using the assembler. But here it is shown that sometimes it is possible to use nothing but normal wabiForth words and still make a inlinable primitive. This seems a contradiction as a primitive is a word which does not call other words. But inlining sometimes makes it possible.

The principle is this: wabiForth by default inlines as much words as possible. And an inlined word is, per definition, not called anymore.
If a definition for a word has no calls at all to existing words in the dictionary, it is in fact a primitive. And primitives can be made inlinable.

With regard to speed, inlining a definition is only useful if the resulting definition is short, usually not longer than 8 opcodes. Definitions with more opcodes hardly, or not at all, benefit from inlining. This because of the efficient branch-prediction and pre-emptive branching of the ARM-processor. Furthermore, converting a wabiForth definition into a primitive is only useful if the compiler was able to generate efficient code. But sometimes this is the case and then a inlinable primitive can be made using wabiForth words instead of the assembler.

An example is the words BOUNDS, a common word although not in the ANSI-standard. It converts a starting address and length into the upper and lower bounds, ready for use by a DO...LOOP, It is defined thus:

```
: BOUNDS ( addr len -- upper lower ) over + swap ;
```

If you enter '**SEE bounds**' you can see that it compiles to 6 opcodes. Two opcodes for fastprolog and return (called 'fastnext') to the calling word, and four opcodes to handle the actual functionality.

```
see bounds

name:BOUNDS   len:6 lnk:00052D18 hid:0 inl:0 imm:255 wid:0
   1 00052DFC  E92D4000  _-@_     | fastprolog lr only
   2 00052E00  E599C000  ____     | copy 2nd to reg:w
   3 00052E04  E08CB005  ____
   4 00052E08  E589B000  ____
   5 00052E0C  E1A0500C  __P_
   6 00052E10  E8BD8000  ____     | fastnext lr only
```

From the listing it is clear that no calls to other words are made. So there is nothing which prevents the programmer to use CODE: thus:

```
code: boundsc over + swap ; 4 inlinable

see boundsc

name:BOUNDSC  len:7 lnk:00052B04 hid:0 inl:4 imm:255 wid:0
   1 00052BE4  E599C000  ____     | copy 2nd to reg:w
   2 00052BE8  E08CB005  ____
   3 00052BEC  E589B000  ____
   4 00052BF0  E1A0500C  __P_
   5 00052BF4  E12FFF1E  _/__     | codenext
```

The wabiForth version runs in 8 cycles, the version using CODE: runs in 3–4 cycles. The internal version, of course an optimised inlinable primitive, runs in 3 cycles. So in this specific example the compiler found an almost optimal solution. Please note that this is an exception. Usually an internal primitive is faster than a version coded in wabiForth.

The word **>BIT** can also be coded in this way. >BIT returns a number with only the bit at the position n set. So 1 >BIT returns 1, 3 >BIT returns 8 etc. It is a word which might be used when isolating bits from a number.

```
: >BIT ( n -- 2^n )
  1 swap lshift ;

see >bit

name:>BIT   len:4 lnk:00053140 hid:0 inl:0 imm:255 wid:0
   1 00053290  E92D4000  _-@_     | fastprolog lr only
   2 00053294  E300C001  ____
   3 00053298  E1A0551C  __U_     | LSHIFT: top is reg:w..
   4 0005329C  E8BD8000  ____     | fastnext lr only
```

As with the example with **BOUNDS**, there is nothing which prevents us from coding it thus:

```
code: >BITc 1 swap lshift ; 2 inlinable

name:>BITC   len:5 lnk:00053290 hid:0 inl:2 imm:255 wid:0
   1 000532D0  E300C001  ____
   2 000532D4  E1A0551C  __U_     | LSHIFT: top is reg:w..
   3 000532D8  E12FFF1E  _/__     | codenext
```

The words >BIT coded in normal wabiForth takes about 5c to run. The version using **CODE:** and **INLINABLE** runs in 1 cycle, which is faster.

Example of a longer assembly routine

This is the **sieve-benchmark** as published in BYTE in 1983. It counts the number of primes upto 16k and does that 10 times. In 1983 the fastest computer in existence, the CRAY-1 did this, using optimized Fortran, in 0.11 seconds. The Raspberry Pi 3b+, using assembly, does it in 877 micro seconds. Which is 126 times faster than the CRAY-1.
For fairness sake, coded using wabiForth it takes around 1850 micro seconds. Still almost 60 times faster. It clearly shows the unbelievable progress computing has made.

This routine is not picked as an example because of its beauty. Rather because it shows a number of useful things when using the assembler.
For instance it is a nice example of mixing assembly and normal wabiForth words. It shows the use of brackets around blocks of assembly words, even if a block only consists of 1 opcode. It also shows the use of macro's. And how to save and restore registers in an assembly routine.

```
\ the assembler macro's -----

: prologmax r13, {, r0, r4, r-r, r6, r8, r-r, v, w, }!,
  stmfd, ;
: nextmax r13, {, r0, r4, r-r, r6, r8, r-r, v, w, }!,
  ldmfd, ;
: pushdts ( reg -- )      \ pushes reg on top of stack
    top, dts, 4 i-]!, str,
    top, swap mov, ;

lbl[                      \ reset of labels

\ ten times the sieve -----
: asmsieve \ ( odd# -- #primes, mcs )
[ prologmax ]             \ brackets needed!!

  mcs swap
  dup 8 +
  allocate                \ array in high memory

[ r1, top, mov,           \ top in r1
  top, dts, 4 ]i+!, ldr,  \ update stack position

  r6, top, mov,           \ top in r6
  top, dts, 4 ]i+!, ldr,  \ update stack position

  r0, 10 movw,            \ r0=M -> 10 iterations

\ do 10 iterations
label1:
  r2, 0 movw,             \ zero for each iteration
  top, dts, 4 i-]!, str,
  top, r1, mov,           \ r1=start array
```

```
      w, r6, 2 lsr#, mov,     \ w=r6 right shifted by 2
      w, w, 0 1 i#, add,
      top, dts, 4 i-]!, str,
      top, w, mov, ]          \ push content of w on stack

      [hex] 01010101 [decimal]
      fill                    \ flags in array

    [ r3, 0 movw,
      r4, 0 movw,

    label3:
      r7, r1, r3, +], ldrb,   \ get r3'th element
      r7, 0 0 i#, cmp,
      >lb4, beq,

      r7, r3, r3, add,
      r7, r7, 0 3 i#, add,    \ prime is i+i+3
      r8, r7, r3, add,        \ k=prime+i

    label6:
      r8, r6, cmp,            \ if k>odd#
      >lb5, bgt,

      r4, r1, r8, +], strb,   \ flags[k]=0
      r8, r8, r7, add,
      >lb6, b,                \ repeat if k<=no of odd#


    label5:
      r2, r2, 0 1 i#, add,    \ r2=count of primes

    label4:
      r3, r3, 0 1 i#, add,
      r3, r6, cmp,
      >lb3, blt,              \ end sieve-loop

      r0, r0, 0 1 i#, subs,   \ 1 from r0=m
      >lb1, bne,   ]          \ end 10*-loop

      mcs swap -
    [ r2, pushdts ]           \ primes# on stack
      swap
    [ nextmax ] ;
```

Enter:

```
    8190 asmsieve . .
```

and you will get something like 950 and 1899 printed. The first number is the time in microseconds to complete 10 counts. The second is the actual count of primes. It should be 1899...

5CH code

Rationale: Coding in assembly is necessary if you want the fastest routines possible. But assembly code can take up a lot of space in your source-code. It would be a nice feature that, once you are satisfied with a routine in assembly, you are able to make the source-code compacter. **5ch**-code does just that. It allows for opcodes to be put into source-code in a compact fashion.

CAVE: Only code without jumps to other words can safely be converted to 5ch-code. This is because jumps to other words are not always relocatable in wabiForth. So definitions which use a mix of Forth and assembly could easily crash when converted into 5ch-code. Use SEE to check whether a definition contains any jumps to other words.

5CHDUMP (addr, n --): converts n opcodes starting at addr into a 5ch code of 1 or more lines and prints these lines. These lines can then be copied and pasted into source-code.

5C| (--) converts a string of 5ch-code into opcodes, and commas these at HERE and onwards in a cache-safe way. This word is usually used by creating lines of 5ch-code using 5CHDUMP.

The example here shows how the word fast23= looks when it is converted into 5CH code. It is also shows that making it inlinable is still possible.

Original:

```
code: fast23= [
  top, 0 23 i#, cmp,
  top, 0 0 i#, ne, mov,
  top, 0 0 i#, eq, mvn,
] ; 3 inlinable
```

5C| version:

```
code: fast23= 5c| S|__k%FB`X!qp$s ; 3 inlinable
```

Cache-handling

CACHE-HANDLING

All cache-handling is normally done by wabiForth. The caches are initialised during the boot process. When words are added to the dictionary, or when words are removed from the dictionary using FORGET, wabiForth handles the coherency between the data and instruction-caches and handles the branch-predictor- and translation table caches.

LIMITATIONS OF CACHE-SETUP

The setup of a cache-system is always a balancing-act between performance, functionality and reliability. Although the system-design of wabiFORTH usually focusses on performance, in case of the setup of the cache-system reliability was prime. Therefore it was decided to run wabiFORTH with full coherence between the different cores. As such this is a bit slower than without coherence. But the big advantage is that wabiFORTH is 'rock-solid'. For instance: adding words using **EVALUATE** to a running program from within that program is possible, without fear of ABORTS or crashes due to coherency-problems.

There is one user-word related to cache-handling:

CLEANCACHE: (--): flushes the data-cache, invalidates the data and instruction-caches, and invalidates the branch-predictor, translation table and look-aside buffers. The result is that all data in the memory is up-to-date, and all caches are filled afresh when data, opcodes or branches are accessed in memory.

The biggest benefit of the word is that **it creates a known starting-condition**. This is especially useful when doing performance measurements as it raises the consistency of measurements. And obviously if you want to experiment with the effect of caches on a program, it might have some value. CLEANCACHE is a pretty slow word, it can easily take more than 90 thousand cycles to complete. So you do not want to use it in fast loops.

The other area where the word might be useful is when using external agents like the DMA-controllers. External agents act independent of the CPU and thus do not know whether data in the memory is valid or invalid. By using the word CLEANCACHE a known starting state of the caches is created.

ARM-processor related functionality

WabiForth contains rudimentary functionality to get values from a few specific registers in the ARM-processor. The registers provide data on the features and status of the processor. For more information see the ARMv8 documentation of the ARM-corporation.

CPU_PARWRITE: (addr -- n) for a given address returns the memory attributes for a write-action to that address. The address itself is not changed.

CPU_PARREAD: (addr -- n) ditto for a read-action from that address.

CPU_ID_MMFR0: (-- n) returns content of CPU-register ID_MMFR0 - describes memory-features
CPU_ID_MMFR1: (-- n) ditto for ID_MMFR1
CPU_ID_MMFR2: (-- n) ditto for ID_MMFR2

CPU_ID_DFR0: (-- n) returns content of CPU-register ID_DFR0 - gives top level information about the debug system

CPU_CTR: (-- n) returns content of CPU-register CTR = cache-type-register

CPU_PMCEID0: (-- n) returns content of CPU-register PMCEID0
CPU_PMCEID1: (-- n) ditto for PMCEID1

CPU_PMCCFILTR: (-- n) returns content of CPU-register PCCFILTR
CPU_PMCR: (-- n) returns content of CPU-register PMCR
CPU_CNTFRQ: (-- n) returns content of CPU-register CNTFRQ = frequency of the counter
CPU_CPSR: (-- n) returns content of CPU-register CPSR = current processor state register
CPU_FPSID: (-- n) returns content of CPU-register FPSID = describes floating point processor features

CPU_DFAR: (-- n) returns content of CPU-register DFAR = Data Fault Status Register
CPU_DFSR: (-- n) ditto for DFSR = Data Fault Address Register

The ID_ISARn registers of the ARM-processor describe the available features of the specific processor wabiForth is running on. Again, see the ARM documentation for technical details.

CPU_ID_ISAR0: (-- n) returns content of CPU-register ID_ISAR0
CPU_ID_ISAR1: (-- n) ditto for ID_ISAR1
CPU_ID_ISAR2: (-- n) ditto for ID_ISAR2
CPU_ID_ISAR3: (-- n) ditto for ID_ISAR3
CPU_ID_ISAR4: (-- n) ditto for ID_ISAR4
CPU_ID_ISAR5: (-- n) ditto for ID_ISAR5

Air Traffic Controller

```
(*
 Air Traffic Controller
 Based on a game by Dave Mannering for Creative Computing 1978-1981
 This original wabiForth version (C) 2022 - J.J. Hoekstra
 Requires wabiForth 4.1.0 or higher
 28852 bytes 184 definitions ( ?? bytes noinline )
*)

forget atcshield : atcshield ; word# unused

\ constants ******************************
25 constant xdir
15 constant ydir
27 constant #plane  \ 0=no_plane, 1=plane A, etc.
16 constant column

\  0 constant pstat \ { state-engine plane (0/1/2/3) - 0=inactive - 1=proceed,
        2=waiting
\  1 constant ppj   \ prop/jet - (0/1)
\  2 constant phgt  \ height (0-15)
\  3 constant porig \ origin (0-11) - 1 of 10 entry-points or 1 of 2 airfields
\  4 constant pdest \ destination (0-11) - cannot be full rndm
\  5 constant pxpos \ x (0-24) - the origin - ditto
\  6 constant pypos \ y (0-14) - ditto - ditto
\  7 constant pdir  \ direction (0-7)
\ 10 constant pstrt \ aligned 16 bit number - start number 99*4 -> meer als 8 bits
\ 12 constant pxnew \ newx (0-24) - the destinaton position
\ 13 constant pynew \ newy (0-15) - ditto
\ 14 constant phnew \ new height (0-15) -ditto }

 0 constant oexit
 1 constant oxpos
 2 constant oypos
 3 constant odirc

12 constant origplane
 4 constant origcol

\ values ********************************** {
30 value #mins
#mins 4* value #steps

 0 value escflag
 0 value opd4plane
 0 value opdracht
 0 value getal
 0 value microstep
 0 value prop_phase
 0 value presentstep
 2 value screenwrite
 5 value screenshow
 0 value linesprinted
 0 value dodemo

\ string literals *************************
 s" N NEE SES SWW NW" sliteral $directions

\ arrays *********************************
create [planes] #plane column * allot
create [orig] origplane origcol * allot
create $input 16 allot
create $seed 16 allot                       \ }
```

```
\ routines ********************************
: defwins   \ {
    \ window 2 - field A of radar
        790 496 2 makewin
           gray 2 >wincanvas
                2 wincls
              3 2 >winrand                  \ grijze rand
    translucent 2 >wincanvas
                2 wincls
          white 2 >winink
           true 2 winvisible                \ translucent

    \ window 5 - field B of radar
        790 496 5 makewin
           gray 5 >wincanvas
                5 wincls
              3 5 >winrand                  \ grijze rand
    translucent 5 >wincanvas
                5 wincls
          white 5 >winink
          false 5 winvisible                \ translucent

              2 0 win#>task#

    \ windows 1 - copyright etc
          false 1 winvisible

    \ window 0 - achtergrond
          black 0 >wincanvas
          white 0 >winink
                0 wincls
                0 winhome

    \ window 3 - lijst met active viegtuigen
        230 496 3 makewin
          794 0 3 >winorig
       vlyellow 3 >wincanvas
                3 wincls
              4 3 >winrand
    translucent 3 >wincanvas
                3 wincls
          black 3 >winink
           true 3 winvisible

    \ window 4 - orders
        610 264 4 makewin                   \ was 790 264
          0 500 4 >winorig
           gray 4 >wincanvas
                4 wincls
              3 4 >winrand                  \ grijze rand
    translucent 4 >wincanvas
                4 wincls
              7 4 >winrand                  \ marge voor duidelijkere text
          white 4 >winink
           true 4 winvisible

    \ window 6 - various
        410 264 6 makewin
        614 500 6 >winorig
           gray 6 >wincanvas
                6 wincls
              3 6 >winrand                  \ grijze rand
    translucent 6 >wincanvas
                6 wincls
          white 6 >winink
```

```
          true 6 winvisible
; \ }
: showscreen2                               \ { for double-buffering
    2 to screenshow 5 to screenwrite
    true 2 winvisible false 5 winvisible ;

: showscreen5                               \ for double-buffering
    5 to screenshow 2 to screenwrite
    true 5 winvisible false 2 winvisible ;

: switchscreens                             \ synced screen-flip
    begin scr_sync until                    \ flip if screen sync = 'true'
    screenwrite 2 = if showscreen2
    else showscreen5 then ;                 \ }
: switchpropphase prop_phase 1 xor to prop_phase ;

0 value timera
: resettimer ( -- ) centis to timera ;
: gettimer ( -- n ) centis timera - ;

: win4 4 0 win#>task# ;
: win6 6 0 win#>task# ;
: win4emit win4 emit win6 ;
: win4bs win4 backspace win6 ;
: win4cr win4 cr win6 ;

: *[planes] ( p col -- addr ) swap column * [planes] + + ;
: plane@    ( p col -- n )    *[planes] sc@ ;
: plane!    ( n p col -- )    *[planes] c! ;
: 16plane@  ( p col -- n )    *[planes] h@ ;
: 16plane!  ( n p col -- )    *[planes] h! ;

: stat@     ( p -- dest ) 0 plane@ ;            \ { = pstat
: stat!     ( p -- dest ) 0 plane! ;

: prop@     ( p -- dir ) 1 plane@ ;             \ = ppj
: prop!     ( n p -- ) 1 plane! ;

: height@   ( p -- height ) 2 plane@ ;          \ = phgt
: height!   ( n p -- ) 2 plane! ;

: orig@     ( p -- dest ) 3 plane@ ;            \ = porig
: orig!     ( d p -- ) 3 plane! ;

: dest@     ( p -- dest ) 4 plane@ ;            \ = pdest
: dest!     ( d p -- ) 4 plane! ;

: xpos@     ( p -- dest ) 5 plane@ ;            \ = pxpos
: xpos!     ( d p -- ) 5 plane! ;

: ypos@     ( p -- dest ) 6 plane@ ;            \ = pypos
: ypos!     ( d p -- ) 6 plane! ;

: direc@    ( p -- dir ) 7 plane@ ;             \ = pdir
: direc!    ( n p -- ) 7 plane! ;

: xnew@     ( p -- dir ) 12 plane@ ;            \ = pxnew
: xnew!     ( n p -- ) 12 plane! ;

: ynew@     ( p -- dir ) 13 plane@ ;            \ = pynew
: ynew!     ( n p -- ) 13 plane! ;

: pstart@   ( p -- start ) 10 16plane@ ;        \ = pstrt
: pstart!   ( p -- start ) 10 16plane! ;

: newhght@  ( p -- height ) 14 plane@ ;         \ = phnew
```

```
: newwhght!  ( n p -- ) 14 plane! ;              \ }

: newxy@ ( plane -- x y ) >r r@ xnew@ r> ynew@ ;
: xy@ ( plane -- x y ) >r r@ xpos@ r> ypos@ ;

: planestat! opd4plane stat! ;

: planeactive? ( p -- t/f ) stat@ 0<> ;
: planemoving? ( p -- t/f )
    stat@ dup 0<> if
        2 5 within if
            false
        else
            true
        then
    else
        drop false
    then
    ;
: .plchar ( p -- ) 64 + emit space ;
: >orig origcol * [orig] + >r r@ 3 + c! r@ 2 + c! r@ 1 + c! r> c! ;
: clear[planes] [planes] #plane column * 0 fill ;

: fill[orig] \ {
\  cn  x  y  dr #
    9  0  8  2  0 >orig
    8  5  0  4  1 >orig
    7 10  0  3  2 >orig
    6 18  0  4  3 >orig
    5  0  3  3  4 >orig
    4 11 14  7  5 >orig
    3 18 14  0  6 >orig
    2 24 14  7  7 >orig
    1  5 14  0  8 >orig
    0 24  8  6  9 >orig
    0 10  8  6 10 >orig
    0 14  4  7 11 >orig
    0 18  8  6 12 >orig
    0  5  8  0 13 >orig ; \ }
create movetbl  \ {
     0 c, -1 c, \ N
     1 c, -1 c, \ NE
     1 c,  0 c, \ E
     1 c,  1 c, \ SE
     0 c,  1 c, \ S
    -1 c,  1 c, \ SW
    -1 c,  0 c, \ W
    -1 c, -1 c, \ NW }

: (xstep) ( dir -- step )
    dup 0 8 within if 2* movetbl + sc@
    else cr ." illegal value in (xstep): " . abort then ;
: (ystep) ( dir -- step )
    dup 0 8 within if 2* movetbl + 1+ sc@
    else cr ." illegal value in (ystep): " . abort then ;

: clear$seed $seed 5 32 fill ;
: fill$seed cr ." please enter a 5 character SEED: " $seed 5 accept drop ;
: $>seed clear$seed fill$seed $seed 5cval ;
: setseeds $>seed dup rndmseed ( rndm32 space . ) ;
: origtbl@ ( orig ocol -- n ) swap origcol * [orig] + + sc@ ;   \ test of sc@

\ {
: (coord2xy) ( x y -- sx sy )
    30 * 34 + swap 30 * 34 + swap ;
: (.line) ( x y x y -- )
```

```
    green 0 >winink 0 winline white 0 >winink ;
: .alllines 5 0 do
      i oxpos origtbl@     i oypos origtbl@     \ start line
    (coord2xy) 1+ swap 1+ swap
    9 i - oxpos origtbl@   9 i - oypos origtbl@ \ end line
    (coord2xy) 1+ swap 1+ swap
    (.line) loop ;
: (.bbox) ( x y -- ) 4 - swap 4 - swap 11 11 0
    black 0 >winink drawbox white 0 >winink ; hidden
: (.bdot) ( x y -- ) (coord2xy) 2dup (.bbox) 3 3 0 drawbox ; hidden

: .alldots ydir 0 do xdir 0 do i j (.bdot) loop loop ; hidden
: .bluelines lblue 0 >winink 24 1 do
     805 i 20 * 19 + 208 0 horline loop ; hidden
: (.1char) ( char orig -- ) \ prints char airfields/navi-beacon
    >r r@ oxpos origtbl@
       r> oypos origtbl@
       (coord2xy)
       7 - swap 3 - swap
       0 winchar ; hidden
: .4chars 130 10 (.1char) 131 11 (.1char) 132 12 (.1char) 132 13 (.1char) ; hidden
: .entries 10 0 do
       i 48 + i (.1char)
    loop ;
: .whitesheet vlyellow 0 >winink 798 0 228 494 0 drawbox .bluelines
       ; hidden
: .paintfield black 0 >winink 0 wincls
    .alllines .alldots .4chars .entries .whitesheet
    ; hidden
: .origdest ( o/d -- )
    dup 0 10 within if ._ else
        10 = if ." #" else ." %" then then
    ;
: .drctn ( d -- ) 2* $directions drop + 2 type
       ; \ }
: .plane ( number -- )                      \ {
    >r
    r@ 64 + emit                            \ =plane-character without space
    r@        prop@
              0= if ." P"
              else  ." J"
              then
    r> height@ .hex
    ;                                              \ height-character }
: .planel ( number -- )                     \ { help function
    >r
    r@              .plane
    r@ orig@        .origdest
                    ." ->"
    r@ dest@        .origdest
                    space
    r@ xpos@        .2d_
        [char] .    emit
    r@ ypos@        .2d_
                    space
    r@ direc@       .drctn

    rdrop
    ;                                              \ }
: backtonormal                              \ {
    0 0 win#>task# vdcyan 0 >wincanvas
    white 0 >winink rstwins cls
    ;                                                          \ }
: drawsqrplane ( x y )                      \ { draws square under plane char
    black screenwrite >winink
    8 - swap 10 - swap 19 17 screenwrite drawbox
```

```
    white screenwrite >winink
    ;                                                                   \ }
: (mcstx) ( x p - new_x )                   \ { calc microstep in x-dir
    dup direc@ (xstep) >r                   \ ( x p ) r:step -1, 0 or 1
    prop@                                   \ jet or prop
    if                                      \ if jet... ( x p )
        r> microstep * +                    \ ( x+microstep*dir )
    else                                    \ ( x p )
        prop_phase if                       \ prop_phase ( 0/1 ) for prop-planes
            r@ microstep *                  \ calc actual x for a prop second phase
            r> 30 * + 2 / +                 \ ditto
        else
            r> microstep * 2 / +            \ ditto first phase
    then then
    ;
: (mcsty) ( y p - new_y )                   \ calc microstep in y-dir
    dup direc@ (ystep) >r                   \ ( y p ) r:step -1, 0 or 1
         prop@                              \ jet or prop
    if                                      \ if jet... ( y p )
        r> microstep * +                    \ ( y+microstep*dir )
    else                                    \ ( y p )
        prop_phase if                       \ prop_phase ( 0/1 ) for prop-planes
            r@ microstep *                  \ calc actual y for a prop second phase
            r> 30 * + 2 / +                 \ ditto
        else
            r> microstep * 2 / +            \ ditto first phase
    then then ;
: (addmicrosteps) ( x y i -- new_x new_y )
    >r r@ (mcsty) swap r> (mcstx) swap
    ;                                                                   \ }
: (drawplane) ( x y plane# -- )             \ {
        >r
        2dup drawsqrplane                   \ done with black to winink
        2dup r@ 64 + -rot                   \ x y ch x y
        9 - swap 9 - swap
        screenwrite winchar                 \ 1st char printed ( x y )
        9 - r> height@ 48 + -rot
        screenwrite winchar                             \ 2nd char printed
    ;                                                   \ }
: .planesfield                              \ {
    screenwrite wincls
    black screenwrite >wincanvas
    #plane 1 do
      i planeactive? if
        i xy@ (coord2xy)                    \ scr_x scr_y
        i planemoving? if                   \ add microsteps if not waiting at airport
            i (addmicrosteps)
        then
        i (drawplane)
      then
    loop
    translucent screenwrite >wincanvas
    ;                                                                   \ }
: myinput ( -- len )
        $input 4+ 8 accept dup $input !
        ;
: checkmin ( -- 0/value )
    0 $input >number 0<> if drop 0 else then
    ;
: input#min ( -- 18-99 ) ." How long should the game last? (18 - 99 min): "
    myinput 0= if 25 else checkmin then     \ default of 25 min on return
    18 99 clip 4* to #steps
    ;
s" ALRHMPS#%" sliteral orders$
\ : makeopdracht ( char -- )                \ codes the tasks 1-9 with 0 for invalid
\    0 to opdracht
```

```
\    >caps
\    orders$ bounds do
\        dup i c@ = if
\            i orders$ drop - 1+ to opdracht
\            leave
\        then
\    loop drop
\    ;
: makeopdracht ( char -- )                  \ codes the tasks 1-9 with 0 for invalid
    >caps orders$ instring 1+ to opdracht
    ;
: checknumber ( char -- f/t )               \ also sets getal
    >caps 48 -
    dup 0 6 within if
        to getal true
    else
        drop -1 to getal false
    then
    ;
\ ** state engine input-handler
\  0 -> 0 chars - waiting for A-Z for plane - goto 1 - bs not possible
\  1 -> 1 chars - waiting for "A, L, R" for opdracht -> goto 2 - bs -> goto 0
\           Hold, Maintain, Proceed, Status, approach_#, approach_%, -> goto 3
\  2 -> 2 chars - waiting for 1-5 for opdracht -> goto 3 - bs -> goto 1
\  3 -> 2/3 chars received - do opdracht, cr, goto 1 - bs -> goto 2

0 value is=
: input_handler ( key -- )
    dup 8 = if                              \ backspace!
        is= 1 = if
            win4bs
            0 to is=
        then
        is= 2 = if
            win4bs
            1 to is=
        then
        drop exit
    then
    dup 0 33 within if drop exit then       \ exit on all ctrl chars and space

    >r r@ win4emit
    is= 0 = if r> 64 - to opd4plane 1 to is= exit then
    is= 2 = if r> 48 - to getal     3 to is= exit then
    is= 1 = if r> makeopdracht
        opdracht 4 < if
            2 to is=                        \ if opdracht 1, 2, 3 -> 3 chars input
        else
            3 to is=                        \ if opdracht > 3 -> 2 chars input
        then exit
    then
    ;
: fromairfield ( p -- )                     \ {
    >r 0 r@ stat!                           \ =waiting at airfield
    40 rndm 10 < if                         \ 25/75%
         11 r@ orig!                        \ orig=11
    else
         10 r@ orig!                        \ orig=10
    then

    12 rndm r@ dest!                        \ dest 0-11 (inc airfield)
                                            \ #TBD: bias for busy routes
    0 r@ height!
    0 r> newhght!
    ;                                                   \ }
: overflight ( p -- )                       \ {
```

```
    >r
    10 rndm dup
        r@ orig!                            \ orig 0-9
    9 swap - r> dest!
    ;                                                \ }
: toairfield ( p -- )                       \ {
    >r
    10 rndm r@ orig!

    4 rndm 1 < if                           \ 25/75%
        11 r> dest!
    else
        10 r> dest!
    then
    ;                                               \ }
: makeorides ( plane -- )                   \ { origin & destination
    >r                                      \ plane
    6 rndm dup                              \ 0-5

    2 < if drop                             \ 0, 1
        r> fromairfield                     \ from airfield 33%
    else
        4 > if                              \ 5
            r> overflight                   \ overfligth 16%
        else                                \ 2, 3, 4
            r> toairfield                   \ to airfield 49%
        then
    then
    ;                                               \ }
: fillxyd ( plane -- )                      \ {
    >r
    r@ orig@                               \

    dup oxpos origtbl@  r@ xpos!
    dup oypos origtbl@  r@ ypos!
        odirc origtbl@  r@ direc!
    rdrop
         ;                                          \ }
0 value step*100    0 value rndmfact
: correctstart ( -- )                       \ {
    1 pstart@                               \ get 1st starttime
    #plane 2 do                             \ for all other planes
        dup                                 \ ( lower lower )
        i pstart@                           \ ( lower lower higher )
        > if                                \ ( lower )
            dup i pstart!
        else
            drop i pstart@                  \ get next lower
        then
    loop drop ;                             \ }

0 value tempstart   0 value temporig    0 value tempplane
: goback ( pl relevante_orig startt -- height ) \ {
    to tempstart
    to temporig
    to tempplane

    1 tempplane 1- do                       \ negative do can have equal inputs
        temporig   i orig@ = if             \ origs equal?
            tempstart                       \ starttime of plane being checked
            i pstart@                       \ starttime of lower plane
            - 13 <                          \ calculate diff and check
            if                              \ if < 13 (=3 min)
                i height@                   \ if smaller -> get hight of lower plane
                1+ dup                      \ add 1 to it and ready
                tempplane height!
```

```
                tempplane newwhght!
            then
            leave                           \ always leave if origs were equal
        then
    -1 +loop
    ;
: checkheights ( -- )
    #plane 2 do                             \ plane 1 cannot be to low
        i orig@ >r                          \ orig_of_plane being checked in R
        r@ 0 10 within if                   \ only entry-points - not airfields
            i                               \ plane
            r@                              \ orig_of_plane
            i pstart@                       \ starttime of entry of this plane
            goback                          \ for this plane go back to check
         startttimes
        then rdrop
    loop
    ;
: fillstarttime
    #steps 60 - 100 * 26 / to step*100      \ last plane 15 min before eot
    step*100 95 100 */ to rndmfact          \ larger rndmfact -> more piling of planes
    0 1 pstart!                             \ first plane always at step 0

    #plane 1- 2 do
        i step*100 *
        rndmfact rndm
        2 rndm 0 = if - else + then
        100 / i pstart!
    loop
    step*100 #plane * 100 / 26 pstart!      \ last plane always 15 min before eot
    correctstart                            \ make sure that planes start in order of
         number
    checkheights                            \ avoid collision of 2 planes at entry }
    ;
\ : testdata                                  \ { #TEST!!
\    20  6 stat!                             \ plane F turns @ * to land @ #
\    40  8 stat!                             \ plane HP
\    30 22 stat!                             \ plane VP holds at next beacon
\    40 19 stat!                             \ plane SP
\    11 19 dest! ;                           \ plane SP change dest }
: defplanes                                 \ {
    \ setseeds
    clear[planes]
    #plane 1 do      0 i stat!
        3 rndm 0= if 1 i prop! then         \ was 25% jets
                     6 i height!
                     6 i newwhght!
                       i makeorides
                       i fillxyd
    loop
    fillstarttime
    ( testdata ) ;                              \ }
: gameinit                                  \ {
    rndmreset
    defwins
    fill[orig]
    defplanes
    .paintfield
    uartclear

    2 to screenwrite                        \ setup of double buffering
    5 to screenshow

    6 0 win#>task#                          \ 6=other 4=entry screen

    0 to presentstep
```

```
    ;                                                   \ }
: .upcommingplanes                          \ { prints list of planes becoming active
    2 to linesprinted
    3 winhome
    3 0 win#>task#
    true 0 uart>task#

    ." upcomming planes   " cr
    #plane 1 do
        i pstart@
        presentstep -
        1 3 within if                       \ 1 5 -> planes next minute
            i space .planel cr
            1 +to linesprinted
        then
    loop
    linesprinted 2 = if
        1 spaces ." ---" 16 spaces
    else
        20 spaces
    then
    win6
    false 0 uart>task#
         ;                                                  \ }
: .activeplanes                             \ {
    3 0 win#>task#
    true 0 uart>task#

    2 +to linesprinted
    cr ." active planes       "
    #plane 1 do
        i stat@ 0 > if
            cr i space .planel
            1 +to linesprinted
        then
    loop

    linesprinted 24 < if
        25 linesprinted do
            cr 20 spaces loop
    then
    win6
    false 0 uart>task#
;                                                   \ }

\ *************  from here game logic *******

: getxstep ( p -- step ) direc@ 2* movetbl + sc@ ;
: getystep ( p -- step ) direc@ 2* movetbl + 1+ sc@ ;
: revdir ( dir -- dir+180graad ) 4 + 7 and ;
: (leftdir) ( dir -- dir+180graad ) 1 - 7 and ;
: (rightdir) ( dir -- dir+180graad ) 1 + 7 and ;
: leftdir ( p -- )
        >r r@ direc@ (leftdir) r> direc! ;
: rightdir ( p -- )
        >r r@ direc@ (rightdir) r> direc! ;
: (makenewxy) ( plane -- ) >r
    r@ getxstep r@ xpos@ +
    r@ xnew!                                \ new x
    r@ getystep r@ ypos@ +
    r> ynew!                                \ new y
    ;
: makenewxy ( plane -- ) >r
    prop_phase r@ prop@ + if                \ 0=prop/1=jet -> 0/1/2 -> if 1 or 2 -> do
         plane
        r@ (makenewxy)
```

```
    then rdrop                              \ 0=> altijd prop met prop phase 0
    ;
: boundaries? ( x y -- f/t )                \ true if x, y is outside of boundaries
    0 ydir within swap 0 xdir within and 0=
    ;
: planeatexit ( p -- f/dest t )
    >r
    r@ dest@ dup                            \ 2*destination of plane
    oxpos origtbl@ r@ xpos@ =               \ compare x of plane and x of dest
    swap
    oypos origtbl@ r@ ypos@ =               \ ditto for y
    and if                                  \ combine flags
        r> dest@ true
    else
        rdrop false
    then
    ;
: correctexit? ( p -- f/t )                 \ { check correct x, y and direction
    >r r@ planeatexit if                    \ f/orig t
        odirc origtbl@  revdir              \ ( r_dir ) -> reversed entry direction of
         orig
        r> direc@ =                         \ ( r_dir dir ) -> compare 2 directions
    else
        rdrop false
    then
    ;                                       \ }
: deactivateplane ( p -- ) 0 swap stat! ;   \ checked
: activateplane   ( p -- )                              \ for planes from orig to
         airfield autohold=state=30
    >r r@ orig@ 10 < 2 + r@ stat!                \ 1=active or 2=waiting at an
         airport

    r@ orig@ 10 <
    r@ dest@ 9 > and if                                     \ if orig < 10 and
         dest > 9 -> autohold
         30 r@ stat!
    then
    rdrop
    ;
: adaptheight ( p -- )
    >r
    r@ planemoving? 0= if                   \ exit if plane does not move
        rdrop exit
    then

    r@ newwhght@ r@ height@ -
    dup 0> if
        r@ height@ 1+ r> height! drop exit
    then
    0< if
        r@ height@ 1- r> height! exit
    then
    rdrop
    ;
: prepplanes ( -- ) \ { new xy in destination active planes & activates planes
    #plane 1 do                             \ for all planes
        i stat@ 0= if
            i pstart@ presentstep = if
                ." activating plane "       \ debug
                i .plchar cr                \ debug
                i activateplane
            then
        then

        i planemoving? if                   \ only moving planes
            i makenewxy
```

```
        then

        i stat@ 3 = if                      \ for 3=permission to start
            prop_phase 0= if                \ wait till prop-phase=0
                1 i stat!                   \ goto 1=proceed
                1 i height!
            then
        then

        i stat@ 4 = if                      \ for 4=permission to start
            3 i stat!                       \ goto 3=permission to start
        then
    loop
    ;                                       \ }
: updatexy ( p -- )                         \ { newxy of moving planes to xpos/ypos
    >r r@ newxy@ r@ ypos! r> xpos!          \ }
    ;
: field#? ( x y -- f/t ) 8 = swap 10 = and 0<> ;
: field%? ( x y -- f/t ) 4 = swap 14 = and 0<> ;

: abortlanding? ( p -- )
    >r r@ stat@ 40 =
    r@ height@ 0 = or if                    \ trying to land? (state=40 or height=0)
        1 r@ stat!
        1 r@ height!
        1 r@ newhght!
        ." plane " r@ .plchar ." aborting landing" cr
    then rdrop
    ;
: landed? ( p -- f\t )                      \ {
    >r
    r@ planemoving? 0= if                   \ not landing if not moving
        rdrop false exit then

    r@ xy@ field#? if                       \ west direc=6
        r@ direc@ 6 =  r@ height@ 0 =
        and r@ dest@ 10 = and 0<> if        \ landed @ #!
            rdrop true exit
        else                                \ not landed...
            r@ abortlanding?
            rdrop false exit
        then
    then

    r@ ( newxy@ ) xy@ field%? if            \ north west direc=7
        r@ direc@ 7 =
        r@ height@ 0 =
        and
        r@ dest@ 11 =
        and 0<> if                          \ landed @ %!
            rdrop true exit
        else                                \ not landed...
            r@ abortlanding?
            rdrop false exit
        then
    then
    rdrop false                             \ }
    ;
: (landed) ( p -- )
    dup deactivateplane
    ." plane " .plchar ." landed" cr        \ }
    ;
: handleplanestep ( p -- f\t )              \ checks landing, exit and updates XY {
    >r
    r@ landed? if
        r> (landed) true exit then
```

```
    r@ newxy@ boundaries? if                \ if boundaries passed with updated xy
        r@ correctexit?                     \ passed at valid exit with correct dir...
        r@ height@ 5 =                      \ passes at valid height?
        and if
            r@ deactivateplane
            ." plane " r@ .plchar ." left your area" cr true
        else drop
            r@ deactivateplane              \ debugging
            ." BOUNDARY ERROR plane " r> .plchar cr false exit
        then
    then
    rdrop true                              \ }
    ;
: checkplanes ( -- f/t )                    \ { check for
    #plane 1 do
        i planeactive? if
            i handleplanestep 0= if
                i deactivateplane
                false ( unloop exit )       \ debugging
            then
        then
    loop true
    ;                                                                   \ }
: updateplanes ( -- )                       \ {
    #plane 1 do
        prop_phase i prop@ + if             \ 0=prop/1=jet -> if 1 or 2 -> do plane
            i planemoving? if               \ only moving planes
                i updatexy
            then
        then
    loop
    ;                                                                   \ }

( state_engine ) \ ************************ {
\ (*
\    0 -> not active - goto 0
\    1 -> proceed - goto 1
\    2 -> wait for take-off at airfield - goto 2
\    3 -> prepare for takeoff in propphase=1 - goto 1
\    4 -> prepare for takeoff in propphase=0 - goto 3
\   10 -> turn  45 left  - goto  1
\   11 -> turn  90 left  - goto 10
\   12 -> turn 135 left  - goto 11
\   13 -> turn 180 left  - goto 12
\   14 -> turn  45 right - goto  1
\   15 -> turn  90 right - goto 14
\   16 -> turn 135 right - goto 15
\   17 -> turn 180 right - goto 16
\   20 -> at * turn sharp to # airfield - goto 20 or 1
\   21 -> at * turn sharp to % airfield - goto 21 or 1
\   29 -> circle 45 left - goto 29
\   30 -> at * start to circle - goto 30 ( => * not yet reached ) or 29
\   40 -> goto height=0 and start landing - goto 40 - goto 1 on wrong landing
\ *)

: dolanding ( p -- )
    >r r@ height@ 1- 0 max r@ height!       \ lowers height with 1
    0 r> newhght! ;                         \ newheight always 0
: (@navbeacon?) ( x y -- t/f )              \ { 5,8 - 18,8
    8 = if dup 5 = swap 18 = or 0<>
    else drop false then ; \ #checked }
: @navbeacon? ( p -- )                      \ state=30
    >r r@ xy@ (@navbeacon?) if
        r@ leftdir                          \ start circling
        29 r@ stat!                         \ state engine to 29
```

```
    then rdrop ;
: gotofield# ( p -- )                       \ state=20
    >r r@ xy@ (@navbeacon?) if
        6 r@ direc!                         \ goto airfield%
        40 r@ stat!
    then rdrop ;
: gotofield% ( p -- )                       \ state=21
    >r r@ xy@ (@navbeacon?) if
        7 r@ direc!                         \ goto airfield%
        40 r@ stat!
    then rdrop ;
: t180l ( p -- )                            \ { state=13
    >r r@ leftdir 12 r> stat!   ;
: t135l ( p -- )                            \ state=12
    >r r@ leftdir 11 r> stat!   ;
: t90l ( p -- )                             \ state=11
    >r r@ leftdir 10 r> stat!   ;
: t45l ( p -- )                             \ state=10
    >r r@ leftdir 1 r> stat!    ;
: t180r ( p -- )                            \ state=17
    >r r@ rightdir 16 r> stat!  ;
: t135r ( p -- )                            \ state=16
    >r r@ rightdir 15 r> stat!  ;
: t90r ( p -- )                             \ state=15
    >r r@ rightdir 14 r> stat!  ;
: t45r ( p -- )                             \ state=14
    >r r@ rightdir 1 r> stat!
    ;                                       \ }
: updatedoing ( plane -- )                  \ {
    >r r@ stat@
    dup 40 = if drop r> dolanding   exit then \ state=40 can never be undone
    dup 30 = if drop r> @navbeacon? exit then \ start circling when at *
    dup 29 = if drop r> leftdir     exit then \ continue circling
    dup 20 = if drop r> gotofield#  exit then \ if on navbeacon -> direction = west
    dup 21 = if drop r> gotofield%  exit then \ if on navbeacon -> direction = west
    dup 13 = if drop r> t180l       exit then
    dup 12 = if drop r> t135l       exit then
    dup 11 = if drop r> t90l        exit then
    dup 10 = if drop r> t45l        exit then
    dup 17 = if drop r> t180r       exit then
    dup 16 = if drop r> t135r       exit then
    dup 15 = if drop r> t90r        exit then
    dup 14 = if drop r> t45r        exit then
    drop rdrop
    ;                                       \ }
: updatestateengine ( -- )                  \ {
    #plane 1 do
        prop_phase i prop@ + if             \ every other cycle for prop
            i planeactive? if
                i updatedoing               \ do action and make new_doing in table
                i adaptheight
            then
        then
    loop
    ;                                       \ }
: startmodule   ( -- f/t )                  \ {
    win4 cls uartclear
    ." play game? (y/n)"
    key [char] n = if
        backtonormal true exit
    then false win6
    ;
: userinput ( -- f/t )
    win4 cr input#min
    fill$seed gameinit win6
    ;
```

```
: endmodule ( -- f/t )
    win4 uartclear
    cr ." play another game? (y/n)"
    key [char] n = if
        true
    else
        gameinit false
    then win6
    ;                                       \ }
         : .debuginput
    win4
    space ." -> " ." pl: " opd4plane .
    ." - opd: " opdracht .
    ." - num: " getal .
    win6
    ;
: checkopdracht ( -- f/t )                  \ false on inactive plane
    .debuginput

    opd4plane planeactive? 0= if
        win4 5 spaces ." --------" win6     \ <= "dead air" response
        0 to opd4plane
        false exit
    then

    getal 0 6 within 0= if
        win4 5 spaces ." SAY AGAIN?" win6
        false exit
    then

    opdracht 1 =
    getal 0= and
    opd4plane stat@ 29 = and if
        win4 5 spaces ." UNABLE - HOLDING" win6
        false exit
    then

    opdracht 1 =
    getal 0= and
    opd4plane planemoving? 0= and if        \ no landing if waiting to start
        win4 5 spaces ." WAITING TO START" win6
        false exit
    then

    opdracht 1 =
    getal 0<> and
    opd4plane stat@ 40 = and if             \ cleared for landing
        win4 5 spaces ." UNABLE - LANDING" win6 \ dead-air is also appropriate
        false exit
    then

\   opd4plane stat@ 2 5 between
    opd4plane planemoving? 0=               \ if waiting
    opdracht 1 <> and if                    \ no other task than altitude
        win4 5 spaces ." UNABLE - WAITING" win6
        false exit
    then

    win4 space ." ROGER!" win6           \ ROGER mogelijk later printen
    true
    ;
: doopdracht ( -- )
    opdracht 1 = if                         \ 1=altitude
        getal opd4plane newhght!

        opd4plane stat@ 2 = if
```

```
            opd4plane prop@ 0=
            prop_phase      0= and if
               4 planestat!                \ if prop=0 and prop_phase=0 -> status=4
            else
                3 planestat!             \ otherwise status=3
            then
            opd4plane (makenewxy)           \ fill new x&y
        then

        getal 0= if                         \ move status to landing
            40 planestat! then

        exit
    then

    opdracht 2 = if
        getal 1 5 within if
            getal 9 + planestat! then
        getal 0 = if
            30 planestat! then              \ hold at nav-beacon
        getal 5 = if
            20 planestat! then              \ at nav-beacon turn to airfield #
        exit
    then

    opdracht 3 = if
        getal 1 5 within if
            getal 13 + planestat! then
        getal 0 = if
            1 planestat! then               \ proceed
        getal 5 = if
            21 planestat! then              \ at nav-beacon turn to airfield %
        exit
    then

    opdracht 4 = if
        30 planestat! then                  \ hold=start circling at nav-beacon

    opdracht 5 7 within if
        1 planestat! then                   \ maintain and proceed

\   opdracht 7 = if .status then            \ status

    opdracht 8 = if
        20 planestat! then                  \ proceed to airfield #

    opdracht 9 = if
        21 planestat! then                  \ proceed to airfield %
    ;
: entrymodule ( -- )                        \ { catches keystrokes
    key? if
        key >caps
        input_handler
        is= 3 = if
            checkopdracht if
                doopdracht
            then
            win4cr
            0 to is=
        then
    then
    ;                                       \ }
: gameloop
    gameinit
    startmodule if exit then
    begin
```

```
        userinput
        0 to prop_phase
        uartclear cr
        #steps 0 do
            i ." step: " . cr
            i to presentstep
            prepplanes                      \ activate plane & update destinations
            .upcommingplanes
            .activeplanes

            checkplanes 0= if
                ( false unloop exit ) then  \ debug
            30 0 do                         \ 30 microsteps of ~500ms
                i to microstep
                .planesfield
                resettimer begin
                    entrymodule
                    gettimer 10 >
                until
                switchscreens
            loop
            updateplanes                    \ destination to source
            updatestateengine               \ change directions, states, heights etc.
            switchpropphase
        loop
    endmodule                               \ f/t
    until                                   \ exit on
    backtonormal
    ;

\ end game **********************************
\ ******************************************

unused -      . ." bytes"
word# swap -  . ." definitions"

: .pl                                       \ #debug - shows planes
    #plane 1 do
        cr 8 spaces i .planel
        2 spaces
        i stat@ .
        i pstart@ .
        i xnew@ .
        i ynew@ .
    loop ;

: btn backtonormal ;
```

– end of text –