J.J. Hoekstra

sub-version: .c

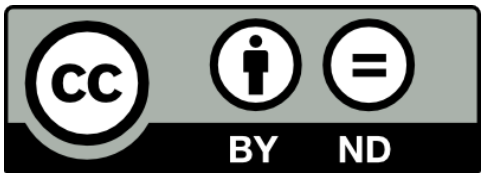
wabiPi4 programmers-guide

for wabiPi4 v3.5.2

WARNING

a Raspberry running wabiPi4
must be cooled adequately

Copyright and license



Foreword

In 1983 I got my first home-computer, an Oric-1, and I was blown away by it. No matter that it was slow, had limited memory, and was riddled with bugs. What fun I had!
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, baddest, 'bestest' computer in the world, by far, was the CRAY-1. Needless to say that at times I fantasised about owning a computer with the unbridled power of a CRAY-1. That fantasy later expanded: the build-in programming language had to be Forth.

In 2016 I started a project trying to create a system as enjoyable as my 80s home-computers - with a decent build-in Forth, and with a decent performance. Ideally with a performance as close as possible to a CRAY-1. WabiPi4 is the main result of this project.

The performance of wabiPi4 is wild! Using the 'Sieve' benchmark from BYTE magazine, wabiPi4 is over 120 times faster than a CRAY-1. As only around 90 CRAY-1's where ever build, wabiPi4 is more powerful than the combined power of all CRAY-1's ever build...

Dream fulfilled? I should think so!

Home computing in the early 80's focused on DIY programming. Graphics, sound-generation and limited interfacing were important building blocks of the fun. The best home-computers also had a build-in assembler. WabiPi4 stays true to this original focus.

WabiPi4 also follows another 80's tradition: like any home-computer from that time, it is 'blessed' with quirks and faults. For instance: most modern technologies are not supported. Things like USB-ports, bluetooth or wired/wireless networking.
And do not start me about this programmers-guide. It is incomplete, unstructured and an eternal work in progress.

It is unlikely that any of this will change a lot in the future. WabiPi4's raison d'être is the enjoyment of programming and playing around, like with the early home-computers. But with more memory and very, very much faster. 'Vintage computing on a lot of steroids'.

WabiPi4 is what it is. Enjoy it and have fun, or move on to other, better systems. (like noForth-T, CiForth, iForth, eForth or Mecrisp)

Content

Forth novices

As of April 2025 wabiPi4 contains 3053 definitions (a.k.a. words). Not a huge number compared to some other Forth systems, but daunting if you're a beginner.
Fortunately most of the definitions are irrelevant when learning Forth. They are related to topics like home-computer functionality, IO, the sound-generation system and the graphical system. And the ARM-assembler accounts for some 600 definitions.
Once you have learned a bit of Forth, there is a lot to explore within wabiPi4.

Enter **.CATEGORIES** and you will get an overview of the different categories in the dictionary:

```
====
 306 constants
 114 values
  56 variables
   8 2constants
   3 2values
   7 2variables
 499 inlinable primitives
 174 primitives
1333 words
 553 created with does>
   0 deferred
====
3053 definitions in total
```

This manual presumes knowledge of Forth. There are excellent introductory books available if Forth is new to you. The classic books "**Starting Forth**" and "**Thinking Forth**" by Leo Brodie are a good, if somewhat dated, starting point. And Elisabeth Rather, the second Forth-programmer in history, has co-authored an excellent book: "**Forth Programmer's Handbook**".

And don't forget membership of a local Forth Interest Group. It is probably the best way of starting with Forth.

WabiPi4 – the focus is on fun

In the introduction I already mentioned that with wabiForth and wabiPi4 I try to realise a dream from the early home-computer era.

This project was started in a haphazard way and with only the haziest of direction or knowledge. The development-environment, ARM assembly, the Raspberry Pi and the GNU assembler were all new to me, and it took a while before I started to grasp some of the technical fineries.

I have always been reasonably up-to-date with my IT. My present laptop, phone, pad and smart-watch all have amazing features and are integrated with each other. A real marvel of technology. What I miss though is the shear pleasure I had with my first computers.

Once I understood that trying to recreate that enjoyment was an excellent goal for my project, the direction became clear. And I produced the following 'essentials' list:

MY PERSONAL LIST OF SUBJECTIVE ESSENTIALS

A build-in programming language – Nothing beats 'switch on and hammer away'. That is the only real 'home-computer' experience. It is clear that this means that an update to the language is more complex. But for me that is actually an advantage. Getting used to one set of bugs is better than having to cope with unknown bugs after numerous updates.

Forth as programming language – I absolutely wanted Forth as the build-in programming language. I love the quirkiness and madness of this language.
Forth transforms every programming-problem into a puzzle. Solve the puzzle and you understand the problem.
An additional goal: the included Forth should support the home-computer related functionality with specific build-in commands.

Plenty of memory – my original specification was to have at least 1 but preferably 8 MB of RAM. Once the project was based on the Raspberry Pi, memory-space became a moot point. The first Raspberry Pi I used, the 1B+, had 512 MB of RAM, 64 times more than my ideal amount. And the present Raspberry I use, the rPi4, has 4 GB of RAM.
However, a lot of memory is only useful if you can actually use it. This sounds obvious, but some Forth-systems limit themselves in this respect. My Forth should be able to handle all memory.

A build-in assembler – I always was a bit jealous of the owners of the venerable BBC-computer. They could use assembly within their BASIC-programs whenever they wanted. No loading of compilers, no linking of sub-programs. Just use the opcodes when you need them. I wanted that.

Speed – I like to dabble with simulations and artificial life and evolution and such like topics. And these topics benefit greatly from a fast computer.
I use the BYTE Sieve benchmark from 1982 to compare the performance of computers. My first home-computer, the Oric-1,

ran that Sieve 10 times in 5500 seconds. That was too slow for any decent simulation. So at the time I developed a simple basic-compiler for the Oric, and programs compiled with it ran around a 100 times faster. And that speed was enough to run decent small simulations.
Thus that was what I was aiming for with my own system: a Sieve time of 55 seconds. Or higher of course. A CRAY-1 'did' the Sieve ten times in only 0.11 seconds. Coming close to that would be very nice.

My '*need for speed*' was the reason why from the start wabiForth compiles inlined native code.

A good sound-generator - my Oric-1 had a 3 channel AY-3-8912 sound-chip, and playing with this was a lot of fun. Creating little tunes or adding sound to games was half the fun. The other half of the fun was just playing around with the different components of the sound-generator: like the ADSR, the noise-generator, the different wave-forms. My dream-computer had to have sound, good sound.

A user-accessible graphics screen - I liked peeking and poking on the screen of my Oric-1. Whether for making my own games, or experimenting with graphics, or creating colourful simulations. What you need for this is a screen with a straightforward memory-map and some colour-capabilities. User-definable characters and sprites, with collision-detection, are nice to have.

Interfacing - Connecting stuff to a computer is almost a must. LEDs, motors, sensors etc. Interfacing can be done with individual in/output lines, but as far as I am concerned also with the I2C, I2S or SPI protocols. The I2C-protocol was published in 1982, the SPI-protocol was first mentioned by Motorola in 1983 and the I2S protocol was introduced by Philips in 1986. So these fit nicely with the spirit of what I wanted to do. And they make interfacing faster and easier.

Access to the CPU - I loved experimenting with the hardware of the Oric-1 and Sinclair QL. Getting snippets of assembly to run, or messing around with the different registers and memory-locations. Clearly my own system had to be as open as possible for experimentation. No access-control and no off-limit sections please.

Reliable storage - the main problem with both the Oric-1 and Sinclair QL was the unreliable storage of programs. My own computer should be able to store and load programs without worries. A lot of storage is not essential, but long-term reliability is.

String handling - I like playing with text and strings. And I like the way Microsoft-BASIC handles strings, with functions like **mid$**, **right$** and **left$.** What I do not like is a length-limit of 255 characters. So a must-have was BASIC-like string-handling, without arbitrary limits on the length of the string.

Floating-point arithmetic - Floating point calculations are often essential for the simulations I program. Integers or fixed point floats lack the required dynamic range. The standard IEEE 32bit floating point numbers are good enough, as long as they are fast.

Other wishes - A fast and reliable random-number generator and an easy way to accurately measure time-intervals were two minor wishes.

THINGS *NOT* ESSENTIAL FOR MY SYSTEM

Internet connectivity, USB support, networking, WiFi, 3D graphics, scalable fonts, etc. All obviously essential in modern computers, but not for what I do with computers for fun.

WHY THE NAME WABIFORTH AND WABIPI4?

The name wabiPi4 comes from the Japanse notion/concept: '**wabi-sabi**'.

Translating these two words is easy:

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

Understanding the deeper meaning, and the role, of these two words in Japanese Culture is complex and probably impossible.
If attempted, an explanation could start with something like: 'You can only be happy if you accept that nothing is ever perfect or will last forever'.
The second phase is: 'if nothing can be perfect, then at least let's make sure that the imperfect is as beautiful and enjoyable as possible'.

Apply this to wabiPi4 and you get something like this:

wabiPi4 is unfinished and imperfect

but I did my utmost to make it really enjoyable.

Cooling the wabiPi4 system

The wabiPi4 system is based on a Raspberry 4b with 4 or 8 GB. During the initial phase of the project, it became clear that a Raspberry Pi was the easiest way of getting a fast processor connected to a large memory.
Most of the other hardware on the Raspberry (USB, networking, Bluetooth etc.) is irrelevant for wabiPi4.

Minimum wabiPi4 system:

1. Raspberry 4b+ with 4 or 8 GB and a good cooling solution
2. Power supply unit 5v, 2.5A or better
3. serial to USB conversion cable
4. terminal program on a PC (OSX, Windows, Linux)
5. SD-card with wabiPi4

A good power-supply is critical for a Raspberry to run reliably. Overall power is important, but the stability of the delivered voltage is even more important. A short dip below 4.6 volt will result in a reset or crash. If this happens you could, if you dare, try to raise the voltage of the power-supply. Aim for 5.20-5.25 volt. Or get a better power-supply...

Power-consumption of the pi4b

The actual power-consumption of a pi4b running wabiPi4 is between 4.5-5.7 Watt (measured on the 220v side). A high graphical workload or complex sound-generation add a couple of Watt. Not a lot, but enough to make good cooling essential!

STEP 1: ADD COOLING - COOLING IS ESSENTIAL

A Raspberry running Linux does not need CPU-cooling, as Linux lowers the CPU-frequency if it gets too hot. In addition, Linux enables only those parts of the CPU which are in active use.

WabiPi4 does the opposite

WabiPi4 overclocks the CPU, works 3 of the 4 cores pretty hard, activates almost all of the CPU, and at all times, even when waiting for user-input, generates high memory-traffic. And, to add insult to injury, wabiPi4 does not check the temperature of the CPU and will happily let the CPU fry itself. And so:

A Raspberry running wabiPi4 must be cooled!

Add a large cooling-element to it. Ideal are the cooling cases of Joy-It. Or a heat-pipe based cooling. Large (25x25mm or 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

Connect the Raspberry via a serial-to-USB cable to a PC. On the PC a terminal program is used 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. There is no need for line- or character-delays. Pins 8 and 10 are the serial I/O. See the chapter on GPIOs further-on.

Connecting a monitor

A monitor can be connected to the Raspberry with a HDMI cable. The Raspberry Pi4b has two HDMI ports, wabiPi4 uses the left port.

Unfortunately the HDMI connector on the Raspberry 4B is a micro-HDMI version. This port is mechanically rather fragile, take care!

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

STEP 3: LOAD WABIPI4 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

Included in wabiPi4 are some primitive I2C drivers for a 20x4 character LCD-screen, 384 kB eeprom and a RTC. And there is a I2S driver for sound. Obviously the user can develop other drivers, for I2C or SPI or anything else.

Project Forth Works on GitHub has some interesting examples for specific I2C and SPI drivers.

Operating-system

WabiPi4 is a bare-metal implementation of Forth. There is no operating system running in the background of WabiPi4.

No Linux, no Android, no RTOS, no nuthink ...

After completion of the boot-process, the following tasks are running:

1. on **core0**: wabiPi4 forth system
2. on **core1**: the graphics-system PIXEL
3. on **core2**: the keyboard input and buffer system
4. also on **core2**: the sound-system WAVE

As **core1** and **core2** handle the background tasks, wabiPi4 on **core0** is free from interrupts, and is available all of the time. This is optimal for performance and helps with exact time-measurements.
The only interrupt possible for **core0** is a user-abort. This is when the user wants to interrupt a running program, for instance because the program is stuck in a loop, by pressing 'FnCtrlBackspace'. **Core2** checks for this combination and if found will interrupt **core0**.

Any other functionality normally provided by an operating system is either handled by wabiPi4, has to be provided and coded by the user, or is not possible.

ACCESS RIGHTS

The 'h' in 'software development' stands for 'happiness'...

Modern CPUs are build around the concept of limited user-access. It is much easier to severely limit access than to grand full access to a user. However, an important goal during the development of wabiPi4 was full user-access to the CPU, in the case of the Raspberry Pi4 the ARM Cortex-a72. In the 80s, experimenting with the processor of a home-computer was an exciting activity for many users, it certainly was for me. WabiPi4 *had* to support experimentation with the Cortex-a72.

In practise this means that the wabiPi4 does not control, manage or limit access-rights to the CPU in any way. On the contrary, the system runs in (secure) supervisor-mode with invasive debug-mode switched on. This gives the user the most complete access possible. In addition, every available setting granting extra access to the CPU is enabled.

The only part of the system not accessible are the registers which are only available in unsecure modes, specifically the setup-registers for HYP-mode. This limitation could have been resolved by making the state switchable between secure and unsecured. But this mandated either a complex memory management-system or lower performance. And reality is that the use of HYP-mode setup-registers is rare in the Forth world.

All other registers are accessible without limits. You can simply write to a memory-mapped configuration-register of the CPU. Put the new value and address on stack and use '!'. The system might crash, but you *can* try. In addition all of the configuration opcodes are available in the assembler.

As an aside: the **HYPmode** (or hypermode) is relevant for WabiPi4. One example is setting the **CNTVOFF** register during the boot process. This **CNTVOFF** register contains an offset from the physical counter for the virtual system counter, and is only accessible in (unsecure) HYP-mode.
The other is **self-hosted DEBUG.** Self-hosted DEBUG-mode is enabled in WabiPi4 at the highest debug-level (ie. invasive DEBUG-mode). This is the only way to give the user in supervisor mode access to the CPU cycle-counter. Enabling self-hosted DEBUG for supervisor-mode can only be done in HYPmode.

There is no dedicated support in WabiPi4 for self-hosted debugging. But if you want to have a go at developing such support, please go ahead. As far as I know, all the registers related to debugging are available to the programmer.

As the system runs in 'supervisor' mode, it is, by the way, perfectly possible to program a Hyper-mode management system with wabiPi4. 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. I might add this functionality in future, but this is not guaranteed nor in the planning.

Starting, aborting and exiting wabiPi4

Starting wabiPi4
Wabi starts by switching the power **'on'.** If the start of WabiPi4 is successful you will be greeted by a boot-screen:

```
 ------------------------------------------------
   WABIPI4 EXTENDED FORTH v3.5.2
   (C) 2025 J.J. HOEKSTRA

   4'035'293'452 BYTES FREE -- CPUf 1800.01 MHz

   Ready
 ------------------------------------------------

 sieve (assembly):    663 us  (pass: < 700) ->  11.7 * IBM-3033 assm. 1899
 sieve (forth):       885 us  (pass: < 900) -> 124.2 * CRAY-1 Fortran 1899
 core1 MMU-test:     1111 us  (pass: < 1150)
 core2 MMU-test:     1111 us  (pass: < 1150)
 uptime wabiPi4:    0.168 s   (pass: > 0.000s)
```

The bootscreen gives information on the system and the results of 5 system-sanity checks.

The first part is inspired by the start-screen of my first computer, the 'ORIC-1' from 1983. The wabiPi4-variant shows the version number, a copyright notice, the amount of free memory and a measured CPU-frequency.

The second part shows the results of 5 sanity-checks. If all 5 checks pass, there is a good chance that the system is running as intended. Mind you, no guarantees. But for these 5 checks to run successfully, a lot of steps in the setup-process have to be executed correctly.

Test 1 (sieve in assembly) - shows that **core0** is running as intended. There is also a bit of totally unnecessary bragging, as it shows that the sieve benchmark in assembly runs more than 11 times faster than the IBM-3033 (nicknamed 'the big one') in assembly (and IBM was cheating...). Finally the number 1899 proves that the sieve actually found the right amount of integers.
Test 2 (sieve in forth) - shows that wabiPi4 is compiling, optimising and executing correctly. And notice that the sieve benchmark using wabiForth runs ~120 times faster than a CRAY-1 using optimised Fortran (the CRAY people were not cheating).
Tests 3 & 4 (core1 and core2 MMU-test) - show that cores 1 and 2 are configured and running correctly.
Test 5 (uptime) - shows that the clock-system is running. You might notice that the first time wabiForth starts, an uptime of around 0.168s is reported, although starting takes around 5 seconds. This is because starting the Raspberry Pi4 hardware takes 4.8 seconds. Much slower, by the way, than a Raspberry Pi3, which starts within 1 second. (The rPi4 uses a slow EEPROM for the initial boot. And because it is an EEPROM, the rPi4 can now also be bricked, in contrast to all previous Raspberry models. But alas, such is progress...)

Reset of wabiPi4

COLD restarts wabiPi4 without resetting the CPU. Most system-settings are reset and any previous program deleted. Memory is not cleared by **COLD.** Using **DUMP** you can see remnants of previous programs after executing **COLD.**
The settings for the compiler and the screen are not reset. This allows the programmer to experiment with the effect of the compiler settings on the Sieve benchmark.

A full reset of the whole system is done with RELOAD. RELOAD resets every part of the system in accordance with the architectural specifications of ARM and Broadcom. After the reset wabiPi4 is loaded afresh from the SD-card. Any program and any data in memory is lost and all WABI-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 not a full system-reset, but it does usually function.
You can overwrite the vector at address 0x0 if you want to. A rather theoretical example where this might be useful is experimenting with ARM64 code. A reset is needed to switch to ARM64-mode, and this address points to the code to be executed after the reset. For more info see **CPU_RMR64.** This is an untested feature.

User-abort -> how to exit 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 wabiPi4 program and return wabiPi4 to where commands can be typed in.
The memory is not cleared by this abort and the dictionary stays intact. 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 something is seriously wrong with wabiPi4. Or you are not using an Apple OSX computer, which is also not good. In those case only a hard reset, by switching the system 'off' and 'on' (AKA power-cycling), will help. Any programs and data in the system will be lost and memory will be filled with unknown data.

The 'debug-over-reboot' feature of the ARM-CPU is active, as wabiForth is running in 'invasive self_debug_mode'. This sometimes makes it necessary to switch off for 10-20 second or so to really get a reset, just like old home-computers. Remember the 80's saying: "This is not a bug, this is a feature!"? Yeah, this is like that...

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 then aborts the program.

Exit of wabiPi4
is done by switching the Raspberry ‘off’. The classic word **'BYE'** is available (and adorable) but has a peculiar will of its own and thus its use is discouraged.

COLD (--) restarts wabiPi4, resets most of the system-variables and reloads the tools-dictionary. The compiler optimisation-settings and screen-settings are not reset.

RELOAD (--) restarts the Raspberry Pi, reloads the firmware of the Raspberry and reloads wabiPi4. (synonym: **REBOOT**)

ABORT (--) immediately exits a running program and puts control back with the user. It does not clear memory, so the user can inspect items like variables etc. If an abort happens during compilation of a word, **HERE** is put back to where it was before the compilation of the word started. Thus avoiding memory leaks.

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 functions inside a definition.

BYE (--) has no useful functionality. Where would you go? What is there outside of WabiPi4? Not even a bare emptiness...

Limits of the wabiPi4 system

WabiPi4 is a 32 bit system. All registers, the cells, the address- and data-busses are all 32 bit. WabiPi4 has access to 4 GB of memory in the system, also on a pi4 with 8 or 16 GB memory. In wabiPi4, the memory of a 4 GB Pi4 is split into a lower 1 GB part and an upper 3 GB part. The dictionary of wabiForth is located in the lower part, the upper part is available for data.

maximum size dictionary	983 MB
maximum size allotted memory block within the dictionary	983 MB
maximum size allocated memory block in data-part of memory	2863 MB
maximum size string	strings have a 32 bit index - depending on how they are defined, the maximum size is 983 or 2863 MB.
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. Utterly irrelevant in normal coding, but it could be relevant for automatically generated code.
maximum number of definitions in the dictionary	Limited by the size of the hash-table. Between 4753485 (worst case) and 5592373 (best case) definitions. Likely enough for most applications.
maximum number of vocabularies in the dictionary	Up to 255 word-lists can be defined in the system. Up to 8 can be put in the search-order-list at the same time.
depth of stacks	There are four stacks in wabiPi4: data, return, cpu-return and floats. Each stack is 2048 cells deep. No checks are done on under or over-runs.

CPU-specific wabiPi4 limitations:

The words **2@** (two-fetch) and **2!** (two-store) take an address as argument. Contrary to wabiForth for the rPi3b+, for wabiPi4 these addresses do not have to be 4-byte aligned.

ARM32 related limitations
Some opcodes in the ARMv8 architecture 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.

Dictionary

The wabiPi4 dictionary is a linked list of definitions. Listing words in the dictionary can be done using the ANSI standard **WORDS** or 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 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 first character specified after **AWORDS.** For instance: typing '**AWORDS d**' will show all words starting with 'D' or 'd'. **AWORDS** prints all non-hidden words, irrespective of word-lists, and is case-insensitive.

WORDSL (--): prints out a list of all words in the dictionary including the execution-token and the inline-length. A word can only be inlined if the inline-length is larger than zero, and inlining is switched 'on'. In addition the inline-length of the word must not be larger than the length contained in the value **MAXINLINE.** The default behaviour is that all primitives with an inline-length larger than 0 are inlined.
As a side note: the default maximum inline-length on the Raspberry 3b+ is set by to 8 opcodes, as inlining with more than 8 opcodes never gives any speed-advantage with 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. This word is also a good test to check if the integrity of the dictionary is still intact, as it executes a plausibility check on all links in the dictionary.

#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 then onwards will print the last 10 words.

THE HEADER

Each definition in the dictionary has a header which contains the link to the next word, the name of the definition and essential meta-data. The header is key to correct functioning of the dictionary.

The header has a fixed length of 48 bytes. The name of the definition can contain a maximum of 32 characters. The first 4 bytes of each header are presently available to the user. Byte 4 and 5 are reserved for future use.

header of a wabiPi4 definition

byte	content of byte/word/string
0	free
1	free
2	free
3	free
4	reserved for future use
5	reserved for future use
6	object_id: 1=value, 2=variable, 3=constant, 4=2variable, 5=2constant, 6=deferred, 7=primitive, 8=created w/ DOES>, 9=wabiForth def, 10=inlinable primitive, 11=2value
7	hidden flag: 0=print, 1=do not print
8	link word - points to XT of next word
:	...
:	...
:	...
12	inlining length: 0=never inline - 1 or higher: number of opcodes to be inlined if inlining is active
13	compiling=-1 immediate=0
14	WID (wordlist identifier) - 0=FORTH
15	length name - max=32 characters
16	1st char name
:	2nd char name
:	...
:	...
47	...
48	1st opcode - address = XT
:	

DO...LOOP & FOR...NEXT

DO...LOOP

WabiPi4 follows the ANSI standard: **DO**, **LOOP**, **?DO**, **+LOOP**, the indexes **'I'** and **'J'** and **LEAVE**, **UNLOOP** and **EXIT** are available. In addition the non-ANSI word **'K'** is available.

WabiPi4 specifics

WabiPi4 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 can be helpful in finding bugs early.

Limits of DO...LOOP in wabiPi4

The distance of the jump back from **LOOP** to **DO** cannot exceed 32MB. A non-issue for normal programming, but it might 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 wabiPi4 system. Two CPU-registers are dedicated permanently to the **DO...LOOP.** This in itself makes looping a lot faster. And it also allows for additional optimisation using inlining and compounding.

The following example compares moving a memory-block from source to a destination with the **DO...LOOP** and with **MOVE. MOVE** uses the fastest method (according to the ARM-corporation) in assembly. Obviously **MOVE** is faster, but the **DO...LOOP** doesn't do too badly.

```
decimal
100000000 allocate drop constant tstsource
100000000 allocate drop constant tstdest

: fillsource 25000000 0 do     \ fill with random numbers
     rndm32 tstsource i 4 * + !
  loop ;

: tst1  25000000 0 do
     tstsource i 4 * + @       \ get a word
     tstdest   i 4 * + !       \ store a word
  loop ;

: tst2 tstsource tstdest 100000000 move ;

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

The maximum bandwidth of the memory-system of the Raspberry 4b is around 5 GB/s, and the GPU consumes ~1.3 GB/s of that. So **MOVE** comes close to the maximum.

method	typical execution times (microseconds)	relative to MOVE	bandwidth (MB/s)
DO...LOOP	~134359	253%	~1488
MOVE	~52973	100%	~3775

The **DO...LOOP** is 2.5 times slower than **MOVE.** Pretty impressive considering that **MOVE** uses the fastest possible method in assembly.

Some specific reasons why the **DO...LOOP** is slower:
1. more opcodes are used for the move of the data
2. the forth-words between **DO** and **LOOP** manipulate the stack which consumes memory-bandwidth in itself.

What if the GPU is not active?
In that situation the whole of the memory-bandwidth is available for the move of data, and both routines are faster:

method	typical execution times (microseconds)	relative to MOVE	bandwidth (MB/s)
DO...LOOP	~88956	258%	2248
MOVE	~38487	100%	5196

Memory-alignment
MOVE, **@** and **!** function with and without memory-alignment. But whereas the performance of the **DO...LOOP** is largely independent of alignment, the **MOVE** routine can slow down somewhat when data is moved from or to an unaligned address.

Please note that using:

```
create tstdest   100000000 allot
create tstsource 100000000 allot
```

to define the two memory-blocks would result in slower execution of the code.

There are two reasons for this:
1. this creates non-inlinable, and thus slower, words to put an address on stack. Normally not noticeable, but clearly so when you do 25 million loops with a **DO...LOOP**, where these two words each put a value on stack every loop.
2. **ALLOT** reserves space in the dictionary-area. The memory-attributes (especially related to caching of data) for this area are for mixed use, ie executables and data. The consequence is slower performance compared to data contained in memory with the data-only attribute. (see **'ALLOCATE'**)

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** construct is faster than the **DO...LOOP**, uses less cells on the return-stack and is smaller and easier to implement. All note-worthy advantages.

Reality-check...

On larger CPUs, like the Cortex-a72 in the Raspberry 4b, testing shows that there is no speed-advantage. And saving a few cells on the return-stack or a few words of memory is irrelevant if you have GigaBytes of RAM available.

FOR...NEXT - choices to be made

As there is no speed-advantage to be gained, wabiPi4 provides the **FOR...NEXT** loop mainly for compatibility-reasons. Preferably with as many different Forth-implementations as is possible.

Being 'compatible' is not an easy task; **FOR...NEXT** is not part of the ANSI-standard. The ANSI committee found that standardising **FOR...NEXT** was impossible, 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.

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 **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 eluded the ANSI-committee.

The wabiPi4 FOR...NEXT implementation

WabiPi4 supports both main variants of **FOR...NEXT: FOR_pre** and **FOR_post.**

The following features are available in wabiPi4 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 wabiPi4. 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_post** variant is, for instance, compatible with eForth and iForth and many other implementations. The **FOR_pre** variant is compatible with noForth-T.

Please note: UNLOOP and LEAVE from **FOR...NEXT** are not always available in other Forth-implementations; do not use them if you want to be as compatible as possible 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 wabiPi4 it is included for compatibility reasons. This word is one of those concepts which either grows slowly on you, or not at all.
It *is* a powerful word. You will find several interesting examples of the word in the book: "eForth Overview" by dr. C. H. Ting. The essence of the word is that it is a structured and more powerful alternative for the **?DO...LOOP.**

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 wabiPi4, **AFT/THEN** can also be used in the **DO...LOOP.**

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 wabiPi4 a large input-value is needed. A value of a hundred million gives a time-out time of around 5 seconds. This large number is very energy-inefficient, as the CPU is working very hard while waiting. It also looks silly. In the **assembly-examples** chapter there is an example showing how, by mixing in one opcode with the assembler, a more energy 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 ;
```

Stack manipulation

The ANSI standard defines 26 words concerning stack manipulation:

single	double	floating point	double floating point
DUP	2DUP	FDUP	
DROP	2DROP	FDROP	
OVER	2OVER	FOVER	
SWAP	2SWAP	FSWAP	
ROT	2ROT	FROT	
R>	2R>		
R@	2R@		
>R	2>R		
?DUP			
NIP			
PICK			
ROLL			
TUCK			

In wabiPi4 these words function as described in the standard.

From the icy blue empty cells in the table above, it is clear that quite some functionality is left out of the ANSI standard. For instance, there is no standard way of pushing a float on the return-stack, there is no **2NIP**, etc.
At the time of the development of the ANSI-standard, there was a movement to limit the number of words in de standard. Too many words was seen as too difficult to remember for Forth-programmers. And so words deemed less important were left out. The unfortunate and Kafkaesque result is that a programmer now has to remember which words do *not* exist...

Fine if you are a Forth-purist, *glory be!* Less so if you use floats in a time-critical manner. For instance in a neural network, or a nice fat simulation.
Obviously it is possible to create the missing words from the ones available. But these high-level definitions slow the execution down.

WabiPi4 adds most missing words and then some. They are all implemented as optimised primitives, ensuring optimal performance.

Use of any one of these words renders a program non-standard, unless you provide an alternative high-level implementation of the offending word. But inclusion makes your program run faster. Sometimes hardly measurable, sometimes clearly noticeable.

3DUP (n o p -- n o p n o p): copies the top 3 values on stack

4DUP (n o p q -- n o p q n o p q): copies the top 4 values on stack

1DROP (n o -- n): exactly the same function as **DROP**. But has the same opcode structure as **2DROP**, **3DROP** etc. This makes **1DROP** useful in timing measurements.

-ROT (n o p -- p n o): moves the top value to the 3rd position of the stack and moves the other two values up one position.

RDROP (--) r:(n --): drops value from return-stack

?2DUP (dn -- double_zero / dn dn): conditional duplicate of double. If dn is unequal to zero -> duplicate the dn.

2NIP (dn dm -- dm): remove dn from stack.

2PICK (dn*x n -- dn*x dn): pick the nth double value from stack and put it on top.

2ROLL (du du-1...d0 u -- du-1...d0 du): Like the ANSI word **ROLL**, but for double values. U is a single. Programmed in hi-level Forth.

2-ROT (dn do dp -- dp dn do): moves the top double to the 3rd double position of the stack and moves the other two double values up one double position.

2TUCK (dn do -- do dn do): copies double on top of stack to below 2nd double

2RDROP (--) r:(d --): drops a double value from the return-stack. Equal to '**RDROP RDROP**'.

For the words related to the floating point stack, see the chapter on that.

Stacks (data, user-return, CPU-return, FP)

WabiPi4 is has 4 stacks relevant to the system. The classic data-stack, the user-return stack, the CPU return-stack and the floating-point stack.

The only non-standard part is the two return-stacks. The CPU return-stack forms the basis for the branch-predictor. Mixing user-data and CPU-data on one return-stack would make the predictor a lot less efficient. The branch predictor of the CPU makes its predictions solely based on the data on the CPU return-stack and the data in r13 and r14. Any time software pushes or removes data from this return-stack for its own use, the branch-predictor would react by resetting itself. This to ensure integrity of the system.

The branch predictor is important for the performance of the CPU. A predicted branch and return together take 3 cycles. An unpredicted branch and return take a minimum of 12 cycles, or more if the data is not available directly. This is especially detrimental for a software-environment with lots of small subroutines. Like a Forth system...

Analysis showed that separating the user and CPU data onto two separate stacks did not have negative consequences. All the relevant words for handling the return-stack stay the same. The advantages are relevant: the CPU runs optimally, and the system is overall less likely to crash. The decision to split the stacks up was in fact an easy one.

The **ANSI** standard contains 2 definitions 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 used in the ANSI standard. For instance **SP@** and **SP!** are used in the example sources for **CATCH** and **THROW** in the **ANSI** standard, although they are not part of the **ANSI**-standard.

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.

USDEPTH (-- n): puts depth of the user return-stack on stack.

The ANSI standard mentions and uses two words although they are not part of the ANSI standard: **SP@** and **SP!.** WabiPi4 adds a few similar words which give the programmer full control over the stacks.

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

FP! (addr --): puts the address on stack into the float-stack pointer

GETTING DATA FROM AND TO R13, R14 AND R15.

This section is only directed at programmers who like to crash a program often. All these definitions are experimental.

Using the following words, the programmer can read the contents of, and write data into the CPU return-stack pointer (=r13), the link-register (=r14) and the program counter (=r15).

These words are included to give the programmer ultimate control. But especially writing to these registers is very risky as it directly influences the program-flow of a running system. **Playing with this usually leads to fatal crashes,** where a power-cycle is needed. But then again, how would you learn if you do not test?!

R13@ (-- addr): puts the content of CPU register 13 on stack. Register 13 is the CPU return-stack pointer. In the chapter **'Finding Yourself'** R13@ is used in an example.

R13! (addr --): puts the address from the stack in CPU register 13. Register 13 is the CPU return-stack pointer.

[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 link-register of the ARM-processor.

R14@ (-- addr): fetches the address contained in register 14 of the CPU. This word can only be used in interpreting mode and will always return the same value.

R14! (addr --): writes the value on stack to register 14.

INLINING must be on for this word to work correctly

This word changes the address where the program returns to at the end of the presently active subroutine. And so it directly influences program-flow at the end of the active routine.

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

GOTOADDR (addr --): writes the value on stack into register 15, and so directly enforces a program jump to the address on stack. Almost any use of this word leads to a crash or another failure. Certainly not for the faint of heart.

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. On the pi3b+ this word is clearly faster than **MOVE,** thus the name. But on the pi4b, with a more efficient memory and store/load system, this is rarely the case. It is provided to be compatible with wabiForth for the pi3b+.
Contrary to the pi3b+ version, there is no technical need to align the origin and destination addresses for **FASTMOVE.** But using unaligned addresses can be up to 25% slower.

In the **DO...LOOP** chapter there is an example comparing the speed of **FASTMOVE** and **DO...LOOP**

Please note: in wabiPi4 v2.5.3 and earlier, **FASTMOVE** contains a bug which resulted in errors when moving data up. From v2.6.0 onwards this bug has been resolved.

FASTFILL (addr, no_of_words, n --): Fills the memory starting at address with the 32b word n, no_of_words times. **FASTFILL** on the Raspberry pi3b+ was clearly faster than **FILL,** but on the pi4b there is no difference. **FASTFILL** is provided here for compatibility with pi3b+ programs.
Addresses can be non-aligned at little or no loss of speed.

Store, fetch, variables and values

I remain adamant that local variables are not only useless, they are harmful...

Charles Moore, 13apr1999

WabiPi4 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. And naturally the standard words **VALUE, TO** and **+TO** are included. Local variables are not provided (see quote on the left).

DOUBLE VALUES

The ANSI standard is lacking standards for double values, which seems a tad inconsistent. In wabiPi4 double values are available, and **TO** and **+TO** handle both **VALUE** and **2VALUE** correctly.
Overall there is little or no speed advantage in comparison with double variables. **VALUES** and **2VALUE** are used to avoid having to write directly to memory, thus avoiding an important source of bugs, bugs which can be difficult to track.
The word **M+TO** is available to add/subtract single numbers to/ from a double value, it is a bit faster than **+TO.**

2VALUE <name> compiling:(double --) execution:(-- double): like **2VARIABLE**, stores double numbers. **TO** and **+TO** stay the same. They correctly handle both **VALUE** and **2VALUE.**

M+TO <name> (n --): adds a single to a double **VALUE.** Using **M+TO** is faster than **+TO** for a **2VALUE.** So if you want to add a small number to a double, for instance to count something, use **M+TO.** Compared to **M+2!, M+TO** is around 25% faster.

SPECIFIC STORE AND FETCH-WORDS

The following words are included for performance reasons. They can all easily be programmed in Forth, but as modern CPU's usually have dedicated opcodes for this functionality, it makes sense to include them as primitives.

Halfword fetch/store

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

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

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

Signed character fetch

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

Add-stores

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

H+! (n addr --): raises the 16bit value at address with 16bit value n

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

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

Auto-indexing fetches

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 with maximum string-lengths of 256 characters. Using **C@+** is an efficient way of going thru a character array.

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 2.

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

@+ (addr -- addr+4, n): fetches a value from address and raises the address with 4. This is the same as **COUNT** in wabiPi4. Using **@+** is an efficient way to go thru an array.

2@+ (addr -- addr+8, d): fetches a double value from address and raises the address with 8. Using **2@+** is an efficient way to go thru an array of doubles.

Auto-indexing stores

C!+ (c addr -- addr+1): stores character **c** at addr and leaves addr+1 on stack

H!+ (h addr -- addr+2): stores halfword **h** at addr and leaves addr+2 on stack

!+ (n addr -- addr+4): stores value **n** at addr and leaves addr+4 on stack

2!+ (d addr -- addr+8): stores double **d** at addr and addr+4, and leaves addr+8 on stack

Overview of all available variants of fetch/store words:

element	fetch	auto-indexing fetch	store	auto-indexing store	memory addition
char/byte	C@	C@+	C!	C!+	C+!
signed char/byte	SC@	SC@+			
halfword	H@	H@+	H!	H!+	H+!
signed halfword	SH@	SH@+			
word (cell)	@	@+	!	!+	+!
double	2@	2@+	2!	2!+	D+2!
mixed word and double					M+2!

Some other words doing fetch and store

There are six words defined which do a very specific sort of fetch, namely 16, 24 and 32 bit BIG-ENDIAN fetch/store from and to an unaligned address. This can easily be programmed in Forth with **C@**, **C!** and **LSHIFT**. But as inlinable primitives they are 10-20 times faster. I found these words particularly useful with large scale simulations, where speed of execution is very relevant.

16B@ (un-addr -- u): fetches a 16bit unsigned number (ie. without extension of the sign-bit) from an unaligned address in BIG-endian fashion.

24B@ (un-addr -- u): fetches a 24bit unsigned number (ie. without extension of the sign-bit) from an unaligned address in BIG-endian fashion.

32B@ (un-addr -- u): fetches a 32bit number from an unaligned address in BIG-endian fashion.

16B! (16b un-addr --): stores a 16bit number BIG-endian in an unaligned address.

24B! (24b un-addr --): stores a 24bit number BIG-endian in an unaligned address.

32B! (32b un-addr --): stores a 32bit number BIG-endian in an unaligned address.

Comparisons

WabiPi4 contains all comparison operators from the ANSI standard. These are: **0<, 0>, 0=, 0<>, <, =, <>, >, U<, U>, WITHIN, D0<, D0=, D<, D=, DU<, F0< and F0=**. All function as specified in the standard.

The ANSI-standard operators form only a small subset of possible comparisons. For instance, the standard only describes 2 floating-point comparisons. Other floating-point comparison-operators have to be constructed by the programmer. This is at least less efficient than having these available in the dictionary. The more so as modern processors are extremely efficient in this area.
So it is logical that wabiPi4 contains more operators than those contained in the standard.

All comparisons in wabiPi4 are inlinable primitives, and all are optimised for the Cortex-a72 processor in the Raspberry pi4b.

The following **additional comparison-operators** are available:
<=, U<=, >=, U>=, 0<=, 0>=, 2^=, EVEN?, ODD?,

<< CHAPTER TO BE COMPLETED >>

Integer numbers and calculations

WabiPi4 supports all the ANSI definitions for calculations with integer numbers. All calculations are programmed as optimised primitives and, where beneficial, primitives are inlinable.

Floating point numbers and calculations

The float package is under development

The **classic Forth-way** of handling floating point calculations is to avoid them. Just act as if it does not exist. And if fractional numbers are a must, a fixed point number system is preferred. Very smart algorithms have been devised to use fixed point for basically all possible calculations. On older CPUs using fixed point has an additional advantage, it is usually faster than full floating point calculations.
With modern CPUs however, calculations with floating point numbers are as fast as integer number calculations (or even faster). And there are topics, for instance when filtering techniques signals, where integers or fixed point simply do not work well.

WabiPi4 supports 32b IEEE floating point calculations. In addition some elementary 64b floating point calculations (like addition, subtraction, multiplication, division and square root) are available. **The focus for wabiPi4 is speed,** and that is also reflected in the floating point routines in wabiPi4. The algorithms are fast, but ultimate accuracy is certainly not reached. Do not use wabiPi4 floats for your next tax-calculations.

SETUP OF THE FLOATING POINT SYSTEM

WabiPi4 uses a separate stack for the floating point calculations. For this float-stack most of the usual stack-manipulations are available. The stack manipulations for the floating point stack always start with an '**F**'. Examples are: **FDROP**, **FDUP**, **FSWAP**, **FOVER** etc. A complete list can be found further on.

WabiPi4 supports transcendental functions like **sine**, **cosine**, **logarithms** etc. Again here, the fastest possible implementation, and not the most precise implementation is chosen. A neural network developed with these floats will work nicely. An accountancy program will probably not.

FORMAT OF FLOATING POINT NUMBERS

If during interpretation of user-input a sub-string is not recognised as a word in the dictionary or an integer number, wabiPi4 will try to convert the sub-string to a floating point number, provided BASE equals 10. The main rule is that a floating point number always has an 'e' as part of the written description.

The following 6 examples are all valid floating point representations of the value 1:

1.0e 1.e 1e 1e0 0.1e1 10e-1

FLOATING POINT STACK (FPS) MANIPULATION

FDUP f:(a -- a a): duplicates the top value on the floating point stack (fps)

F2DUP f:(a b -- a b a b): duplicates the top 2 values on the floating point stack (fps)

FDROP f:(a --): discards the top value from the fps (floating point stack)

F2DROP f:(a b --): discards the top 2 values from the fps

F3DROP f:(a b c --): discards the top 3 values from the fps

F4DROP f:(a b c d --): discards the top 4 values from the fps

FOVER f:(a b -- a b a): duplicates the 2nd value on the fps

FSWAP f:(a b -- b a): swaps the order of the 2 top values on the fps

FROT f:(a b c -- b c a): moves the 3rd value on the fps to the top

F-ROT f:(a b c -- c a b): moves the top most value on the fps to the 3 rd position.

FPICK (n --) f:(-- nth_value): copies the n-th value from the fps to the top of the fps. This word changes both the data-stack and the floating point stack.

FTUCK f:(a b -- b a b): copies the top value on the fps to the 3rd position

FNIP f:(a b -- b): discards the 2nd value on the fps. Equal to **'FSWAP FDROP'**, but 1.5 cycles faster. Yeehj...

MOVING FLOATS TO AND FROM THE RETURN-STACK

Please note: the user-return stack is used as temporary storage for both the data-stack and the floating point stack. The result is that integer and floating point values can be mixed on the return-stack. It is up to the programmer to ensure that the right value is used at the right time.

F>R f:(a --) r:(-- a): moves the top value on the fps to the return stack for later use

FR> r:(a --) f:(-- a): moves the top value on the return stack to the fps

FR@ r: (a -- a) f: (-- a): copies the top value from the return stack to the fps, the value on the return stack is not dropped.

FLOATING POINT CALCULATIONS

F+ f:(a b -- a+b): adds the 2 top values on the fps and returns the result

F– f:(a b –– a–b): subtracts the 2 top values on the fps and returns the result

F* f:(a b –– a*b): multiplies the 2 top values on the fps and returns the result

F/ f:(a b –– a/b): divides the 2 top values on the fps and returns the result

FSCRT f:(a –– sqrt(a)): takes the square root of the top value and returns the result

F^2 f:(a –– a^2): multiplies the top value with itself and returns the result

FD+ f:(da db –– d(a+b)): adds the 2 top double values on the fps and returns the result as a double

FD– f:(da db –– d(a–b)): subtracts the 2 top double values on the fps and returns the result as a double

FD* f:(da db –– d(a*b)): multiplies the 2 top double values on the fps and returns the result as a double

FD/ f:(da db –– d(a/b)): divides the 2 top double values on the fps and returns the result as a double

FDSQRT f:(da –– sqrt(da))): takes the square of the double value on the fps and returns the result as a double

FD^2 f:(da –– da^2)): multiplies the double value on the fps and with itself and returns the result as a double

fmin
fmax

fabs
fnegate

fdabs
fdnegate

ftan
fcos
fsin

fatan
facos
fasin

fln
flog
fexp
fpowf

fceil
ffloor
fround

```
f>
f<

f= << mist

f0<>
f0>=
f0>
f0<=
f0<
f0=
```

FETCHING AND STORING

F! (addr --) f:(s --): store single floating point value 's' in address 'addr'.

F@ (addr --) f:(-- s): fetch the single floating point value from address 'addr'

FD! (addr --) f:(d --): store the double floating point value 's' in address 'addr'.

FD@ (addr --) f:(-- d): fetch the double floating point value from address 'addr'

MOVING VALUES TO AND FROM FLOATING STACK

>FPS (n --) f:(-- n): moves the 32 bits on the data stack to the floating point stack WITHOUT CONVERSION.

FPS> (-- n) f:(n --): moves the 32 bits on the floating point stack to the datastack WITHOUT CONVERSION.

FPS>D (-- d) f:(d --): moves the double on the float stack to the datastack WITHOUT CONVERSION.

D>FPS (d --) f:(-- d): moves the double on stack to the floating point stack WITHOUT CONVERSION.

FDUP> (-- n) f:(n -- n): pushes a bit-exact copy of the top value on the float stack to the data stack. This word was used a lot during debugging of the float-package.

CONVERSIONS

F>S (-- n) f:(n_fl --): converts the floating point single value n to an integer on the data stack.

S>F (n --) f:(-- n_fl): converts the single value n to the floating point representation on the floating point stack

FD>FS f:(d_fl -- s_fl): converts the double floating point single value n to an double floating point value on the floating point stack.

FS>FD f:(s_fl -- d_fl): converts the single float value to the double floating point value on the floating point stack

FLOATING POINT CONSTANTS

FPI f:(-- pi): puts pi as floating point value on the float stack.

F180/PI f:(-- 180e/pi): puts 180/pi as floating point number on the float stack

FECONST f:(-- e): puts 'e' (Eulers number) on the float stack. Equivalent to **2.718282e**

VARIOUS OTHER FLOAT RELATED DEFINITIONS

FVALUE interpretation: f:(a --) execution: f:(-- a): using FVALUE the programmer can create floating point values which return the value contained on the float stack. It functions exactly the same as VALUE for integers. Presently programmed as a high-level word.

example:

```
1.0e fvalue myfloat
```

creates a definition called MYFLOAT which, when called will put the value 1.0e on top of the float stack.

FTO f:(a --): fills a **FVALUE** with the top value on the fps. Presently programmed as a high-level word, and thus slow.

F. f:(a --): (FDOT) prints the top value of the float stack in a float representation with 6 digits precision. PLEASE NOTE: this definition is inspired by a vintage Forth83 example, and is not very precise. It will very often print slightly too low or high numbers. For the intended use however it is accurate enough.

F, f:(a --): (FCOMMA) writes the top value from the float stack in memory at the position specified by **HERE**, and adds 4 to **HERE**.

FRNDM f:(-- 0.0e-1.0e): pushes a random number between 0.0e and 1.0e on the float stack in 12c. Uses the same random generator as **RNDM32** to generate a 32b random number, which is then converted to the a float number in the range of 0.0e-1.0e.

HARDWARE CONTROL OF THE FLOATING POINT CO-PROCESSOR

The **FPSCR register** controls the floating point co-processor and it contains a detailed overview of the status of the floating-point co-processor. With this register it is for instance possible to change the rounding-method used.

For an overview of the Floating Point Status and Control register see the documentation from ARM.

FPSCR@ (-- content_fpscr): pushes the content of the **floating point status and control register** on the data-stack

FPSCR! (new_content_fpscr --): moves the top-value of the data-stack to the **floating point status and control register** on the data-stack

Example:

```
fpscr@ bit21 or fpscr!
```

changes the rounding method from the default 'round_to_nearest' to 'round_to_plus_infinity'**.**

User Input

User input is via the serial line at **GPIO 14 (=pin 8)** and **15 (=pin 10),** and technically is handled by the WAVE-loop running on core2. This loop polls the UART-hardware 44100 times per second for input. The advantage of this way of handling the input is that **core0** can run interrupt-free.

Received characters are put into a circular buffer with a size of 2 MB. This buffer can store over 3 minutes of received characters at the actual speed of ~9700 B/s (UART speed 115200), ensuring that the system is reasonably resilient against buffer overruns. You can find definitions to manage the input-buffer below.

Adherence to the ANSI standard:
The entry-system is the basis for the normal Forth user-input words. **KEY, KEY?, EXPECT, SPAN, >IN,** and **TIB** function as described in the ANSI-standard. **ACCEPT** works as described in the standard but has, as additional feature, the handling of 'esc'.

The user can interrupt a running program by pressing 'fn', 'ctrl' and 'backspace' at the same time. If this combination of keys is observed by **core2,** a fast interrupt request is send via de GIC (global interrupt controller) to **core0,** which then aborts the running program and puts the user back in control. This is very handy if for instance a program is stuck in an endless loop.

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 definitions are available

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

The following definitions 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 likely 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

User Output

DOT ROUTINES

Numeric output is via the standard ANSI numeric output words: **.** (dot), **U.** (u-dot), **.R** (dot-R), **U.R** (u-dot-R), and the pictured number output words **<#** (less-number-sign), **#>** (number-sign-greater), **#** (number-sign), **#S** (number-sign-s), **HOLD**, **SIGN**. They all follow the standard. The word **-SIGN** is available for a more usual format.

In addition several words are available which print in common formats. I use these so often that I included them in the permanent dictionary. Saves a bit of source code.

Numeric output is based on the value in **BASE**, unless otherwise specified. The standard ANSI words for handling this are available: **BASE**, **DECIMAL**, **HEX**, **BINARY**. They all follow the ANSI standard. The system aborts as soon as BASE is used by the system, if a value outside the range of 2-36 is contained in BASE.

BASE handling:

BASE can only have a value between 2 and 36 (inclusive). The system checks this when relevant. The following word gets the value of BASE, but aborts if BASE contains an illegal value.

GETBASE (-- base): puts the value contained in the variable BASE on the stack. Aborts if the value is outside the range 2-36 (inclusive) and returns BASE to decimal.

Pictured numeric output

-SIGN (--): used inside **<#** and **>#.** Places a 'minus_sign' when the input number is negative, but does not place a sign when the number is zero or positive. This is the preferred convention for most people.

Variations of DOT words

.BYTE (c --): prints the lower 8 bits of a value in the present **BASE**

.SBYTE (c --): prints a signed byte. That is bit 7 is extended as sign-bit. For instance the value 130 is printed as -126, and the value 255 is printed as -1. Uses present **BASE**.

.BIT (n --): prints a zero if value equals zero, otherwise prints a one.

.FLAG (n --): prints a zero if value equals zero, otherwise prints a minus one.

.BIN (n --): prints in binary, independent of actual present BASE

.8BIN (n --): prints the lower 8 bits of value n in binary, independent of actual present **BASE**, including leading zeroes.

.16BIN (n --): prints the lower 16 bits of value n in binary, independent of actual present **BASE**, including leading zeroes.

.32BIN (n --): prints the 32 bits of value n in binary, independent of actual present **BASE**, including leading zeroes.

.64BIN (d --): prints the 64 bits of double value d in binary, independent of actual present **BASE**, including leading zeroes

.64BIN_ (d --): prints the 64 bits of double value d in binary, including leading zeroes, and without space after the printed number

.HEX (n --): prints in hex, independent of actual present **BASE**

.RHEX (n r --): prints in hex, independent of actual present **BASE**, right aligned with r characters

.32HEX (n --): prints value n in hex, independent of actual present **BASE**, including leading zeroes

.64HEX (n --): prints double value d in hex, independent of actual present **BASE**, including leading zeroes

.64HEX_ (d --): prints the 64 bits of double value d in hex, including leading zeroes, independent of actual present **BASE**, and without space after the printed number

.DEC (n --): prints in decimals, independent of actual present **BASE**

FIXED FLOAT printing

.1DEC (n --): prints value n in decimals, as if divided by 10. For instance the value 123 is printed as 12.3

.2DEC (n --): prints value n in decimals, as if divided by 100. For instance the value 1234 is printed as 12.34

.3DEC (n --): prints value n in decimals, as if divided by 1000. For instance the value 12345 is printed as 12.345

.4DEC (n --): prints value n in decimals, as if divided by 10000. For instance the value 123456 is printed as 12.3456

String handling

String handling is an important area of WabiPi4. So important that there is a system of overflow-protected string variables available, in addition to the ANSI sting-words.

See the next chapter for an overview of overflow protected string variables.

Strings in wabiPi4 have a 32 bit index and thus are not limited to 255 characters. In addition, the handling of strings in wabiPi4 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), C" (c-quote), SEARCH, COMPARE, /STRING (slash-string), -TRAILING, CHAR, [CHAR], CHAR+** and **CHARS** function as specified in the ANSI-standard, with the following exceptions:

- **SLITERAL** also works during interpretation
- the use of **DOUBLE-QUOTES** in the string definition words.

SLITERAL

In wabiPi4 SLITERAL can also be 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 disallows changing this string, however wabiPi4 does not guard against this. Complying with the standard is the responsibility of the user.

An example can be found below.

```
SLITERAL   interpreting: ( addr len ... <ccc> -- )
           exec: ( -- addr len )
```

DOUBLE-QUOTES

The words ." , S" and C" (and ABORT") have an additional feature. In standard ANSI Forth the string contained in these words cannot contain the character '"' (double-quote). This character marks the end of the string, and so the first occurrence of a double-quote after these 4 words ends the string.
In wabiPi4, but only in these 4 words, two double-quotes are interpreted as being one double-quote, which does not end the string.

Example of printing double quotes:

```
." ""Hello"""
```

will print:

"HELLO" (including the double quotes)

Example of the definition of a string containing double quotes:

S" cr ."" Hello "" 1 . cr 2 ." SLITERAL teststring

Once a string has a name it can be typed like this:

teststring type

will print:

cr ." Hello " 1 . cr 2 .

It can also be evaluated if it is valid wabiForth like this:

teststring evaluate

will print:

Hello 1
2

As you see with the double double-quotes **EVALUATE** can also handle string-words correctly.

In addition the following definitions are available:

>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'

$>CAPS (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. Case-insensitive.

An example of the use of **$>CAPS.**

```
s" a string without capitals" sliteral mystring
: show mystring type ;
: makecaps mystring $>caps ; \ << 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 '**mystring $>lower**' will change it back to all lower characters.

Overflow protected string variables

There seem to be two problems with the strings as defined in the ANSI standard. One is that the functionality seems limited compared to other programming languages. Anybody who has ever programmed in BASIC with MID$ etc. will recognise the feeling. The second problem is that there is no overflow-protection build in. And that can be a real problem.

For a quick lesson on the deficiencies of your Forth-system, pick up a Basic book and start implementing all of its examples...

John S. James - 1985
Rochester Forth Conference

Fortunately, with the ANSI standard words it is possible to provide extended string-functionality by creating overflow-protected **string-variables.** The following definitions are all done in hi-level wabiForth, with the exception of **CADD$ and COVER$.** For most words their performance depends mostly on the efficiency of the word **'MOVE'.** For the character oriented words however, this is different. These two word are both over 10x faster in assembly compared to hi-level Forth.

A string-variable consists of 2 parts: a named definition, created in the dictionary; and a buffer, created in the data-area of wabipi4, where the actual string resides.
The data-area has ~2860MB available space, and, consequently, a string-variable has a maximum length of ~2860MB. To put that into some context: all works, sonnets and plays, of Shakespeare taken together, add up to around 5.4 million characters. Which clearly illustrates how long a 2860MB string actually is.

All string-variable related words will abort if the input is not a previously defined **$var.** This avoids string-related words erroneously overwriting parts of memory.

These are the definitions:

GENERAL FUNCTIONS

$VARIABLE (max_len _$VARIABLE_ <name>): $VARIABLE is used to define string-variables with a maximum length.

An example: **1000 $VARIABLE example$**

defines a string variable named **example$** with a maximum length of a 1000 characters, and allocates the buffer.

$DEL ($var --): deletes the given string-variable by setting the length to zero. The string itself is still visible with words like **DUMP** or **C@.** If you do not want that, use the word **$CLEAN** after **$DEL.**

$CLEAN ($var --): fills the empty (unused) space of the given string-variable with zeroes. Any contained string is not overwritten, only the unused space in the buffer after the string is zeroed.

$TYPE ($var --): prints the given string-variable. Only prints the first 2500 characters. Does the same as '**GET$ 2500 MIN TYPE**'. Defining comparable words, if something different is wanted, is easy.

$ADR ($var -- addr): for a given string-variable returns the start address of the contained string.

$LEN ($var -- length): for a given string-variable, returns the length of the contained string.

$MAX ($var -- max): for a given string-variable, returns the maximum length the contained string can have.

STRING MANIPULATION FUNCTIONS

These definitions change an existing string by adding to or deleting from a string variable.

ADD$ (addr len $var --): add the string defined by 'addr len' at the end of the string contained in the given string-variable. If the available space in the buffer is shorter than the length of the string to be added, only the available space is filled. Thus ensuring that the maximum length is never overwritten. No error-messages are generated if this happens.

ADD$LF (addr len $var --): as ADD$, but also adds an ASCII <LF> character (=10) after the added string. Sometimes handy.

ADD$BL (addr len $var --): as ADD$, but also adds an ASCII <space> character (=32) after the added string. Sometimes handy.

CADD$ (char $var --): adds a character at the end of the string contained in the given string-variable, provided there is space left in the buffer. This word can, for instance, be used to add ASCII-control characters to a string. This definition is an inlinable primitive in assembly.

COVER$ (char pos $var --): overwrites the character at position **pos** with the character **char**. Pos must be inside of the string. This definition is an inlinable primitive.

OVER$ (addr len start $var --): overwrites the string contained in the given sting-variable with the string defined by 'addr len' by overwriting from the point defined by 'start'. If the available space in the buffer is not enough to contain the whole of the string to be added, only the part which fits is written. If start is outside of the string, no action is done.

INSERT$ (addr len start $var --): inserts the given string into the string contained in the given string-variable. In any situation, the maximum length is kept.

ASNEW$ (addr len $var --): sets the string in the given string-variable to the string defined by 'addr len'. Any existing string is removed from the string-variable.

RSUB$ (len $var --): removes 'len' number of characters from the right side of the string contained in the given string-variable. If 'len' is bigger than the length of the string contained in the string-variable, the whole string is deleted.

LSUB$ (len $var --): removes 'len' number of characters from the left side (ie the start) of the string contained in the given string-variable. If 'len' is bigger than the length of the string contained in the string-variable, the whole string is deleted.

MIDSUB$ (start len $var --): removes the part of the string contained in the given string-variable defined by the start and the length of the part contained. If the start-point lies outside of the string, nothing happens.

STRING SELECTION DEFINITIONS:

These words select part of a string in a string-variable, but do not change the string. The programmer can use the returned addr and length to change the content of the string if so wanted.

GET$ ($var -- addr len): for a given string-variable, returns the address and length of the contained string.

RSEL$ (len $var -- addr len): selects 'len' number of characters from the right side of the string contained in the given string-variable by returning the address and length of the selected part.

LSEL$ (len $var -- addr len): selects 'len' number of characters from the left side of the string contained in the given string-variable by returning the address and length of the selected part.

MIDSEL$ (start len $var -- addr len): selects 'len' number of characters from the starting point 'start' of the string contained in the given string-variable by returning the address and length of the selected part.

EXAMPLES OF THE USE OF STRING VARIABLES

Creating a new string-variable:

```
100 $VARIABLE MYSTR$  \ creates an empty string-variable
s" This is example text..." mystr$ add$ \ adds a string

mystr$ $type
   output: This is example text...
```

Setting a new string in a string-variable

```
s" A second, new, sentence." mystr$ asnew$

mystr$ $type
   output: A second, new, sentence.
```

Adding a string to itself:

```
mystr$ get$ mystr$ add$  \ copies the string after itself
```

```
mystr$ $type
   output: A second, new, sentence.A second, new, sentence.
```

Deleting a string:

```
mystr$ del$

mystr$ $type
   output:
```

MIXING STRING-VARIABLES AND ANSI STRING WORDS

The string-variables work well together with the ANSI string-words. The following example is a demonstration of this. The demo also shows the strengths and limits of the string-system and WabiPi4 in general.

The demo does the following:

1. **create a string-variable**, named 'rn', with a maximum length of 1 billion (a 1 with 9 zeroes) characters.
2. **fill this string** with 1 billion random characters.
3. **search for the text "Wabi"** and count how often it occurs in the 1 billion character string.

There are few systems where you can leisurely create a 1 billion character long string, or search for a sub-string within that string. The reason for this might be that such functionality is very rarely needed. But let's not limit ourselves by such an argument! WabiPi4 can handle such a string with aplomb.

T**he execution-time for the demo** is around 15s. Of that 15 seconds, 13 seconds is spend filling the string with 1 billion random characters. The actual search and count takes around 2 seconds.

```
1000000000 constant $size \ yes, that is 9 zeroes...
$size $variable rn

: rndm$ ( n $var -- ) ." - filling "
   swap 0 ?do
      94 rndm 32 +      \ make a random ASCII character
      over cadd$        \ add character to string
   loop drop ;

: $count ( addr len $var -- count ) ." - counting "
   get$ 2swap 2>r       \ put search-string on r:
   0 -rot               \ counter
   begin
      2r@ search
   while                \ s:( c, a, l )
      rot 1+ -rot       \ add 1 to counter (c)
      swap 1+ swap      \ add 1 to address (a) for next search
   repeat
   2drop 2rdrop ;

: go ( -- ) \ prints number of times "Wabi" occurs
   rn $del              \ clear string
   $size rn rndm$       \ fill string with random characters
   s" Wabi" rn $count   \ count how often "Wabi" occurs
   . ;
```

If you type: **T[GO]T.**

you will get as output something like this:

t[go]t. - filling - counting 11 15'168'040us

which tells you that it took 15.2 seconds to find 11 occurrences of the string "Wabi". Isn't it amazing that even a string consisting of random characters knows about Wabi? (and, obviously, any other 4 ASCII character word in existence...)

I have played a lot with this demo. Different search-strings, different lengths of the random string etc. One thing I did was defining a string with 3.002.790.000 random characters. About the maximum possible.

This was the output:

t[go]t. - filling - counting 43 45'547'491us

A 3 billion character long string; created, filled with random characters, searched and 43 matches found. All in 46 seconds... Nice, very nice indeed...!

STRESS-TESTING THE DICTIONARY

The following code does nothing useful. It creates 5 million different definitions in the library in about 55s. Call it again and an ABORT will happen, because the hash-table is full. The system's still usable, but new definitions cannot be added. With **WORD#** . you can see the number of words in the dictionary, with **MK$LF** from below it is 5590205, just below the theoretical maximum of 5592373. The dictionary-space has than been used for around 77.3%.

```
nodupwarn
128 $variable TMPS

: MK$LF
  s" : def" tmps add$
  dup n>string tmps add$bl
  s" ."" Well... This is exiting..."" " tmps add$
  n>string tmps add$bl
  s" . ;" tmps add$lf ;

: 5million fastmode \ fastmode speeds up the process a bit
  cr
  t[ 5000000 0 do
  tmps $del
  i mk$lf
  tmps get$ evaluate
  loop
  ]t. slowmode ;
```

With words like **LWORDS**, **WORD#** and **SEE** you can inspect the results. In the chapter on the HASH-TABLE the 5 million word dictionary, created with the routines above, is used to show the effects of a large dictionary on the performance of a hash-table.

Memory-map and statistics of wabiPi4 system

WabiPi4 is a pure 32bit system. The registers, the cell-size and the address-bus are all 32bit. The databus is 32bit from the perspective of the programmer, but in fact is 128bit to speed up reads and writes.
WabiPi4 supports a 4 or 8 GB Raspberry Pi 4b. For the 8 GB version only the lower 4 GB is accessible. Support for a 2 GB Raspberry Pi 4b might be added in future. A 1 GB pi4b will not be supported.

SYSTEM MEMORY MAP

item	start in memory	end	size
CPU exception vectors	0x00000000	0x00000020	32 B
SOC data-area	0x00000020	0x00008000	~32KB
wabiPi4 system	0x00008000	~0x00099FF4*	~583KB*
user-dictionary	~0x00099FF4*	0x3D800000	983 MB
internal data and variables	0x3D800000	0x3E000000	8 MB
GPU-memory	0x3E000000	0x40000000	32 MB
PIXEL buffers	0x40000000	0x41000000	16 MB
task memory	0x41000000	0x43000000	32 MB
hashtable	0x43000000	0x47000000	64 MB
xt-list	0x47000000	0x49000000	32 MB
data memory	0x49000000	0xFC000000	2864 MB
SOC memory mapped registers	0xFC000000	0x00000000	64 MB

*: depends on the version of the wabiPi4 system

The following definitions return data on the free space in the main sections of the memory:

UNUSED (-- n **):** returns available memory in bytes available for new definitions in the dictionary - ANSI standard word

MEMFREE (-- u **):** returns the total amount of free memory in bytes available on the system

DATAFREE (-- u **):** returns the total amount of free data memory in bytes still available on the system. Both **ALLOCATE** and **TEMPALLOCATE** take their allocations from this pool.

.AVSIZEWORD (-- **):** prints the average number of opcodes (1 opcode is 4 bytes) in a word.
The average is calculated using the total size of the dictionary and the total number of definitions and the average size of a header. It is mainly for informal purposes. It is not a reliable statistic as allotted blocks of memory are also part of the dictionary. A slightly more accurate statistic can be reached by using **ALLOCATE** instead of **ALLOT** when defining blocks of memory. Allocated blocks of memory do not influence the calculation of **.AVSIZEWORD.**

The wabiPi4 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 the address contained in constant **STARTDICT.**

2. **Base-dictionary:** this is where the assembly code resides. Code for the following functionality is contained:

   - the setup of the SOC/CPU
   - initiation of the PIXEL graphic and WAVE sound system
   - input system
   - primitive Forth definitions
   - interpreter
   - optimising compiler

   Parts 1 and 2 together form a primitive FORTH system able to run independently. The base-dictionary starts at the address contained in the constant **STARTDICT** and ends at **STARTHILEVEL.**

3. **HiLevel-dictionary.** This part of the dictionary is where wabiPi4 is defined in its final form. It adds useful functionality like drivers and tools. 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. Executing a **COLD** also reloads this dictionary. The hilevel-dictionary starts at the address contained in the constant **STARTHILEVEL** and ends at **ENDBOOT.** The source-code for the hi-level dictionary is contained in wabiPi4 in a readable format. It starts at the address returned by **SOURCEHILEVEL.**

4. **User-defined definitions.** Occupies the memory from **ENDBOOT** upto **HERE.**

The following words return statistics on the size of different parts of the wabiPi4 system.

STARTDICT (-- u): constant - returns address of first definition

STARTHILEVEL (-- u): constant - returns address of first definition of hi-level part of dictionary
SOURCEHILEVEL (-- u): constant - returns address of source-code of the hilevel dictionary

ENDBOOT (-- u): constant - returns address of first free memory-location after booting but before any definitions are added by the user.

SZSYSDATA (-- u): size of the internal block of memory.
SZBASEDICT (-- u): size of the base-dictionary.
SZHILEVELDICT (-- u): size of the hi-level dictionary.
SZUSERDICT (-- u): size of the user-part of the dictionary.
SZSYSTOTAL (-- u): size of total wabiPi4 system.
SZDICT (-- u): size of dictionary.

Memory allocation

WabiPi4 largely follows the ANSI-standard with regard to memory allocation. The main difference to the standards are the words **FREE** and **RESIZE.**

Both **FREE** and **RESIZE** are CRC-protected, to reduce the chance of unwanted data-erasure due to a spurious error in a program. The words **FREE** and **RESIZE** 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.

NON-STANDARD WORDS:

In addition to the standard words there are two words focused on temporary data. They are: **TEMPALLOCATE** and **FREETEMPDATA. TEMPALLOCATE** is intended to allocate a buffer for data which is only needed on a temporary basis. **FREETEMPDATA** frees all memory allocated to temporary data.

The rationale is that with these words temporary and permanent buffers can be separated by the programmer. This makes it possible to clear the temporary buffers from memory once no longer needed. This usually avoids complex and time-consuming garbage-collection.

ALLOCATE (u -- addr t/f): allocates a block of memory at least u bytes long in the data-area. To optimise cache-allocation, the allocated block of memory will always be 64 byte aligned. 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.

Memory reserved with **ALLOCATE** can be faster than memory reserved with **ALLOT.** This is due to different cache-specifications for the memory. Memory in the data-area is flagged as 'EXECUTE-NEVER'. Because of this the default aggressive pre-loading of the instruction-cache is avoided.
For programs with a lot of data being accessed in a random or semi-random fashion, the difference in speed might be noticeable.

ALLOCATE works from high to low data-memory, and all empty data-memory space is available. A maximum of 2864 MB (=~3.000.000.000 bytes) of memory (can be allocated, in 1 block if so needed.

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

RESIZE (addr - f/t): 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 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.

TEMPALLOCATE (u -- addr/u, f/t): allocates u bytes for temporary data. If u is larger than the available free memory, a true-flag is returned. If the allocation is successful, the address of the buffer and a 'false' flag are returned. The allocated block of memory will always be 64 byte aligned, to optimise cache-allocation.

FREETEMPDATA (-- u): frees all temporarily allocated buffer space in memory. Returns the actual number of bytes freed. This can be larger than the total size of the requested buffers due to the 64 bytes alignment of memory blocks.

Example:

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

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

```
1000000 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

#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

Wordlists (a.k.a. vocabularies)

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

WabiPi4 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:

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 occurs.

ONLY: According to the ANSI-standard the minimum search-order is implementation dependent. In wabiPi4 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.** This word is not part of the ANSI standard, but in common use.

HIDDEN (--) A definition with **HIDDEN** after it will not be printed with **WORDS.**

The combination of vocabularies and **HIDDEN** is pretty powerful. **HIDDEN** separates the functionality of 'do_not_print' from the coding functionality of a library. Using vocabularies to control printability of definitions can lead to very hard to find bugs, especially if the programs get larger.

.VOCS (--) Prints an overview of all defined vocabularies. It 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
```

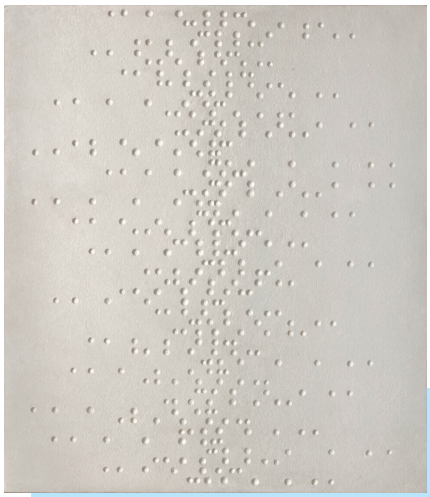
ORDER shows the present search-order**.** Than **VOCABULARY** is used to make a new vocabulary with the name **NEWVOC. ALSO NEWVOC** adds **NEWVOC** to the top of the search-order list**. DEFINITIONS** ensures that new definition are added to the new vocabulary**.** Finally **ORDER** shows the final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

Random number generator

Generating high quality random numbers reliably is surprisingly complex, and care has to be taken to avoid pitfalls. WabiPi4 has 3 different methods for the generation of random numbers. A very fast, deterministic generator of excellent quality called **RNDM**. And a slow non-deterministic method which generates true random numbers called **TRNG**.



GERHARD VON GRAEVENITZ

Weiße Struktur (Zufallsverteilung auf vertikaler Achse II)

1960

The wabiSystem also needs random numbers for system-management purposes, in some cases quit a lot. To avoid loosing the deterministic character of the main generator (**RNDM**), these numbers are generated by an independent 3rd generator called **SYSRNDM**. **SYSRNDM** might be of use if a programmer needs a random number but wants to keep the internal state of the main **RNDM**-generator intact. **SYSRNDM** is a bit slower than the main method, and the quality of **RNDM**-generation is only fair.

These three methods should cover all needs, apart from serious cryptographic work.

TRUE RANDOM NUMBERS

are generated by the iProc-RNG200 true random generator from Broadcom. The output *cannot* be predicted, there is no periodicity, and it is impossible to reseed this generator. The generation of random numbers depends on the random jitter of ring-oscillators.
This True Random Number Generator (**TRNG**) conforms, according to Broadcom, to NIST SP800-90B, NIST SP800-22, FIPS 140-2, and BSI AIS-31. However, the wabiPi4-implementation itself has not been certified.

The **TRNG** is slow. After a full reboot of the system, it takes the hardware 0.7s to generate the first random number. This delay guarantees that even the first generated number is fully unpredictable. After the first number, a new 32bit random number is generated every ~112000c. So you can generate around 16000 random numbers per second.

To speed up the availability of a true random number, the **TRNG** has a buffer to store up to three random-numbers. In case a number is available in the buffer, reading this number takes around ~3350c. WabiPi4 handles this automatically.
The **TRNG** is typically used to seed the other random generator once or periodically. The effect is fast generation of unpredictable random numbers.

Sanity-check of TRNG

To comply with regulations governing random-number generation, such as BSI AIS-31, a hardware sanity-check of the generator must be performed before retrieving a random-value. WabiPi4 complies with this rule.
If the sanity-check shows that an error within the **TRNG** generator was flagged by the hardware, a full reset of the TRN-generator is performed and the **TRNG** error counter raised by 1. This count is available in the value **TRNGERRORCNT**.

Please note: the returned values are always unpredictable, also after a full reset of the **TRNG**. A reset of the TRNG does not reset seeds or anything comparable to that, as seeds do not form the basis of the random numbers generated.

Definitions for the True Random Generator

TRNG (n -- m): returns a random number between 0 and n minus 1 in ~3350c. If the 3-value buffer is empty then it takes ~112700c to return a next random number.
A sanity-check is performed before a number is returned. In case the sanity-check fails, a hardware-reset of the **TRNG** is done to ensure correct functioning. In case of an error, **TRNGERRORCNT** is also raised by 1 to flag the failure.

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

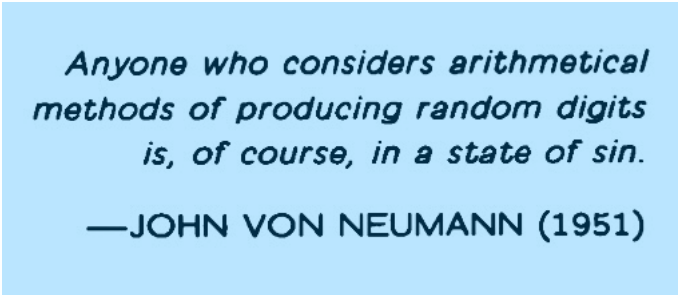
TRNGERRORCNT (-- u): Value containing the number of times an error with the TRNG-generator was reported by the hardware since booting of the system. This should stay 0x0 at all times.

RESETTRNG (--): resets and (re)starts the true random generator (**TRNG**). It is called during start-up of the system. After a reset the generated numbers are always fully unpredictable. It takes 0.7 seconds after a reset before the first random number is available. There is no way of halting the TRNG-generator.

THE MAIN RANDOM-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 9c.

This method is intended as an efficient, 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 by Blackman and Vigna). However, some unlikely sequences of equal numbers are known. Fortunately these unlikely sequences are so rare that they cannot be measured statistically. To put in perspective how rare: the chance that you encounter the end of the solar-system is larger than the chance that you will encounter such an unlikely sequence of numbers. And even if you would encounter such an unlikely sequence, you would not notice it.

The authors state that, based on their testing, this method can be used for any serious task. For any critical work you should test for suitability before use. It should also be noted that the method is not intended for serious cryptographic work.

The XOSHIRO-method has been put into public domain by David Blackman and Sebastiano Vigna, and can thus be used legally for any and all tasks.

The main words in wabiPi4 are **RNDM, TRNG and FRNDM**

RNDM (n -- m): returns a random number between 0 and (n minus 1) in ~9c.
As the XOSHIRO generator resides permanently within the NEON-unit of the ARM-processor it has no influence on - and is not slowed by - the state of the data-caches. In other words, dirty cache-lines do *not* slow the generator down, as would happen with memory based generators. This is especially beneficial in situations where data is read or written at a high rate and at random from or to a large block of memory.

RNDM32 (-- u): returns a 32bit random number in ~9c. It uses the same generator as **RNDM**.

RNDM and **RNDM32** are not used by internal words or by the system. Their status only depends on the user-program.

FRNDM (-- f): returns a 32b floating point number between 0 and 1. This generator is also based on the fast method described above. As a first step a random 32b integer is generated, which is then converted to a floating number between 0 and 1. It returns a number in ~12c.

RESEEDING THE RANDOM GENERATOR

The XOSHIRO algorithm uses 4 internal 64bit seeds to generate the random values, the sequence of random values is exclusively defined by these 4 seeds.
The default method of setting seeds is using **RNDMSEED**. It uses a double value from stack to fill the 4 64 bit seeds, followed by 30 dummy runs to prime the random generator.
A more formal way to reseed the generator is with the word **RNDMTOSEED**. With this word the 4 seeds can be set to any value, and no dummy runs are done. If the present values of the seeds need to be preserved for later use, the word **RNDMGETSEED** copies the seeds to the stack.

RNDMSEED (d --): uses a double value to load the 4 seeds and performs 30 dummy runs to prime the random generator. A quick and easy way of randomising to a new sequence. Any value is acceptable as input. Gives 2^64 different sequences of random numbers, but there is no guarantee that these sequences do not overlap. For normal use totally irrelevant, see **JUMP GENERATION** a bit further on for a bit of background.

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 might lack randomness.

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

.RNDMSEEDS (--): prints the 4 seeds.

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

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

RANDOMIZE (--): sets the seeds of the random generator to four random values. The internal true random generator is the basis for this function, so predictability after **RANDOMIZE** is zero.

JUMP GENERATION

In certain rare use-cases it is important to define 2 or more subsequences of random values which with 100% certainty do not overlap. For a linear method, like the XOSHIRO algorithm, 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, without actually generating all these numbers. The jump-function in wabiPi4 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 maximum rate of random value creation with wabiPi4 is around 150 million values/s. Generating 2^128 samples at this rate would take around 7.2x10^22 years. The jump function saves time by performing this calculation in 5.8 µs. A very nice example of a smart algorithm which is faster than brute-force calculation. In this case 3.9x10^35 times faster.

RNDMJUMP (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.

THE SYSRNDM GENERATOR

This is a basic 32bit random generator with two 32bit seeds, following the **XorShift** principle of Marsaglia, and using the optimised symmetrical setup as published by professor Brent. The period is 1.844*10^19. If you generate random samples at the maximum rate, 120 million/sec, this generator returns to the starting point after 4871 years.
This generator is used by the system when it needs random-numbers. The consequence is that the exact state of this generator is unknown. And so experiments using this generator are not guaranteed to be repeatable.
If you need a quick random number though, while using the main RNDM generator for something big, like a massive simulation, this generator comes in handy.

SYSRNDM (n -- rndm): returns a random number between 0 and n-1 in 10-13 cycles. This generator is memory-based and thus suffers from performance-degradation if the amount of dirtying of cache-lines is high. Performance in those cases can be significantly slower than 10-13 cycles.

SYSRNDM32 (-- rndm): returns a 32bit random number in 10-13 cycles.

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

SYSRNDMRESET (--): resets the sys-random generator to the situation after a **BOOT.**

SYSRANDOMIZE (--): loads the two seeds of the sys-random generator with values from the true random generator.

System constants, variables and values

The wabiSystem depends on a range of variables, values, constants and tables for a correct flow of actions and in general to keep track of stuff. A lot of these are available to the programmer. For instance if you are interested in the internals of the system, or if you want to add new functionality.

The list here shows just some...

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

LATESTWORD (-- addr): value, contains the address of the name of the most recent defined word. '**C!+ TYPE**' prints this name.

LASTXTHERE (-- addr): variable, contains the first memory location of the last XT compiled. Used during optimisation.

DEFINING? (-- n): value, contains a flag which, if true, signifies that the system is compiling a definition. This is NOT the same as '**STATE**'. '**STATE**' can temporarily be put on hold with '**[**' whereas **DEFINING?** tracks if a colon was closed by a semi-colon or not.

COLDCOUNT (-- value): value, returns the number of times a COLD was executed by the system since the last reboot.

TRANS (-- addr): constant, contains base-address of buffer for trans-area (a scratch-buffer which temporarily holds strings during compilation).

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

IBADDR (-- addr): variable, contains the presently used input-buffer address. Usually points to the Text Input Buffer (TIB), but this is changed during the evaluation of text-files.

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 an unsecure memory-block. Unsecure is a term related to the ARM-processor. Programs requiring the HYPER-state of the ARM-processor can only run in, and read/write data from/to unsecure memory.

UNSECUREDATA (-- addr): variable, pointer into available unsecure memory. Some experimental data, for potential future functionality, resides here.

UARTREC (-- value): 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 (-- value): constant, contains length of the UART receive buffer.

GIC_BASE (-- value): constant, contains the base-address of the Global Interrupt Controller (GIC). See the ARM documentation for more details.

FONT1 (-- addr): constant, contains address of the font-table

Time and time-measurement related words

WabiPi4 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 1.8 GHz. Both start counting from zero after a full reboot of the system. The CPU-cycles counter starts counting as soon as the CPU is powered up, whereas the micro-second counter starts counting as soon as wabiPi4 is started.
The micro-second counter is intended 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** (and also because 5.6448 MHz divided by 128 gives 44100, a number useful for sound-generation). It takes more than 103.000 year before this underlying counter wraps around to zero.

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

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.

MCS (-- u): returns the elapsed time in microseconds since the last reboot, as an unsigned single. As **MCS** wraps to zero after 4295 seconds, care should be taken that this 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. A relic from the olden days, when computers struggled measuring periods shorter than milliseconds. But part of the ANSI-standard.

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

SEC (-- u): returns the time in seconds since the restart of the system, as an unsigned single. **SEC** wraps around to zero after ~136 years.

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

THE CPU-CYCLE COUNTER

is accessed with the word **CPUCYCLES.** It starts counting at 0x0 after power-up of the CPU and after **RELOAD.** The counter counts the actual clock-cycles used by CPU-core0, which means that this counter counts up at 1.8 GHz. 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** counts the actual number of cycles. 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 325 years.

Executing **CPUCYCLES** twice directly after each other will show a difference of (usually) 9 to 13 cycles. These 9 to 13 cycles are the time required to put the values on the datastack.
Subtracting 9 to 13 cycles from a measurement will give you the time taken by whatever is between two **CPUCYCLES.** Which makes clear that averaging is needed to get really reliable measurements.

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 most common time shown is 12 cycles. By running **MINIMUM** ten times you can sometimes see the effect of the cache-system and the branch-prediction system.

Contrary to the Cortex-a53 in the Raspberry Pi3b+, the timing of the Cortex-a72 in the Raspberry 4b is called 'deterministic' by the ARM-cooperation. That is the timing of individual tasks is predictable.
That this predictability is not always self-evident shows the following example:

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

Here '**1 1 + drop**' is added to the line of code in the '**MINIMUM**' definition above. The result printed can be anything from 9 cycles to around 1900 or so.
The interesting thing is that the most common timing returned is 9 cycles. Why this longer line of code usually runs faster than '**MINIMUM**' above is unknown to me.

OTHER WORDS FOR TIME-MEASUREMENTS:

.UPTIME (-- **):** shows how long the system has been running since the last start/restart of wabiPi4**.** It uses the format 00h00m00s**.** When programmed correctly, wabiPi4 is a very stable system, uptimes of thousands of hours are entirely possible**. .UPTIME** can track the uptime for at least ~400 years**.**

Interval measurements

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

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

]T (dmcs -- dmcs): calculates the difference in µs with the last **T[**

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

For instance:

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

prints "278 us" or something like that**.** In other words: putting I on stack and dropping it again one hundred thousand times takes around 278 µs**.** But it can also take 223, 333 or 388 µs**.** That depends on the state of the CPU**.**

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

]C (dcpucycles -- dcpucycles): calculates the difference in cycles with the last **C[**

]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 ;
: go c[ tst ]c. ;
go
```

prints "501082 cycles" or something like that**.** (but it sometimes prints a number around 400000 or 700000 cycles)

If you experiment with **CPUCYCLES** you will see that measurements with **CPUCYCLES** have a high variability**.** That variability reflects the reality of modern CPUs**,** which are very fast on average but unpredictable for the performance of individual pieces of code**.**

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:

```
17779 us -> 32.0 cycles
```

The 17780 µs reported is the time (in micro-second) it takes to call the word **TST** a million times. The 32.0 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 ~17.8 nanoseconds. To make clear how short a time that is: light travels only 5.3 meters in 17.8 nanoseconds. In that *very* short time wabiPi4 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 measures microseconds. As many individual timers as needed can be set up this way. Note that the timing of the two different versions is very comparable.

```
2variable timera
: SETTIMERA ( -- ) dmcs timera 2! ; ( 17.0c )
: GETTIMERA ( -- d ) dmcs timera 2@ d- ; ( 17.7c )
```

or like this, using the Wabi-specific 2VALUE:

```
0 0 2value timera
: SETTIMERA ( -- ) dmcs to timera ; ( 17.7c )
: GETTIMERA ( -- d ) dmcs timera d- ; ( 17.0 )
```

SETTIMERA (--): gets a baseline, which functionally is af if timer A was reset 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 at least 11 microseconds (i.e. 11 millionth of a second) to do ten thousand empty do...loops with wabiPi4. Longer times are also possible.

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 constantly polling the elapsed time, and thus is reasonable accurate.

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

BLINK (--): waits for 100 milliseconds before continuing execution. This is the time of a fast blink of the eye, although some sources give a longer duration like 330 milliseconds as the duration of a blink.

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

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

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

Historic functionality

The Raspberry Pi can be seen as a successor to the home-computers from yesteryear. And the Cortex-A72 processor is a member of a long line of ARM-processors, going back almost 40 years.

In the ARM-processor a few remnants of this long history are still visible. Features that were once useful but are now outdated. The same is true of some of the features of early home-computers.
It is my intention to support these features, if only out of respect for all the brilliant people who developed computers into what they have become.

Scratch-byte

In case the 4'033'685'212 (~4 billion) bytes of free memory available in wabiPi4 are not enough, the ARM-CPU contains 1 whole extra byte of scratch-storage in the UART-section.
To make the use of this extra byte as convenient as possible, wabiPi4 contains 2 words: **C!SCRATCH** and **C@SCRATCH.** With these you can store a byte for later use, and fetch it again.
On the pi4b the access-time of this byte-field is not exceptionally long, namely ~15 cycles (around 220 cycles for the pi3b+). Normal memory is at least 15 times faster, but if you really need that 1 extra byte, you'll take all you can get...

To the cynics who just have to notice that these two definitions consume 160 or so bytes of the available RAM; and that it would have been 160 times more efficient to simply leave these words out out; I can only say: 'how stodgy you are...'.

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

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

Fast and slow mode

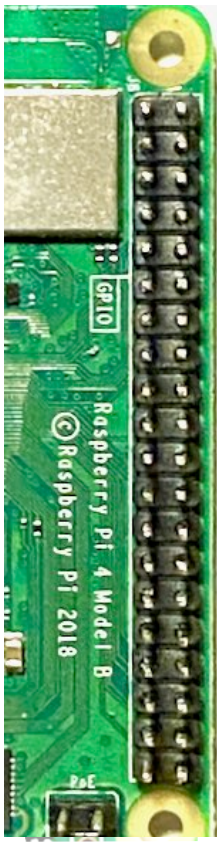
The Sinclair ZX80 computer, introduced January 29, 1980, had a FAST and SLOW mode. The FAST mode resulted in faster execution of programs, at the cost of screen-updates. The CPU in fact simply stopped updating the screen.
This turns out to be a good idea for wabiPi4 as well! By limiting the access of the GPU to memory, the CPU gets more and faster memory-access. Programs which are limited by the speed of the memory can be up to 25% faster in **FASTMODE.** For an example, see the 5MILLION program in the chapter on **overflow protected string-variables.**

FASTMODE (--): limits the memory-bandwidth of the GPU and makes this available to the CPU. This might result in up to 25% faster performance.

SLOWMODE (--): sets the available memory-bandwidth of the GPU to normal.

GPIO and onboard LEDs

The Raspberry Pi4b 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 wabiPi4 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 shows you that the GPIO-numbers are not the same as the pin-numbers. All wabiPi4 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). These two pins are unfortunately not available to the user, even if no HAT is present.

All other GPIO's can be 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 one of 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 if they are not in use.

Use of GPIO-pins in wabiPi4: wabiPi4 has default settings for 8 GPIO's. The UART1 on pin 8 and 10 is critical for communication with wabiPi4, the functionality should not be changed.
The other pins (I2C and PCM) can be changed to other functions.

pin	GPIO	function
pin 8	gpio 14	UART1 - TX
pin 10	gpio 15	UART1 - RCV
pin 3	gpio 2	I2C - SDA1
pin 5	gpio 3	I2C - SCL1
pin 12	gpio 18	PCM - bitclock
pin 35	gpio 19	PCM - left-right
pin 38	gpio 20	PCM - Master clock
pin 40	gpio 21	PCM - serial data out

The Pi4b has a lot more IO-hardware on board than the Pi3b+. The **ALT1** and **ALT2** functions have no use for most programmers. They are for functionality like a liquid crystal bus, alternate databus and JTAG interface. An overview without these alternate functions is easier to read and grasp.

Simplified alternate functions Raspberry Pi 4B

ALT5	ALT4	ALT3	ALT0	GPIO	pin	pin	GPIO	ALT0	ALT3	ALT4	ALT5
				3.3v	01	02	5.0v				
SDA3	CTS2	SPI3_MOSI	SDA1	gpio2	03	04	5.0v				
SCL3	RTS2	SPI3_SCLK	SCL1	gpio3	05	06	0v				
SDA3	TXD3	SPI4_CE0_N	GPCLK0	gpio4	07	08	gpio14	TXD0	SPI5_MOSI	CTS5	TXD1
				0v	09	10	gpio15	RXD0	SPI5_SCLK	RTS5	RXD1
RTS1	SPI1_CE1_N	RTS0		gpio17	11	12	gpio18	PCM_CLK	SPI6_CE0_N	SPI1_CE0_N	PWM0_0
SPI6_CE1_N				gpio27	13	14	0v				
SDA6				gpi022	15	16	gpio23				SCL6
				3.3v	17	18	gpio24				SPI3_CE1_N
SDA5	CTS4	BSCSL SDA/MOSI	SPI0_MOSI	gpio10	19	20	0v				
SCL4	RXD4	BSCSL/MISO	SPI0_MISO	gpio9	21	22	gpio25				SPI4_CE1_N
SCL5	RTS4	BSCSL SCL/SCLK	SPI0_SCLK	gpio11	23	24	gpio8	SPI0_CE0_N	BSCSL/CE_N	TXD4	SDA4
				0v	25	26	gpio7	SPI0_CE1_N	SPI4_SCLK	RTS3	SCL4
				HAT id	27	28	HAT id				
SCL3	RXD3	SPI4_MISO	GPCLK1	gpio5	29	30	0v				
SDA4	CTS3	SPI4_MOSI	GPCLK2	gpio6	31	32	gpio12	PWM0_0	SPI5_CE0_N	TXD5	SDA5
SCL5	RXD5	SPI5_MISO	PWM0_1	gpio13	33	34	0v				
PWM0_1	SPI1_MISO	SPI6_MISO	PCM_FS	gpio19	35	36	gpio16		CTS0	SPI1_CE2_N	CTS1
SPI5_CE1_N				gpio26	37	38	gpio20	PCM_DIN	SPI6_MOSI	SPI1_MOSI	GPCLK0
				0v	39	40	gpio21	PCM_DOUT	SPI6_SCLK	SPI1_SCLK	GPCLK1

WABIPI4 WORDS FOR ALT FUNCTIONS OF THE GPIO'S

WabiPi4 contains definitions for setting and reading **ALT functions** of a GPIO, for setting and reading a GPIO and for pull-up/down resistors. Interrupts related to the GPIOs are not covered by **wabiPi4** itself. This might be added in future as functionality, but don't count on it..

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

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

function	value for input	WabiPi4 constant
input	0	**GPIO_IN**
output	1	**GPIO_OUT**
ALT0	4	**GPIO_ALT0**
ALT3	7	**GPIO_ALT3**
ALT4	3	**GPIO_ALT4**
ALT5	2	**GPIO_ALT5**

GETFUNCGPIO (gpio# -- n/-1): returns the present alternate function of port gpio#. Valid input values are 0-53. **GETFUNCGPIO** will return -1 if an incorrect input-value is used. This definition is intended for debugging and testing.

Examples:

gpio_alt5 20 setfuncgpio
will set GPIO 20 to alternative function 5

20 getfuncgpio
will then return a 2 (see tabel above!)

WABIPI4 WORDS FOR GENERAL I/O OF THE GPIO'S

Naturally it is possible to use the GPIO pins as straightforward input or output. Once a GPIO pin is set as input or output, the following words are available to set or read the value of a pin and to control the pull-up resistors. In addition it is possible to control the drive-strength and hysteresis of pins.

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 ~19.5 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 in the Raspberry Pi4 can only source around 8mA per pin! Much lower than for instance an Arduino, which can source 25mA. Thus the use of a buffer to drive a LED or other effectuators is common.

GETGPIN (gpio# -- 0/1): gets the input-value at port gpio#. The maximum frequency with which ports can be read is ~75 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 more reliable than reading a level once, but takes a couple of dozen nanoseconds 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. Valid gpio# are 0-53, valid input for action is 0-2. If input is invalid then **SETPULLUD** returns without performing an action.

The following table shows what input gives which action.

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

Please note that the logic for pull-up and pull-down is reversed compared to the values for the Pi3!

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**. By setting the pull-up/down resistors to high or low the pin_in will then act as high-impedance output with clearly defined levels.

On the Raspberry Pi 4B it is possible to set the drive-strength, slew-rate and hysteresis of the GPIOs. This feature is not well documented.

Setting values is not done per GPIO, but per group of GPIOs. There are 3 groups. Group 0 contains GPIOs 0-27, group 1 contains GPIOs 28-45 and group 2 contains GPIOs 46-53. The settings are always the same for all GPIOs within a group. On the connector of the RPi4 GPIOs 0-27 available, so only group 0 is relevant.

There are three options available for each group:
```
 1: drive-strength     values 0-7
 2: slew-rate          values 0-1
 3: hysteresis         values 0-1
```

The drive strength specifies at which current (measured in mA) the ports in a group are guaranteed to reach correct output-voltages. The values 0-7 correspond to 1-8mA. The default setting is 8mA.

Please NOTE: this sets the drive-strength, it does NOT set a limit on current.

The maximum current flowing is only limited by the internal and external resistance. The current should never be allowed to go higher than 8mA per GPIO.

The hysteresis can be 0=disabled or 1=enabled. This value only has effect for GPIOs used as input.

The slew-rate can be 0=limited (=slower) or 1=unlimited (=faster)

Coding of the three options:
The value for drive-strength is coded into bits 0-2, the hysteresis is bit 3, and the slew-rate is bit 4. The default value is hexadecimal 0x1F for group 0 and 1, and (for unknown reasons) 0x1E for group 2. 0x1F maximises drive-strength (8mA), enables hysteresis and sets unlimited slew-rate. 0x1E is the same but specifies 7mA instead of 8mA as the drive-strength.

PADSPECS (value group --): controls the settings of the GPIO-PADS.
Bit 2:0 of value contain the drive-strength, bit 3 is the hysteresis enable-bit and bit 4 the slew-rate control bit. The value of any other bit is ignored.
Valid input for group are 0, 1 or 2. Values higher as 2 are interpreted as being group 2.

Switching onboard LEDs on/off

There are two onboard LEDs on the rPi 4b. A red 'power' LED and a green 'status' lED. These are normally controlled by the operating system. But as there is no operating system with WabiPi4, they can be controlled from forth with the following words:

GRNON (--): switches green LED on
GRNOFF (--): switches green LED off
REDON (--): switches red LED on
REDOFF (--): switches red LED off

These words are surprisingly slow. Switching a LED 'on' or 'off' takes 10us. The reason is that the switching is done by sending a task to the GPU, to which the GPU sends a reply back. After that the GPU sends a signal to the hardware. Switching a GPIO port directly is at least 500 times faster.

Of bytes, bits and bobs...

Any Forth easily handles individual bytes and bits with words like **AND** and **OR**. However, assembly usually can do this much faster. Especially if there are specific opcodes for a task. WabiPi4 contains a set of primitive words to benefit from the ARM-opcodes for byte and bit handling. Some of these words are over 35 times faster than the corresponding Forth code.

BYTE & BIT REVERSE & REFLECT

focus on handling specific situations. For instance the word **BYTES><**. It reverses the order of the 4 bytes in a 32b word. This is for instance useful in situations where little-endian has to be changed to big-endian, or visa versa.
Programmed in Forth, it takes ~36 cycles to do the reversal. The ARM-processor has a specific opcode for this task, and it reverses the bytes in 1 cycle.

BYTES>< (n -- n): reverses the order of the 4 bytes in a 32b value. Example: 0xAABBCCDD becomes 0xDDCCBBAA.

To see for yourself, try:

```
HEX AABBCCDD bytes>< u.  \ << note the U.
```

-> DDCCBBAA is printed

REFLECT (n -- reflected_n): reverses the order of the 32 bits in the value on top. So the lowest bit becomes the highest and the lowest becomes the highest bit etc. This is not the same as **BYTES><**.

To see for yourself, try:

```
HEX AABBCCDD reflect u.  \ << note the U.
```

-> BB33DD55 is printed

CREFLECT (char -- reflected_char): reverses the order of the 8 bits of a character.

4CREFLECT (n -- reflected_4chars): reverses the order of the 8 bits of each of the four characters within a 32b value. The 4 characters each stay on their position within the 32b value.

HREFLECT (16b -- reflected_16b): reverses the order of the 16 bits of a 16b value.

2HREFLECT (n -- reflected_2_16b): reverses the order of the 16 bits of each of the two 16b values within a 32b value. The 2 values stay in their position within the 32b value.

BIT COUNTING

There are three word concerning the counting of bits: **POPCOUNT, CNTLZERO and CNTTZERO.** Short for population count, count_leading_zeroes and count_trialing_zeroes.
POPCOUNT simply counts the number of set bits in a value on stack. It is f.i. handy to determine the hamming distance of two values. To get the hamming distance, first XOR two values followed by a popcount of the result, giving the hamming-distance of the two values.

POPCOUNT (n -- count): for a given number counts the number of set bits. This is a non-inlinable primitive based on a NEON routine. Returns a count in 3 cycles.

CNTLZERO (n -- count): for a given number counts the number of zeroes till the first set bit. This is a inlinable primitive and it returns a value in 1 cycles (sometimes even in 0.0 cycles).

CNTTZERO (n -- count): for a given number counts the number of zeroes after the last set bit. This is a inlinable primitive and it returns a value in 1 cycle.

BIT ROTATE RIGHT AND ROTATE LEFT

Most processors offer some kind of bit-rotate for a value. A rotate right of a value means that all bits are shifted 1 or more places to the right and that the bits which are dropped from the right, are introduced again on the left.
Such a rotate forms, for instance, the central part of a lot of cryptographic functions. It is easy to program in hi-level forth, but very much faster in assembly.

WabiPi4 supports the following definitions for bitwise rotates:

ROTR (n m -- n (rotate 32 bits right by m bits)): rotates value n right with m bits. Rotates are 32b based.

DROTR (d m -- d (rotate 64 bits right by m bits)): rotates double d right with m bits. Rotates are 64b based.

ROTL (n m -- n (rotate 32 bits left by m bits)): rotates value n left with m bits. Rotates are 32b based.

DROTL (d m -- d (rotate 64 bits left by m bits)): rotates double d left with m bits. Rotates are 64b based.

All bit-rotates are optimised primitives. The single rotate operations **ROTR** and **ROTL** are inlinable, and return a value in ~1.5 cycles. The double rotates return a value in ~7.5 cycles.

Please do not confuse these operations with **ROLL**, a ANSI standard definition where a single value n positions deep on the stack is rolled to the top of the stack.

BIT FETCH & STORE

focus on setting/clearing/retrieving or counting bits. Usually at a given address or in a bit-array. For instance with the words **BIT@** and **BIT!**.

The word **FLAG@** gets the value of a bit from a given memory address and then converts this value of the bit (0 or 1) to a

flag-value (0 or -1). Very easy to do in Forth with **0<>**, 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.

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 reversed 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!'.

Comparing 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 primitive **BIT@** takes ~3 cycles.
For **FLAG@** the difference is a bit larger, 17 cycles in wabiPi4 and 3c as primitive. Avoiding the conversion saves some 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 an unsigned 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[]@**, **FLIPBIT[]**, **1BIT[]!** and **0BIT[]!** are also available.

BIT[]@ (u addr -- 0/1): Gets the value of the n-th bit from the bit-array starting at base-address addr. The value u is a 32 bit unsigned value. The maximum size of a bit-array is thus 128MB. This word is a inlinable primitive and executes in 2.4 cycles, which is surprisingly fast (as are the following words)

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. This word is a non-inlinable primitive which takes 7 cycles to complete. As do the following two words.

1BIT[]! (u addr --): the same as **1BIT!**, but for bit-arrays up to 128 MB long. Somewhat faster than '**1 u addr BIT[]!**'

0BIT[]! (u addr --): the same as **0BIT!**, but for bit-arrays up to 128 MB long. Somewhat faster than '**0 u addr BIT[]!**'

FLIPBIT[] (u addr --): flips bit 'u' in the bit-array starting at address from 0 to 1 or from 1 to 0. **FLIPBIT[]** can for instance be useful in error-correction routines.

FIELD INSERT

Especially when setting 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** might be 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**.**

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**.**

BOBS

BOBS (? --)**:** seriously**,** don't try this one**.** Nothing good comes from it**...**

Cyclic Redundancy Check (a.k.a. CRC)

The ARMv8 processor in the Wabi system has specific opcodes for the generation of CRCs. Generating CRCs this way is a lot faster than software-based algorithms.
Where relevant, the definitions can use either the polynomial **0x04C11DB7** (=ISO/IEEE) or **0x1EDC6F41** (as published by Castagnoli) for the generation of the CRC. The Castagnoli variant is denoted by an extra **'C'** in the name. The Castagnoli version is especially good for longer messages. The IEEE version is in general very good and excels with shorter messages.

The following words are available:

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

CRC8CB (CRC_previous, byte -- CRC_updated): the same as **CRC8B** but using the Castagnoli polynomial.

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

CRC32CB (CRC_previous, n -- CRC_updated): the same as **CRC32B** but using the Castagnoli polynomial.

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.**

DATACRC32C (CRC_previous, addr_c, len -- CRC_updated): the same as **DATACRC32** but uses the Castagnoli polynomial.

FASTCRC32[] (CRC_previous, addr, len_words -- CRC_updated): does the same as **DATACRC32** but only on 32b words. Especially for short sections of data where a CRC is needed rapidly this word is faster. This is for instance relevant with some error-correction routines where millions of CRC-codes are generated while correcting errors.

FASTCRC32C[] (CRC_previous, addr, len_words -- CRC_updated): does the same as **FASTCRC32[]** but uses the Castagnoli polynomial.

$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 before the generation of the CRC**.**

$CRC32C (** CRC_previous, addr_c, len -- CRC_updated): the same as **$CRC32 but using the Castagnoli polynomial**.**

CRCXMODEM (byte_in --): Generates a 16 bit CRC as used by the 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: (-- addr): variable - used in combination with **CRCXMODEM.**

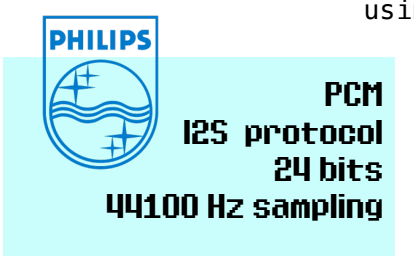
WAVE v2.1 – sound generator

Sound generation was always a 'must have' in the Wabi system. It is known as the WAVE-system, or simply WAVE. Core 2 of the CPU is dedicated permanently to sound-generation, and acts as a de-facto DSP. An easy task for an ARM Cortex-72a core, with its multimedia-focused NEON coprocessor.

The functionality offered by WAVE is much broader than originally intended. A range of functional elements are available. Examples: oscillators, noise generators and VCOs, mixers, an adder and multiplier, an amplifier, an ADSR, static and voltage controlled filters and finally integer and fractional delay buffers.
Sound creation can be controlled with a sequencer on the background, while WabiPi4 does other tasks. Please note that this is not multi-tasking but true parallel processing of two independent cores. Obviously everything is set up and controlled from wabiPi4.

The main purpose of **WAVE** is experimentation with sound-generation. It is **not a *ready_for_use* synthesiser.** You could try to program a synthesiser with WAVE, but it will take quit a bit of programming to reach the functionality of, say, a MiniMoog synthesizer. And a decent physical user-interface with keys, switches and pot-meters is clearly missing.

Version 2 (and later versions) of the Wabi sound-system is a full rewrite compared to v1.0, and is based on different principles. WAVE v1.0 was based on 32b integers, and output was created using Pulse Width Modulation.



Version 2 is based on floating-point calculations and Pulse Code Modulation (PCM). The result is more possibilities, and output has a far higher quality.

Describing all possibilities clearly and in detail is no mean task. Features of the system and explanatory texts in this manual will be added in small steps.

WAVE uses PCM and the I2S protocol to create sound-output. WAVE v1.0 created output with Pulse Width Modulation (PWM). An simple but crude method, with low quality and lots of discernible noise. And a dynamic range limited to less than 20db.

WabiPi4 uses Pulse Code Modulation (PCM). PCM raises the quality and the dynamic range of the generated sound a lot, at the one-time cost of some additional complexity.

WAVE uses 44.1 KHz sampling rate and generates 2 channels of signal. The channels can be used as left and right channel, or as I do, a 'bass'-channel and a sound-channel with a higher pitch. This is up to the user, and depends on the setup of the output. The PCM-module in the Raspberry pi4 can only generate 2 channels of PCM.

CREATING OUTPUT YOU CAN ACTUALLY HEAR

An external Digital to Analog Converter (DAC) and amplifier are needed to convert the PCM signal to an audible audio signal. This is something you have to provide. **WAVE** follows the **I2S** standard and produces a 24bit stereo signal in 64bit frames at 44.1 kHz. This is one of the most common specification available, and any DAC should be able to handle it.

The 3 main **I2S** signals are available at:

gpio 18 = pin 12 =BCLK = bit-clock (@ 2.8224 MHz)
gpio 19 = pin 35 =FS   = frame select (FS)
**   = a.k.a. left/right select (LR)**
gpio 21 = pin 40 =D_out = data-bit out (DO)

for DACs which need it, a master-clock is available at pin 38

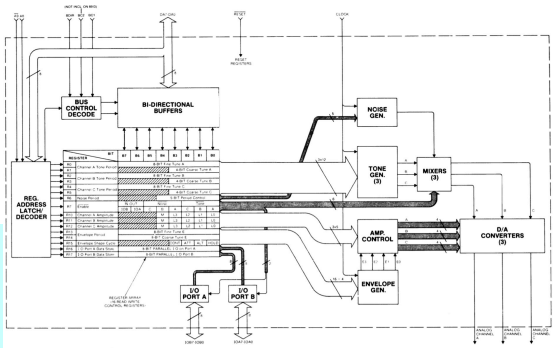
gpio 20 = pin 38 =MCLK = master-clock (@ 11.2896 MHz)

As the frequencies of the signals are pretty high, and the tolerances of most DACs are specified in nano-seconds, the connecting wires must be short. A maximum length of 10cm between the Raspberry and the DAC is advisable. Shielding is normally not needed.

I have tested different DACs and, as hoped, they all functioned well. For instance the 24bit ES9023 DAC, which needs a master-clock, works fine with WAVE.

Of special interest is the MAX98357 3.0 watt amplifier with digital input. What makes this chip so interesting is that Sparkfun sells a tiny board containing the MAX98357 for a few Euro's. This gives you a DAC and a 3.0 watt amplifier combined in one minuscule chip. A godsend for anyone who does not want to solder.
The MAX98357 generates its own master-clock, so 5 wires is all you need (3 signal wires and 2 for supply of power). Add a 4-8 ohm speaker (minimal 3 Watt) and you have sound! With a far higher quality than the home-computers in the 80s and loud enough to be annoying in most sitting rooms. For instance the Visaton **SC 4.9 FL** 8 ohm loudspeaker is a good fit.
Add a bigger amplifier and a decent speaker or set of speakers, and you are in sound-heaven.

THE WAY WAVE WORKS



the venerable ay-3-8910

Any decent home-computer in the 80s used dedicated hardware to generate sound. Either in the form of a sound-generator chip like the ay-3-8910, or a customised UAL chip. Typically such a chip provided 3 or 4 oscillators, a noise-generator and a couple of writable registers to control frequency and functionality like wave-forms, envelope, volume and inter-connections. With some creative programming many different sounds and sound-effects were possible.

WAVE is far more flexible. There is no pre-defined number of oscillators or other functional elements nor a pre-defined structure. Sound is created by the user by defining WAVE elements with appropriate settings, connecting these elements together, and then letting WAVE create the resulting sound.

An ARM-core has enough computing power to update at least 1000 elements in parallel at 44100 times per second. If you want to use a hundred oscillators in parallel to generate a sound, that is perfectly fine. And there is more than enough free capacity to add a lot more elements.

WAVE ELEMENTS

WAVE elements, also called sound-elements, are functional elements which each either generate or manipulate a signal. There are also elements which can log sound-data and other tasks like that.
The minimal setup to get an audible sound is to define one oscillator and an output element and linking these together. Something like this:

```
        440e ~sine   osc1  \ note the float type for Frequency
  osc1 osc1 ~output out1
```

Each individual sound-element is defined by one control-block specific for that one sound-element. So for two sine-oscillators there are two control-blocks. Etc. etc.
A control-block consists of 2 to 15 (or more) 32bit data-words defining what an individual sound-element should do. For instance for an oscillator there is a data-word which specifies the frequency of the sine-wave.

WAVE-blocks are created in the background by the WAVE words. In the example above, **~sine** creates such a block, specific for the sine-oscillator. **~Output** also creates a block, specific for an output-element.
All created control-blocks are put after each other in a linked list (called the WAVE-list). WAVE uses the information in the WAVE-list to generate whatever sound is specified.

Maximum number of WAVE-elements in list
Core 2 runs at 1.8 GHz. The sound-sampling rate of 44100/s in fact means that 44100 times per second the sound is recalculated. Divide 1.8 GHz by 44100 and you see that 40816 CPU-cycles are available per loop (called a tick in the digital sound synthesis-world). With an overhead of 96 cycles the nett number od cycles available is 40720 cycles per tick (ie. per sample/loop) to create the output.

The minimum setup above, a sine-oscillator and output, uses a minimum of 46 cycles per tick. That means that there are still 40674 cycles available.

WAVE-elements vary widely in how many cycles they need to create output. The fastest (**~VALUE**) only uses a few cycles per tick, the slowest (**~24dBVCF**) uses 248 cycles per tick. In practice an average of 25-30 cycles per element is a reasonable estimate. Which means that the CPU has enough capacity to handle and

update around 1300-1600 (40720 divided by 25-30) WAVE elements in parallel.

Unfortunately there is another limit. The size of the L1 data-cache at 32k. As the size of the WAVE-list approaches 32k, the cache performance will slowly deteriorate. The average size of a WAVE element in the list is around 10 32bit cells. This means that if the WAVE-list contains around 800 WAVE-elements, performance will start to go down.

The physical maximum size of the WAVE-list in wabiPi4 is 64kB. But it is unlikely that a 64Kb long WAVE-list functions well. The only way to know for sure is by testing it.

Words are available to track the used and free capacity of WAVE, both for space and CPU-cycles. For small setups of sound-generation the capacity question is irrelevant. For complex setups, maybe hundreds of elements, it is wise to check the use of cycles.

A COMPLETE MINIMAL SETUP IN MORE DETAIL

The code:

```
~reset                    \ reset WAVE and clear the WAVE-list

     440e ~sine   osc1    \ create a 440 Hz sine oscillator
osc1 osc1 ~output out1    \ create an output element

~start                    \ start the sound
  wait                    \ wait 1 second
~stop                     \ halt the sound
```

will sound a 440 HZ sine on both channels for 1 second. After **~STOP**, **~START** will restart the sound again.

Five WAVE words are used in the example:

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

~START (--): starts the WAVE sound generator. It is usually done after setup but can also be done earlier. For instance to hear the effect of individual setup-steps.

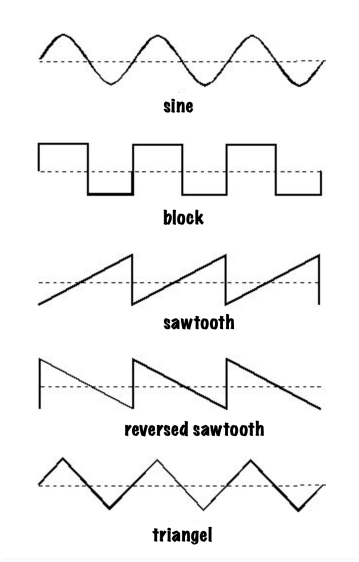
~STOP (--): halts the WAVE sound generator but does not reset it. Executing **~START** will restart it from where it was stopped.

~SINE (f_float --): adds a control-block for a sine-oscillator to the WAVE-list. F_float specifies the wanted frequency. **~SINE** also creates a word with the string following **~SINE** (here **OSC1**). When executed, this word returns the address of the output-field. This address is the location where the output of OSC1, a sine wave with a frequency of 440Hz, is available.
The output-address also forms the base-address for the created sound-element. Relevant if you want to change any of the other data in an element.

~OUTPUT (ch0 ch1 --): adds an PCM-output element to the WAVE-list. Creates a word with the string following **~OUTPUT** ('**OUT1**'

in the example). This word returns the address of the first of two output-fields when called. The output fields contain the actual data send by WAVE to the FIFO of the PCM-unit. The data send to the FIFO is the most significant signed 24bits.
The PCM-unit of the SOC generates the actual pulse-coded output at the GPIOs. WAVE will happily generate signals without an output element, you just can't hear anything.
If you create more than 1 output element in the WAVE list, you still will get output, but usually with loud spurious noise added to the generated sound. The result of two output-elements competing for access to the one FIFO of the PCM-unit.

oscillators



There are presently 8 different oscillators available: 2 sawtooth oscillators, 1 triangle oscillator, 2 block-wave oscillators, a sine-oscillator and 2 arbitrary wave generators. All oscillators have 'frequency' as their main defining characteristic. And any oscillator can be combined with the VCO element to become a voltage controlled oscillator.

Oscillators all have the same frequency-range. The highest frequency is 22.05 kHz (=44.1 kHz divided by 2), the lowest frequency is 0.00001027Hz (0,000020537Hz for the two noise generators).

Oscillators have 5 fields in common:
- float output field - contains output as floating point
- integer output field - contains output as a signed integer
- addition-value - controls the frequency
- phase - for phase-shift of sine and sawtooth signals
- external source address - explained later

Several oscillators have additional specific property-fields.

HOW AN OSCILLATOR WORKS

The simplest oscillator in the WAVE-system is the SAW tooth oscillator. The way it works is very simple: each cycle the addition-value is added to the total-field in mod 2^32 mode. The result is a beautiful saw-tooth shaped wave.
The total is then copied to the integer output field, and the total is converted to a float number between -1.0e and 1.0e. Which is copied to the float output field.

The saw-tooth generator forms the basis for the other oscillators. This ensures that oscillators with the same frequency stay in sync ad infinitum.

Calculating the addition value is done automatically during the setup of an oscillator, based on the input of the frequency as a floating number.

The following code-example creates a saw-tooth generator named **OSC1** with a frequency of 261.63 Hz (middle key piano:

```
261.63e ~SAW osc1
```

The word **~SAW** creates a block consisting of 8 32b cells in the WAVE-table containing the following memory locations/cells:

1. a link-address to the next block
2. the WAVE object-ID - in the case of **~SAW** id=9
3. memory cell for the float output with float value **0.0**e
4. memory cell for the integer output with value **0**
5. an addition-value of 25480536, calculated by the word **~Hz**, resulting in a frequency of 261.63 Hz
6. a phase-field with value **0**
7. an external address-field with value **0**
8. a field to store the total after each tick with value **0**

Finally **~SAW** creates a word, in the example with the name **osc1**. When called, this word pushes the address of the float output-field on the stack. Thus **osc1** forms the reference to the newly created oscillator.

The following table shows the complete 8 cell block created for a saw-tooth oscillator.

The values in grey should not be changed by the user. Changes in the header might crash **core2**, requiring a power-cycle. This 2 cell header is not shown in other versions of this table in this document.

The values in green can be changed by the user using **! (STORE)**, or read with **@ (FETCH)**. Changing can be done at any time, also when WAVE is actively generating sound. For instance if you change the value in the field 'addition value', the frequency of the oscillator will change realtime.

The values in yellow are the outputs. They are updated 44100 times per second when WAVE is active. These values can be changed by the user, although this is usually a useless action.

offset in bytes	field	comments
--	**link-address**	address of next object - if address=0 -> no further objects
--	**WAVE object-ID**	identifies the object described in the block
0	**float output**	address where the output-signal is available as a signed 32bit floating number between -1.0 and 1.0 - this is the main ouput
+4	**integer output**	address of the signed 32bit integer output
+8	**addition value**	this value is added each loop to the total at offset +20 - signed 32b integer
+12	**phase**	allows shifting the phase - 0=no effect - signed 32b integer

offset in bytes	field	comments
+16	external source address	if this field is not 0, then the value at the address is added to the total instead of the addition-factor at offset +8 - this way the output of one sound-element can influence the frequency of another oscillator - 32b address
+20	total	cell where the total is saved after each loop - signed 32b integer

The complete control-block of a sawtooth oscillator

sawtooth oscillators

There are 3 sawtooth oscillators. The sawtooth, the reverse sawtooth and the triangle shaped wave.

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

The reverse sawtooth also produces a sawtooth formed wave but with a reversed profile: it ascends steeply to the maximum and then slowly descends to minimum. In other words, the top of the wave is at the start. It sounds the same as the base sawtooth, its main use is as input to a Voltage Controlled Volume Control (VCVC) to influence volume.

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

~SAW (add_v --): Creates a sawtooth oscillator, oscillating at frequency *f*, and defines a word which returns the address of the float-output.

offset	field	description
0	float output	output between -1.0 and 1.0
+4	integer output	output=(total+phase) - signed 32bit integer
+8	addition value	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer
+12	phase	values other than 0 shift the phase up to +/- 180 degrees - signed 32b integer

offset	field	description
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer

control-block of base sawtooth oscillator

~REVSAW (add_v --): Creates a reversed sawtooth oscillator, oscillating at frequency governed by add_v, and defines a word which returns the address of the float-output.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer
+8	**addition value**	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer
+12	**phase**	values other than 0 shift the phase up to +/- 180 degrees - signed 32b integer
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer

control-block of reversed sawtooth oscillator

~TRIA (F_fl top_fl --): Creates a triangle-shaped wave oscillator, oscillating at frequency governed by add_v and with the top located 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 -> **50e ~%** will put the top in the middle. ~TRIA also defines a word which returns the address of the float-output.

For example:

```
440e 50e ~% ~tria   tri1   \ create a triangle-oscillator
  tri1 tri1 ~output out1   \ create output element
```

produces a sawtooth oscillator with the top at 50% of the width of the wave, and an output element with the just created triangle-wave oscillator (**tri1**) as input in both of its inputs.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer
+8	**addition value**	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer
+12	**phase**	values other than 0 shift the phase up to +/- 180 degrees - signed 32b integer
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer
+24	**position of top**	holds the position of the top - 0=50% - signed 32b integer
+28	**float up value**	value used in the up-phase of the generated wave - calculated in wabiPi4
+32	**float down value**	value used in the down-phase of the generated wave - calculated in wabiPi4

control-block of triangle oscillator

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 unpleasant at any frequency and outright nasty at low frequencies. But the sound is very recognisable, as it was common on early home-computers. The classic sound-generator ay-3-8190 produced block-waves. A 3-6 dB low-pass filter was usually added to get rid of some of the nastiest harmonics. 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_fl --): Creates the most basic of block oscillators, with a fixed 50/50 ratio and oscillating at a frequency governed by the F as floating point number. Also defines a word which returns the address of the float-output.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer
+8	**addition value**	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer
+12	**phase**	values other than 0 shift the phase up to +/- 180 degrees - signed 32b integer
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer

control-block of base block oscillator

~UBLOCK (F_fl ratio_fl --): Creates a block oscillator with a user-definable ratio, oscillating at a frequency governed by F in floating point. Also defines a word which returns the address of the float output-field.
The ratio is a signed integer number between minimum and maximum integer value. Zero gives a ratio of 50%. The '**~%**' word can be used, in which case a percentage as floating point number is expected.

440e 20e ~% ~ublock osc1

Gives an oscillator with a 20/80 ratio and a frequency of 440 Hz.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer
+8	**addition value**	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer
+12	**phase**	values other than 0 shift the phase up to +/- 180 degrees - signed 32b integer
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address

offset	field	description
+20	**ratio**	percentage of time the oscillator outputs max-int - signed 32b integer - 0=50%
+24	**total**	holds total after each loop - signed 32b integer

control-block of universal block oscillator

sine oscillator

There is 1 sine generator as there is only 1 sine. It produces a mellow sound, which some find a bit boring.

~SINE (F_fl --): Creates a sine-oscillator, oscillating at a frequency controlled by F (floating-point number), and defines a word which returns the address of the float output-field.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer
+8	**addition value**	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer
+12	**phase**	values other than 0 shift the phase up/down to +/- 180 degrees - signed 32b integer
+16	**external source address**	if not 0 -> use value at external_source_address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer

control-block of sine oscillator

Glissando effect

It is possible to add a glissando effect to an oscillator. This done by adding a glissando element to an oscillator. Adding a glissando element to an oscillator points the external source address of the oscillator to the output field of the glissando element.

~GLISSANDO (addr_osc F_fl --): Creates a glissando element and adds the effect to an existing oscillator. The reduction factor determines how fast or slow the glissando effect is. A reduction factor of .999e is clearly audible and a good starting point. F-present is filled with the existing f of the oscillator.

440e ~sine osc1

osc1 0.999e ~glissando gli

offset	field	description
0	**integer output**	integer output is the add_value to be used by the oscillator
+4	**address input**	address pointing to add-factor of oscillator
+8	**reduction factor**	reduction-factor - float
+12	**f_present**	f_present is kept a a float add_factor - f_present is updated every tick

control-block of glissando element

Noise generators

There are four noise generators. They each produce slightly different sounding noise, although the difference between the different sounds produced is not that noticeable. If you want to really soften the sound, placing a filter after the noise-generator is more effective.
Three noise-generators are based on the XORshift randomisation method devised by Prof. Marsaglia. One noise-generator is based on the 17 bit random generator of the noise-generator in the **AY-3-8910 Programmable Sound Generator.** An IC much used in the 80s, in both home-computers and arcade-games in that time.

In **WAVE** the random-generators for noise are independent of the random number generators in WabiPi4.

~FNOISE (**add_v** --): produces a noise which changes the output value randomly at the given frequency governed by add_v. This frequency might be noticeable. This generator has the same common features as the other oscillators. So the frequency can 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.
Also defines a word which returns the address of the float-output.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer - **MUST be non-zero otherwise no noise is generated**
+8	**addition value**	on external_source_address=0 add each loop to the total at offset +20 - signed 32b integer

+12	**phase**	values other than 0 shift the phase up to +/- 180 degrees - signed 32b integer
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer

control-block of frequency noise generator

~BNOISE (cycle --): produces a basic noise which randomly changes the output from max-integer to min-integer or vv. at the specified interval of cycles. Also defines a word which returns the address of the float output-field.
The sound of this generator is as harsh as it gets.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output as signed 32bit integer
+8	**skip**	defines the number of cycles to be skipped - 0=generate a new value each cycle - 1=generate a new value each second second, etc.
+12	**total**	counts number of cycles skipped
+16	**seed**	seed for random generator - MUST by non-zero

control-block of basic noise generator

~SNOISE (cycle --): produces a slightly softer noise, as it changes output to a random value at the specified interval of cycles. Also defines a word which returns the address of the float-output.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output as signed 32bit integer
+8	**skip**	defines the number of cycles to be skipped - 0=generate a new value each cycle - 1=generate a new value each second cycle, etc.
+12	**total**	tracks number of cycles skipped
+16	**seed**	seed for random generator - must by non-zero

control-block of soft noise generator

~8910NOISE (*div* --): produces the same noise as the **AY-3-8910.** It produces an output which changes values from -1.0e and 1.0 randomly at a frequency resulting from the following calculation: f=(1000000/16)/div. In the original AY-3-8910 the divisor could have a value from 1 to 31, where the values 1 and 2 resulted in ultrasonic sound. This version accepts values between 3 and 255.
This generator has the same common features as the other wave generating oscillators. So the frequency can be controlled by an external source and phase-control is available. As this generator produces block-waves, phase control acts as a level-shifter rather than shifting the phase.
~8910NOISE also defines a word which returns the address of the float output.

offset	field	description
0	**float output**	output between -1.0 and 1.0
+4	**integer output**	output=(total+phase) - signed 32bit integer
+8	**addition value**	on external_source_address=0: add addition_value each loop to the total at offset +20 - signed 32b integer - calculating an authentic AY-3-8910 frequency can be done with **8910DIV->A**
+12	**phase**	values other than 0 shift the level - signed 32b integer - probably useless
+16	**external source address**	if not 0 -> use value at external source address as addition-factor - 32b address
+20	**total**	holds total after each loop - signed 32b integer
+24	**seed**	holds the seed between loops - as this noise generator is based on a linear feedback shift register random generator, 0 *is* acceptable as value.

control-block of 8910 noise generator

8910DIV->A (div -- addition-value): calculates the correct addition value for a divisor. Useful if you want to change the frequency of this generator to an authentic value. values 3 to 31 create an authentic result. Values up to 255 can be used. The values 1 and 2 on the 8910 resulted in ultrasonic sound, which cannot be re-created with a 44.1 kHz sampling sound-generator.

Example: a **8910noise** generator with a change of output at 3125 Hz:

```
20 ~8910noise ns1
```

The following code

```
10 8910div->a ns1 8 + !
```

changes the random generation frequency from 3125 to 6250 Hz

ADSR

The ADSR was developed by Vladimir Ussachevsky in 1965 for the mini MOOG. ADSR stands for Attack, Decay, Sustain, Release. It was based on the observation that a lot of analog instruments produce a sound which swells to maximum fast, than decays pretty fast to a sound-level where they stay a bit, followed by a slower release to a sound-level of 0.

The **WAVE** implementation is a universal ADSR, where the length of each of the 4 periods and the level of the sustain phase are user-defined. It is possible to specify a zero length for any of the periods. This way any of the 4 periods can be skipped. So, for instance, if you have no need for the sustain-phase, just specify zero as the length. Another feature of the **WAVE-ADSR** is that it can create a one-shot or repeating pattern.

Voltage Controlled Oscillator - VCO

VCOs are traditional elements of synthesisers. In **WAVE**, any oscillator can be turned into a voltage controlled oscillator by preceding it by the VCO element, and connecting them together.

There are four different VCO-types. A full range VCO, a range-limited VCO and two VCOs which only produce correct notes, where one VCO produces all 12 notes in an octave (ie full and half notes), and the other produces the 7 whole notes.

For the two note-based VCOs, the user can specify for the VCO what the lowest and the highest notes are they generate. The range of notes is extensive, 204 notes and half-notes, ranging from **sub sub sub sub sub sub sub sub contra C** (C min 7) to 6-striped (B9). A range of 17 octaves. That is from a frequency of 0.127775 Hz (C minus 7) to 15804.2 Hz (B9). For musically ignorami like me: a normal piano has 88 keys.

In WAVE notes are numbered 0 to 223. Note number 0 = C-7, note 223 = B9. Some examples: note 132 is C4, 141 is A4, the middle key on a piano keyboard. The lowest key on a 88 key piano (A0) is number 93, and the highest note on a piano is 180 in **WAVE.**

The relation between WAVE note-numbers and Midi note numbers is straightforward. WAVE note number are 72 higher than Midi note numbers. In other words Midi note number 0 is note 72 in WAVE.

~VCO (ch0 --): uses the value of ch0 to generate frequencies in the range of 20-12000 Hz. Where -1.0e gives 20Hz, 0.0e gives 1520Hz and 1.0e gives 12020Hz.

offset	field	description
0	**integer output**	the integer output is the add_value as used in the normal oscillators
+4	**address input**	address pointing to input channel

control-block of ~VCO

~RNGVCO (ch0 f_low_fl f_high_fl --): uses the value from ch0 to generate frequencies between f_low and f_high (both as floating point number). In other words, -1.0v results in frequency of f_low, and +1.0v results in in a frequency of f_high.
The first frequency must be the lower of the two frequencies. Very low frequencies (below 30 Hz or so) are not very accurate.

offset	field	description
0	**integer output**	the integer output is the add_value as used in the normal oscillators
+4	**address input**	address pointing to input channel
+8	**offset**	float number - calculated during setup of the element
+12	**multiplication value**	float number - calculated during setup of the element

control-block of ~RNGVCO

~VCO12 (ch0 start_note range_of_notes --): uses the value of the input from ch0 to generate one of range_of_note, starting at start_note. As example: if the start_note has the value of 84, and the range is 4, than the VCO can produce 4 different notes, namely notes 84, 85, 86 or 87, depending on the value of the input channel.

offset	field	description
0	**integer output**	the integer output is the add_value as used in the normal oscillators
+4	**address input**	address pointing to input channel
+8	**lowest note to be generated**	integer - number of lowest note to be sounded
+12	**step-value**	float number - calculated during setup of the element

control-block of ~VCO12

~VCO7 (ch0 start_note range_of_notes --): uses the value of the input from ch0 to generate one of the range_of_notes, starting at start_note. The start_note must be a whole note. If not, then **~VCO7** take the next higher note.
Example: if the start_note is 132, and the range is 4, then the **~VCO7** can produce 4 different notes, namely notes 132, 134, 136 and 137. The notes in between are half-notes and so will not be played by **~VCO7**

If the start note is <<< #TBD complete note-numbering

offset	field	description
0	**integer output**	the integer output is the add_value as used in the normal oscillators
+4	**address input**	address pointing to input channel
+8	**lowest note to be generated**	integer - number of lowest note to be sounded
+12	**step-value**	float number - calculated during setup of the element

control-block of ~VCO7

Delay buffers

Delay buffers form the basis for filters, chorus-effecten and echo-like effects. WAVE has four different types: a universal delay, 2 fixed length delays, and a fractional delay. All do basically the same: store a value or values for later use.
The universal delay can delay as many cycles as wanted, and can be used for anything like filters, chorus and echo-effects. The 1 cycle and 5 cycle delays are mainly useful in filter design. And finally the fractional delay can be useful when the length of a delay buffer should be something different than an integer number.

~DELAY (input *addr length -- output): Uses the buffer starting at *addr to effectuate a delay of the input for 'length' cycles.

The user has to define a buffer before using **~DELAY**, and has to make sure that the buffer has enough capacity. The length can be changed to change the delay-time, also when WAVE is active. The user has to make sure that the buffer has enough capacity for the new value.

offset	field	description
0	**output**	output
+4	**address input**	address pointing to input channel
+8	**length delay**	the delay in number of cycles - unsigned 32b integer
+12	**start address buffer**	start-address of buffer
+16	**index**	pointer to last word written

control-block of ~DELAY

~1CDELAY (input -- output): delays the output for 1 cycle

offset	field	description
0	**output**	output
+4	**address input**	address pointing to input channel - address
+8	**buffer**	buffer to store 1 value

~5CDELAY (input -- output): delays the output for 5 cycles. This delay is often used in comb-filters. But the effect seems minimal.

offset	field	description
0	**output**	output
+4	**address input**	address pointing to input channel - address
+8	**buffer 1**	buffer to store 1 value
+12	**buffer 2**	buffer to store 1 value
+16	**buffer 3**	buffer to store 1 value
+20	**buffer 4**	buffer to store 1 value
+24	**buffer 5**	buffer to store 1 value
+28	**index**	pointer to the next buffer-cell

~FRACDELAY (input *addr length add_v -- output): The length of a buffer is obviously specified as an integer value. But for some applications a buffer with a fractional length would be a nice_to_have. For instance in the Karplus-Strong algorithm (see next chapter) the frequency created depends on the length of the delay-buffer. Only with a fractional buffer is it possible to created correctly tuned tones.

A real fractional buffer does not exist. But it is possible to emulate a buffer which has more or less the same effect. This is done by alternating between two different almost equal lengths of the buffer, thus effectually having a fractional length. For instance if you have a buffer with a length of 30 cells, you could use 29 cells for the first delay-phase, and then switch to 30 cells for the second delay-phase. The result would be a buffer with an effective length of 29.5 cells. Which is fractional.

In WAVE the fractional buffer is controlled by an internal saw-tooth oscillator. The buffer is effectively as long as the length of the wave for each wave-cycle. Also if the average length of the wave is not an integer number. This way any complicated calculations are unnecessary.

offset	field	description
0	**output**	output
+4	**address input**	address pointing to input channel - address
+8	**add_value**	controls the frequency of the underlying sawtooth oscillator
+12	**base-address buffer**	points to the start of the delay-buffer
+16	**max length delay**	the maximum length the delay-buffer - used by the system to avoid writing outside of the buffer
+20	**total**	stores the total from tick to tick
+24	**index**	points to the next cell in the buffer where a value is read and a value is written
+28	**value index of last wrap_to_zero**	keeps the last used length of the buffer

control-block of ~FRACDELAY

Volume control

A sound generator obviously must be able to control volume. WAVE offers both fixed volume-control, using **~AMP**, and two different types of voltage controlled volume control, namely **~FVOL** and **~VCVC**. **~AMP** takes an input channel and multiplies that with a fixed factor. If this factor is below 1, the output of the input channel is lowered, if the factor is higher than 1, the output is raised.
~VCVC and **~FVOL** take the signal of the input channel and control the level of the output by the signal-level of the control channel. **~VCVC** sets the level of the output between 0 to 100%. **~FVOL** sets the level of the output between -100 and 100%. In other words: a negative controlling voltage reverse the output signal. Mostly useless but maybe nice to experiment with.

~AMP (ch0 level_fl -- *output): the level is an integer which is internally converted to a floating point number by dividing the number by a 1000.
~AMP can be used to amplify or to reduce the level of channel 0. Values below 1.0e reduce the output and above 1.0e amplify the output. The conversion from input-level to output-level is a simple multiplication.

The following example reduces the output of osc1 by 50%:

```
          440e ~sine   osc1
    osc1 0.5e ~amp    soft1
soft1 soft1 ~output outp
```

offset	field	description
0	**output**	output
+4	**address input**	address pointing to input channel - address
+8	**level**	float level - controls the reduction or amplification of the output

control-block of ~AMP

The following example raises the output of osc1 with 120%. Please note that clipping effects will be noticed, as the output element always clips the signal back to -100 to 100%. The audible sound is more a square wave than a sine. This can be used to create more or less harmonics.

```
          440e ~sine   osc1
    osc1 2.2e ~amp    loud1
loud1 loud1 ~output outp
```

~VCVC (ch0 control_channel -- *output): uses the level of the control_channel to set the level of channel 0 and puts that in the output. A level of -1.0e of the control_channel results in an output-level of 0% (ie: silence). A level of 1.0e of the

control-channel results in an output-level of 100% (ie: the level of the output is the same as the level of channel 0).

~FVOL (ch0 control_channel -- *output): uses the level of the control_channel to set the level of channel 0 and puts that in the output. A level of -1.0e of the control_channel results in an output-level of -100% (ie: reversed polarity). A level of 0.0e volt results in 0% (ie. silence) A level of 1.0e of the control-channel results in an output-level of 100% (ie: the level of the output is the same as the level of channel 0).

Filters:

Filtering is an important feature for any decent sound-generation system. The WAVE system contains 3 static filters (SF) with a 12dB slope and 3 voltage controlled filters (VCF), one with a 12 and two with a 24dB slope. The filters are coded as floating point primitives using the NEON-coprocessor in the ARM processor, guaranteeing optimal performance.

BIQUAD FILTER << WORK IN PROGRESS

A BiQuad filter is very clever algorithm. It takes only 15 opcodes to program it in NEON-assembly. Of those 15 opcodes, 6 opcodes are needed for loading and storing values and 1 is needed as subroutine-return. Meaning that the actual calculations only require 8 opcodes! Yet it can function as many different types of filter, like low-pass, high-pass, band-pass, band-reject and all-pass (for phase control). And for each of the filter-types it also handles different Qs and gain. This all by setting 5 filter coefficients.
Formally the BiQuad filter is a second-order IIR digital filter. WabiPi4 implements the transposed-direct-form-II version.

WAVE can calculate coefficients for 5 BiQuad filter-types: low pass, bandpass, high pass, band-reject and all-pass.
For each of these functions the user can define the cut-off frequency and the Q of the filter. The amount of gain/loss of a filter will be added in future.

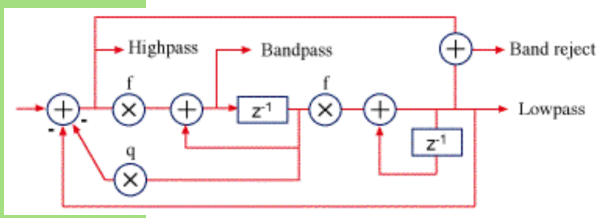
The biggest disadvantage of a BiQuad filter is that there is a lowest cutoff frequency it can handle. The implementation in WAVE can handle cutoff frequencies down to around 300Hz, depending somewhat on settings for Q and gain and the input-signal. Specifying lower frequencies can result in an unstable filter, resulting in a resonating filter. In other words, the filter has become an oscillator. WAVE does not check for a lower boundary, as it could well be that you want to have an oscillating filter.

WAVE also implements the state variable filter which handles all cutoff frequencies below **14 kHz.**

Calculating the 5 different filter-coefficients is relatively complex. A support routine for this is under development. Goal of this routine is that the only input needed is: type of filter, the cutoff frequency, the gain and the Q. If you do not know what to use for Q use 0.707, a often used value. WAVE than calculates the 5 coefficients.

STATE VARIABLE FILTER

A **State Variable Filter** is equally clever as the BiQuad algorithm. With 29 opcodes it is around 2 times longer compared to a BiQuad filter. But it calculates several filter-types, like low-pass, high-pass and band-pass, all at the same time.



The **State Variable Filter** is a direct digital conversion of a analog filter design often used in synthesisers in the 70s. WAVE implements two versions: the optimised setup as described by Andy Simper of **Cytomic** in a technical paper in 2016 (many thanks for that excellent paper!). And a classic implementation which has a lower limit for the cut-off frequency, but which is slightly more accurate and somewhat more suitable as resonating filter.

A disadvantage of the state variable filter is that there is a maximum cutoff frequency it can handle. The optimised version described by Mr. Simper is less sensitive to this. The version used in WAVE can handle cutoff frequencies to around 14 kHz, depending somewhat on input signal and selected Q. The resonating version can handle cut-off frequencies up to around 7kHz.

Two values are needed to define and setup a state variable filter: the cutoff frequency and the Q. These 2 inputs are used to pre-calculate and store 4 constants. This speeds up the calculations for every tick. The third input is the sound channel to be filtered.

If a change in settings is needed, for instance a different cutoff frequency, than the word **~4COEFSVF** can be used. It calculates the 4 coefficients based on a given cutoff frequency and Q. The 4 coefficients can than be stored into the control-block

~SVFILTER (ch0 Fc_fl Q_fl --): ~svfilter creates a state variable filter which filters the signal from ch0, with a cutoff frequency of fc (as float), and with a given quality-factor (Q) (as float). The Q is the reverse of the damping factor, so a filter with a high Q has a higher output (ie resonates) around the cutoff frequency.

~SVFILTER Also creates a word with the string directly after **~SVFILTER** which returns the address of the first output filed. Q is normally within the range of 0.1e-10e. Too high a Q-value can result in an unstable filter, but this can be a wanted effect.

The SVFILTER has 6 output fields. The low-pass is the most used, but others are useful for instance when simulating specific analog instruments.

Apart from the address, all the fields in the block are calculated during setup or by the algorithm for each tick. If another setting is wanted for a filter, then the 4 coefficients (k, a1, a2, a3) need to be calculated anew.

offset	field	description
0	**float low-pass**	output between -1.0 and 1.0
+4	**float band-pass**	output between -1.0 and 1.0
+8	**float high-pass**	output between -1.0 and 1.0
+12	**float band-reject**	output between -1.0 and 1.0
+16	**float peak-pass**	output between -1.0 and 1.0
+20	**float all-pass**	output between -1.0 and 1.0
+24	**source address**	32b pointer to output-field of another WAVE-element like an oscillator or mixer
+28	**K-constant**	float - calculated once during setup
+32	**A1-constant**	float - calculated once during setup
+36	**A2-constant**	float - calculated once during setup
+40	**A3-constant**	float - calculated once during setup
+44	**delay buffer 1**	storage for the first 1 cycle delay buffer
+48	**delay buffer 2**	storage for the second 1 cycle delay buffer

control-block of 12 dB state variable filter

~4COEFSVF (fc Q -- a3 a2 a1 k): Calculates the 4 coefficients voor the State Variable Filter and puts them in reverse order on stack. This way they can quickly be written into the control-block. Use **~4COEFSVF** when a change in settings is needed for an existing filter, for instance a change in the cutoff frequency.

Please note that **~4COEFSVF** cannot be used for the resonating filter as the calculations for the 4 coefficients for that filter are very different.

~RESFILTER (ch0 fc_fl Q_fl --): **~resfilter** creates a state variable filter which filters the signal from ch0, with a cutoff frequency of fc (float), and with a given quality-factor (Q) (float). The Q is the reverse of the damping factor, so a filter with a high Q has a higher output, ie resonates, around the cutoff frequency. All the state-variable filters do that, it is just that this specific implementation is a bit more accurate and thus slightly better suitable for resonation when using a high Q.
~RESFILTER Also creates a word with the string directly after **~RESFILTER**, which returns the address of the first output filed.

Q is normally within the range of 0.1e-10e, but a Q of 100e or even higher can be used if resonation if wanted.

Please note that **~RESFILTER** has 4 output fields. Compared to the **~SVFILTER**, the output-fields for peak and all-pass filters are missing. Due to that the control blocks of **~SVFILTER** and **~RESFILTER** have different lengths. This is especially relevant when changing the address of the source-channel.

Apart from the address, all the fields in the block are calculated during setup or by the algorithm for each tick. If another setting is wanted for a filter, then the 4 coefficients (k, g0, g1, g2) need to be calculated anew.

offset	field	description
0	**float low-pass**	output between -1.0 and 1.0
+4	**float band-pass**	output between -1.0 and 1.0
+8	**float high-pass**	output between -1.0 and 1.0
+12	**float band-reject**	output between -1.0 and 1.0
+16	**source address**	32b pointer to output-field of another WAVE-element like an oscillator or mixer
+20	**K-constant**	float - calculated once during setup
+24	**G0-constant**	float - calculated once during setup
+28	**G1-constant**	float - calculated once during setup
+32	**G2-constant**	float - calculated once during setup
+36	**delay buffer 1**	storage for the first 1 cycle delay buffer
+40	**delay buffer 2**	storage for the second 1 cycle delay buffer

control-block of 12 dB resonating variable filter

Voltage controlled filter

At least part of the succes of the miniMoog, the famous synthesiser introduced in 1965 can be explained by two factors: it introduced two new features: it had the first ADSR, and it had a revolutionary 24 dB Voltage Controlled Filter. These two opened up many new possibilities and enabled generating sounds which were pleasant to listen to.

There are 3 Voltage controlled filters available in WAVE. One 12 dB and two 24 dB filters. They can be seen as an attempt to emulate the functionality of the MOOG filters.
All 3 VCFs are complex pieces of NEON assembly. The **12 dB VCF filter** is 59 opcodes long and consumes 187 cycles per tick. The **low-pass only 24dB VCF** is 67 opcodes long and consumes 206 cycles. And finally the full **24 dB VCF filter** is 118 opcodes

long and consumes 248 cycles per tick. And with that is by far is the biggest and slowest of all WAVE-blocks.

The way the VCFs work

All 3 VCFs have 3 inputs: the input channel which receives the signal to be filtered. An input channel which controls the cutoff frequency, and an input channel which controls the Q of the VCF.

The 12dB VCF and the full 24dB produce 4 output channels: low-pass, band-pass, high-pass and band-reject. The output of the low-pass only VCF is self-explanatory.

The cutoff frequency (fc) can be changed in a range of 40Hz to 12kHz. An input signal of -1.0v results in a fc of 40Hz, an input signal of 1.0v results in a fc of 12kHz.

The Q (fc) can be changed in a range of 0.1e to 8.1e. An input signal of -1.0v results in a Q of 0.1e, an input signal of 1.0v results in a Q of 8.1e.

~12dBVCF (ch0 ch_fc ch_Q --): filters the signal from ch0 with a 12dB slope, controlled by the inputs on ch_fc and ch_Q. Creates 4 outputs: low-pass, band-pass, high-pass and band-reject. The cut-off frequency has a range of 40Hz to 12kHz for values of -1.0e to 1.0e on the input-channel. The Q has a range of 0.1e to 8.1e for values of -1.0e to 1.0e on the input-channel. High Qs can result in resonation.

offset	field	description
0	**float low-pass**	output between -1.0 and 1.0
+4	**float band-pass**	output between -1.0 and 1.0
+8	**float high-pass**	output between -1.0 and 1.0
+12	**float band-reject**	output between -1.0 and 1.0
+16	**source address**	32b pointer to output-field of another WAVE-element like an oscillator or mixer
+20	**source address FC**	32b pointer to output-field of another WAVE-element - controls the cut-off frequency from 40Hz to 12kHz
+24	**source address Q**	32b pointer to output-field of another WAVE-element - controls the Q of the filter from 0.1e to 8.0e
+28	**delay buffer 1**	storage for the first 1 cycle delay buffer
+32	**delay buffer 2**	storage for the second 1 cycle delay buffer

control-block of 12 dB voltage controlled filter

~MOOGVCF (ch0 ch_fc ch_Q --): filters the signal from ch0 with a 24dB slope, controlled by the inputs on ch_fc and ch_Q. Only creates a low-pass output. The cut-off frequency has a range of 40Hz to 12kHz for values of -1.0 to 1.0 on the input-channel. The Q has a range of 0.1 to 8.1 for values of -1.0 to 1.0 on the input-channel. High Qs can result in resonation.
This filter comes closest to the original filter in the MiniMoog, but does certainly not reach the same quality.

offset	field	description
0	**float low-pass**	output between -1.0 and 1.0
+4	**source address**	32b pointer to output-field of another WAVE-element like an oscillator or mixer
+8	**source address FC**	32b pointer to output-field of another WAVE-element - controls the cut-off frequency from 40Hz to 12kHz
+12	**source address Q**	32b pointer to output-field of another WAVE-element - controls the Q of the filter from 0.1e to 8.0e
+16	**delay buffer 1**	storage for the 1st 1 cycle delay buffer
+20	**delay buffer 2**	storage for the 2nd 1 cycle delay buffer
+24	**delay buffer 3**	storage for the 3rd 1 cycle delay buffer
+28	**delay buffer 4**	storage for the 4th 1 cycle delay buffer

control-block of 24 dB low-pass only voltage controlled MOOG filter

~24dBVCF (ch0 ch_fc ch_Q --): filters the signal from ch0 with a 24dB slope, controlled by the inputs on ch_fc and ch_Q. Creates 4 outputs: low-pass, band-pass, high-pass and band-reject. The cut-off frequency has a range of 40Hz to 12kHz for values of -1.0 to 1.0 on the input-channel. The Q has a range of 0.1 to 8.1 for values of -1.0 to 1.0 on the input-channel. High Qs can result in resonation.

offset	field	description
0	**float low-pass**	output between -1.0 and 1.0
+4	**float band-pass**	output between -1.0 and 1.0
+8	**float high-pass**	output between -1.0 and 1.0
+12	**float band-reject**	output between -1.0 and 1.0

+16	**source address**	32b pointer to output-field of another WAVE-element like an oscillator or mixer
+20	**source address FC**	32b pointer to output-field of another WAVE-element - controls the cut-off frequency from 40Hz to 12kHz
+24	**source address Q**	32b pointer to output-field of another WAVE-element - controls the Q of the filter from 0.1e to 8.0e
+28	**delay buffer 1**	1st 1 cycle delay buffer
+32	**delay buffer 2**	2nd 1 cycle delay buffer
+36	**delay buffer 3**	3rd 1 cycle delay buffer
+40	**delay buffer 4**	4th 1 cycle delay buffer
+44	**delay buffer 5**	5th 1 cycle delay buffer
+48	**delay buffer 6**	6th 1 cycle delay buffer
+52	**delay buffer 7**	7th 1 cycle delay buffer
+56	**delay buffer 8**	8th 1 cycle delay buffer
+60	**delay buffer 9**	9th 1 cycle delay buffer
+64	**delay buffer 10**	10th 1 cycle delay buffer

control-block of 24 dB full voltage controlled filter

Keeping track of WAVE

Keeping an overview of WAVE is not always easy, especially with a more complex setup of WAVE-elements. The following words give an overview of the WAVE-list, used and free CPU-cycles, size of the WAVE-list or show an overview of an individual WAVE-element.

~SZLIST (-- n): returns the total size of the WAVE-list in bytes. The actual maximum size presently is 64kB, twice the size of the L1 cache. An abort will occur if the user tries to fill the WAVE-list beyond the 64kB limit.

~CUSED (-- n): returns the number of CPU-cycles used for the last completed tick (=sample-loop). The number of cycles is updated after each tick. Returns 0 when WAVE is not active. Returns 8 when WAVE is active with an empty WAVE-list.

Please note that the value returned is the actual number of CPU cycles used for calculating the last loop. The state of the L1 and L2 caches, the state of the memory system and the state-engine of core2 and the state of the key-input system influence the value returned.

For instance when WAVE is active with an empty list, **~CUSED** can return values higher than the default of 8 cycles. And calling **~CUSED** repeatedly can return different values each time. This reflects the reality of modern CPUs. On average they are fast, but have a semi non-deterministic timing at single low level events.

~CFREE (-- n): returns the free capacity for calculation in CPU-cycles. The maximum number of free cycles returned is 40720 cycles.

.WVC (--): prints the value returned by **~CUSED**. Saves me typing 4 characters, nothing more.

.~LIST (--): prints an overview of the complete WAVE list and shows if WAVE is presently active or not.

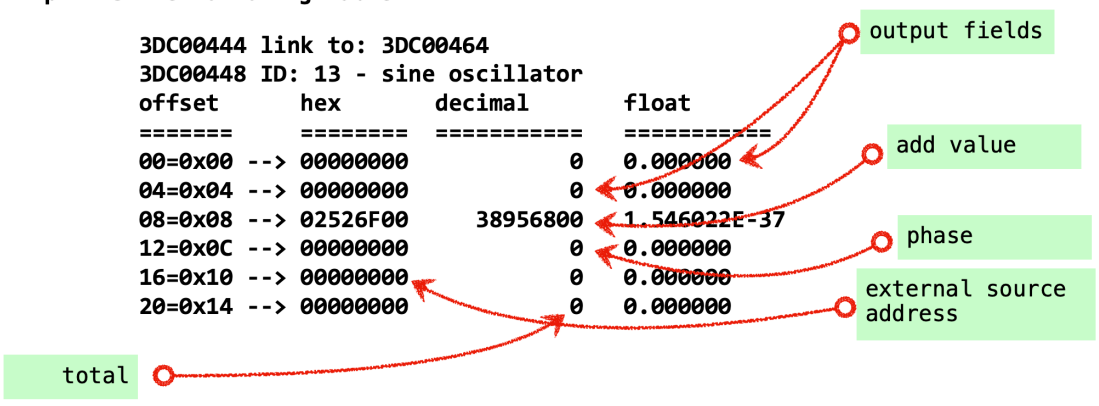
.~BLOCK (address --): prints an overview of the block of an individual WAVE-element, including the header. Needs the address of the first outfield as input, this address is returned by the word created by the WAVE-element. If WAVE is active, shows the actually values.
.~BLOCK is an atomic operation, all values shown are from the same tick (=sample-loop).
You will need to consult the programmers guide to interpret the individual fields, especially as all values are always printed thrice, as hex, decimal and float.

Example:

```
~reset              \ resets WAVE and clears the table
400 ~sine osc1      \ creates a sine-oscillator with name OSC1

osc1 .~block        \ print an overview of the sine oscillator
```

prints the following table:

```
3DC00444 link to: 3DC00464
3DC00448 ID: 13 - sine oscillator
offset      hex       decimal       float
=======     ========  ===========   ===========
00=0x00 --> 00000000            0   0.000000
04=0x04 --> 00000000            0   0.000000
08=0x08 --> 02526F00     38956800   1.546022E-37
12=0x0C --> 00000000            0   0.000000
16=0x10 --> 00000000            0   0.000000
20=0x14 --> 00000000            0   0.000000
```



WAVE v2.1 – example code

This chapter contains **examples of code for WAVE 2.1.** It is a mix of sounds and things I learned while testing WAVE. Use it to get a bit of an idea how to use the words, or as inspiration. Feel free to change values. Experimentation is the only way of getting a feel for the possibilities.

The 2 channel voltage-controlled mixer allows mixing of two input channels in different ratios, where the ratio of the two input channels is controlled by an external voltage. And thus the ratio can be controlled by another oscillator. The example emulates a steam-engine driven locomotive at half speed.

```
\ steam locomotive
\ resources used: 177 CPU cycles - 256 bytes

~reset               5 ~snoise     osc1
              4e 5e ~% ~tria       mod1
      osc1 500e 0.707e ~svfilter   fil1
     fil1 4+ fil1 mod1 ~2chvcmixer mix1
         mix1 1000e 3e ~svfilter   fil2
             fil2 0.8e ~amp        amp1
             amp1 amp1 ~output     outp
~start
```

Ever wondered **how a guitar is simulated** with a synthesiser? One of the ways is using the **Karplus-Strong algorithm.** It was published in 1983 and first licensed to Mattel-Electronics. There are enough articles on the internet to read up on it. It is included here as it shows nicely how a feedback loop is implemented, and it creates a nice sound.

The algorithm functions as follow: to start the process, a delay-buffer is filled with a short pulse of random noise. The output is then filtered by splitting it in two, and delaying one of these with 1 cycle before combining them again. This signal is then fed back into the loop. After each noise pulse, a guitar-like sound is produced. The length of the buffer defines the pitch of the created sound. And by using other filter-setups, different effects can be created.

```
\ Karplus-Strong guitar emulator
\ resources used: 144 CPU cycles - 220 bytes

create DLYBF 2000 cells allot   dlybf 2000 0 fill

~reset     10000e ~fnoise   noise1
          1e 1e ~% ~ublock   pulse1   \ a short pulse every sec
    noise1 pulse1 ~vcvc     noisepulse
 ~nill noisepulse ~2chmixer mix1     \ the cell pointing to
                                     \ ~nill is filled later
   mix1 dlybf 150 ~delay    dly1
             dly1 ~1cdelay  1cdly1
      dly1 1cdly1 ~adder    mix2
      mix2 0.998e ~amp      vol1     \ lowers the signal slowly
       dly1 ~nill ~output   outp
```

```
vol1 mix1 4+ !    \ << exchange the ~nill with a...
~start            \ ...pointer to the output of vol1
```

Creating an echo with a delay buffer. The code below creates an echoing ping. It shows the minimum elements needed.

1. an echo needs a buffer – here the buffer is 44100 elements long – buffers can be too long, but not too short
2. the buffer must be cleared before use for a clear echo
3. the feedback loop must be patched in to create the feedback loop
4. an echo must have a way to die out

```
\ echo
\ resources used: 147c CPU cycles - 192 bytes

create ECHOBUF 44100 cells allot
: clbf echobuf 44100 cells 0 fill ; clbf

~reset                   2000e ~sine  osc1
500 1000 99e ~vol% 3000 25000 ~adsr  ads
                     osc1 ads ~vcvc  mod1

                   ~nill mod1 ~adder mxd1
            mxd1 echobuf 25000 ~delay echo1
                   echo1 0.5e ~amp   feedback

                    mxd1 mxd1 ~output outp

feedback mxd1 4+ !       \ << patch in feedback loop

~start
: ping clbf 1 ads 4+ ! ; \ clear buffer and start ADSR

ping ( ping... ping... ping... ping... )
```

Simulating a cricket. This code emulates the sound of a cricket. It uses two universal block generators to modulate the sound. Fortunately output stops after 3 seconds, as it is an ugly sound. The addition of 1 or more filters would clean the sound up. It also prints the cycles used.

```
\ cricket
\ resources used: 108 CPU cycles - 168 bytes

~reset      4000e ~block   osc1
       30e 40e ~% ~ublock  mod1
        3e 20e ~% ~ublock  mod2
        osc1 mod1 ~vcvc    snd1
        snd1 mod2 ~vcvc    snd2
        snd2 snd2 ~output  out1

: go ~start wait space .wvc wait wait ~stop ; go
```

The **~LOG** can **log the amount of CPU-cycles** used for every tick. It is done by using **WVCYCLES** as input like this:

```
create WVLOG 1024 cells allot  \ create 1024 cell array for log

wvcycles wvlog 1024 ~log LOG1  \ create log of CPU-cycles used
```

Eliminating clicks and pops of an ADSR. If an ADSR is used to modulate a sound, calling that sound rapidly twice after each other can result in audible clicking sounds. The reason is that the second time the ADSR is called, it goes back to 0 immediately before ramping up the attack phase. This rapid change to 0 is what causes the clicks.

Example where clicks can be clearly heard:

```
~reset                    2000e ~sine     osc1
 500 1000 99e ~vol% 3000 25000 ~adsr     adsr1
                    osc1 adsr1 ~vcvc     mod1

                     mod1 mod1 ~output   outp
~start
: go 5 0 do 1 adsr1 4+ ! blink blink loop ;
go
```

The clicks can be suppressed by **filtering the output of the ADSR** with an SVFILTER. By varying the cutoff frequency of the filter, the strength of suppression can lessened or strengthened. However, the filter does reduce the sharpness of the attack-phase, which might be unwanted.

```
~reset
                           2000e ~sine     osc1
  500 1000 99e ~vol% 3000 25000 ~adsr     adsr1
                adsr1 100e 2.0e ~svfilter flt1
                      osc1 flt1 ~vcvc     mod1

                      mod1 mod1 ~output   outp
~start
: go 5 0 do 1 adsr1 4+ ! blink blink loop ;
go
```

The range-limited VCO generates a frequency between a user-definable range of frequencies. The example below sounds a smoothly changing frequency between 440 and 880 Hz.

```
\ up and up and up...
\ resources used: 101c CPU cycles - 112 bytes

~reset            2.0e ~saw    osc1
        osc1 440e 880e ~rngvco vco1  \ < the three elements...
                    0e ~sine   osc2  \ < to use an oscillator...
            vco1 osc2 ~vco>osc       \ < as a VCO
           osc2 ~nill ~output outp
~start
```

Using **a filtered slow noise generator as input to a VCO** gives a unique 'musical' experience. Apparently there was a time when people saw this as the future of music. Fortunately it didn't happen.

In the example a noise generator generates random voltages at 5 Hz. This is twice filtered to avoid clicks and plops as much as possible, and then used as input tot the VCO. The VCO controls a sine-oscillator. Only one input-channel of the output-element is used, that is loud enough.

```
\ modern music
\ resources used: 137c CPU cycles - 224 bytes

~reset                  5e ~fnoise    osc1
          osc1 200e 0.707e ~svfilter  fil1
          fil1 200e 0.70e7 ~svfilter  fil2

                      fil2 ~vco       vco1
                        0e ~sine      sin1
                 vco1 sin1 ~vco>osc \ connect VCO and oscillator

                sin1 ~nill ~output    out
~start
```

The same setup **but now with a range limited VCO** sounds somewhat better. But still sounds bad. Feel free to experiment with the values. See what effects you can get.

```
\ cacofonie-2 - using a range limited VCO
\ resources used: 125-171c CPU cycles - 232 bytes

~reset                 5e ~fnoise    osc1
        osc1 200e 0.707e ~svfilter  fil1
        fil1 200e 0.707e ~svfilter  fil2

          fil2 220e 440e ~rngvco    vco1
                     0e ~sine      sin1
               vco1 sin1 ~vco>osc

              sin1 ~nill ~output    out
~start
```

Different colours of noise. Different kind of noises are traditionally identified with names of colours. White noise is a noise where the sound-energy is spread equally across the spectrum. It sounds like the sharp hissing of steam escaping a steam-locomotive. Pink is noise where the sound-energy in each octave is equal. It is a softer noise with more lower tones. Other names of colours, like red or grey, are used for even softer noise.

The following examples show approximations of the different noise-colours using a low-pass filter.

```
\ white noise
~reset                0 ~snoise     osc1
             osc1 0.5e ~amp         2out
            2out ~nill ~output      outp
~start

\ pink noise
~reset               0 ~snoise     osc1
     osc1 4000e 0.505e ~svfilter   ns1
             ns1 ~nill ~output     outp
~start


\ red noise
~reset               0 ~snoise     osc1
     osc1 1500e 0.505e ~svfilter   ns1
             ns1 ~nill ~output     outp
~start


\ gray noise
~reset               0 ~snoise     osc1
     osc1 2500e 0.707e ~svfilter   ns1
       ns1 400e 0.707e ~svfilter   ns2
              ns2 1.8e ~amp        2out
            2out ~nill ~output     outp
~start
```

Making a sound without an oscillator. Normally an oscillator is the basis for sound-generation. A different way of creating a sound is by 'pinging' a resonating filter. A filter with a very high Q, ie with a very low dampening of the filter will resonate when pinged.

'Pinging' is nothing more than filling the memory of the filter as if it was filtering a signal with maximum amplitude.

For the state variable filter this means putting a 1.0e in one of the buffers and a zero in the other. Please notice that the cut-off frequency is the frequency of the generated sound, and that the Q is very high. The sound has a 'wooden' quality. Different values for the Q give different sounding tones. With a Q-value of 1000e, the sound is more bell-like.

```
\ generating a sound by pinging a filter
\ resources used: 43c CPU cycles - 76 bytes

~reset   ~nill 2000e 100e ~resfilter flt1   \ Q=100!
               flt1 flt1 ~output    outp
~start

: pok  0e flt1 40 + f!       \  0e in buffer 1
     1.0e flt1 36 + f! ;     \  1.0 in buffer 0

: go 15 0 do pok blink blink loop ;
go
```

Creating a glissando effect

In WAVE, the glissando effect is created by adding a glissando element to an oscillator. Nothing else changes. The glissando element takes two values as input: the base-address of the oscillator and a float value for the reduction-constant.
The reduction-constant specifies how much the difference between the wanted generated frequency and the actual frequency is reduced per tick. A value of 0.999e is a good value to start with.
The start frequency of the glissando element is the frequency specified in the oscillator at the time the glissando element was created.

The example switches 10 times between two notes with a clearly audible glissando-effect. Using 0.9999e as reduction-factor is an interesting variation.

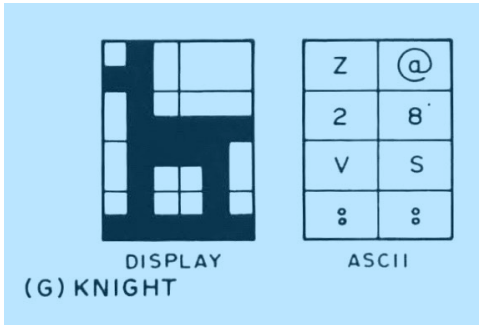
```
~reset      400e ~sine      osc1
       osc1 osc1 ~output    out

    osc1 0.999e ~glissando gli

: nt ~hz osc1 8 + ! ;

: go 10 0 do
      400e nt wait
      900e nt wait
   loop 0e nt ;   \ < creative way of silencing the oscillator

~start go
```

PIXEL grafical system

PIXEL is under development and, more importantly, under reconsideration.

I have the feeling, subjective no doubt, that the present resolution of 1024x768 and the 24bit colours does not reflect the spirit of the kind of computing I want to do. So it might be that in future both the resolution and the amount of colours are reduced.
Yes, you heard that right, reduced. I have experimented with resolutions of 704*528, 768*576, 720*540, 800*600 and 848*480. These resolutions display well on most TVs and monitors, as they are standard HDMI resolutions.
In addition, the Raspberry Pi has the option of using a 256 color table. A screen could thus contain 256 colours chosen from the full 24 bit range. These options have a better vibe than the present very bland 1024*748 24bit colour system. The performance of PIXEL would be higher as well, as the complete screen-memory fits in the level2 cache, also reducing the needed memory-bandwidth. And one could use the colour-table for fast pseudo-animation. Ooh happiness!

Overview of the present 1024*748 24bit system:
The wabiPi4 system contains an experimental and very basic graphical system called PIXEL. Not as basic as the drawing of a knight for a chess-game on the left, but basic nonetheless. The setup of PIXEL tries to follow the way the graphical systems of the home-computers of the mid-eighties worked.



All the management tasks 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 wabiPi4. Developing the latest 3D-shooter is probably not feasible.

THE PIXEL WINDOW-SYSTEM

WabiPi4 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 wabiPi4. The order of visible windows is fixed, window 0 is the lowest and the other windows are visible in order of numbering on top of window 0.

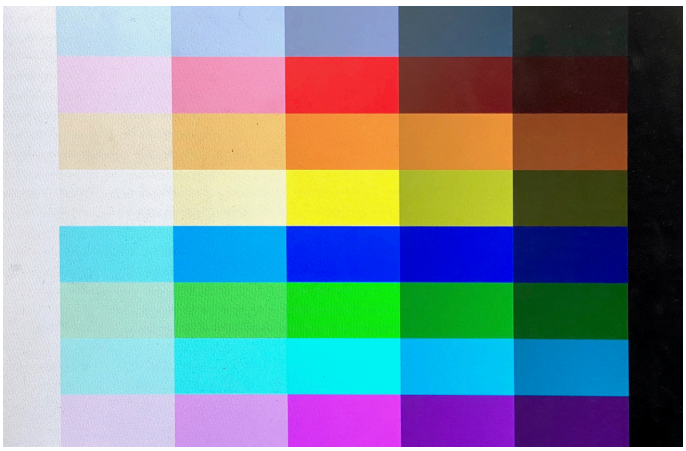
Window 0 is special. It is always visible and it always has the same size as the screen, and thus fills the background completely. Any other visible window floats on top of window 0.
A number of routines are specifically made for window 0 to make use of the fact that the size and memory location of window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

COLOUR-SYSTEM:

WabiPi4 uses 24 of the available 32 bits for defining colours, in the normal RGB order. The 4th byte is the alpha-byte, which on the Raspberry, thanks to a hardware problem, does not function. WabiPi4 can used it 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**



In addition there is the very special color green, defined by my daughter Lize:
greenlo

The Corbusier colour palettes:

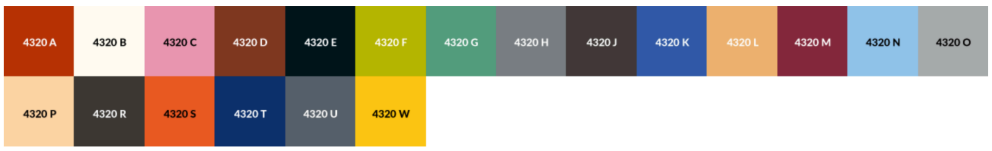
The 1931 and 1959 colour palettes of Le Corbusier have also been pre-defined as CONSTANTS. You need a good monitor to see them as they are intended. The special property of these colours is that all colours match when combined.

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



WORDS AVAILABLE FOR PIXEL

Control of PIXEL:

PIXELWFE (=VALUE): the PIXEL-system functions by aggregating all windows into one screen-buffer, which is then copied to the actual screen memory. Once this is done, PIXEL waits a short period. The length of this waiting-period can be controlled by the user with the value **PIXELWFE.** The default value is 800 and gives a refresh-rate of around 77 frames/sec. (this is the rate with which the windows are updated internally, this is not the rate the screen-buffer is send to the monitor)
The advantage of making the refresh-rate slower is that this makes more memory-bandwidth available to other applications. A

memory-intensive task will usually go a bit faster if the value in **PIXELWFE** is raised.

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 optimised assembly routine, and as such upto 20x faster than an equivalent in wabiPi4.

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 wabiPi4 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 introduction 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 to the returned address gives the actual address of a value.

Compilation/Execution control

COMPILATION/EXECUTION CONTROL IS EXPERIMENTAL AND MIGHT NOT FUNCTION

This section of wabiPi4 was developed with the 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

WabiPi4 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 system-stability of wabiPi4 is not assured due to **CATCH** and **THROW.**

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

Optimisation in wabiPi4

During the development of wabiPi4, 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 wabiPi4

"This is faster, a little bit bigger."

Charles Moore, May 22, 1999

WabiPi4 uses a mix of subroutine threading and native code. At all times wabiPi4, and any Forth program compiled by the user, is running in ARMv8 assembly. This is the fastest way of running a Forth-program for the ARM Cortex-a72 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 wabiPi4.
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 wabiPi4 more stable and resilient against mistakes. For instance if you pop a value from the return-stack by mistake, wabiPi4 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, wabiPi4 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 wabiPi4 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. WabiPi4 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 for I and the loop-limit 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. About a third of all words in the dictionary can be inlined.
Inlining avoids time-consuming calls to subroutines. And the calls that are still made are mostly to words where inlining does not bring speed-advantages. WabiPi4 inlines a lot. Most programs mainly consist of ARM-opcodes with a few subroutine-calls dispersed in-between the opcodes.

Values, variables, constants and literals are always inlined by wabiPi4. There simply is no benefit of not inlining these. The inlining of other words can be controlled by the user (see later).

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 '**+**'.

An 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. 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, wabiPi4 compounds words where possible. As an example: if you put '**OVER +**' in the source code, wabiPi4 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. WabiPi4 assumes that if you do not use the ANSI-standard word, this must be on purpose.

WabiPi4 will also not correct programming-errors like a **DUP** followed by a **DROP**. WabiPi4 assumes that if you do something seemingly so illogical, that undoubtedly you must have a good reason for doing it.

First final remark: additional combinations are added to wabiPi4 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.

Last final remark: the CPU in the pi4b, the Cortex-A72, also does a bit of pruning. It is called 'synthesis' and it is not

well documented. But combinations like '**DUP DROP**' are faster on a Cortex-A72 than one would expect.

User-options of wabiPi4 optimisation:

USER OPTIONS INLINING

WabiPi4 can inline primitives during compilation of definitions. Wether an individual primitive is inlined or not depends on exactly two factors:

- the **inlining-value** available in the header of each primitive.
- the value contained in **MAXINLINE** at the time the primitive is included into a definition.

When the **inlining-value** is 0x0, the primitive is never inlined. This is the case for all non-primitive definitions and some primitive definitions.
When the inlining value is not 0x0, then the primitive is inlined if the **inlining-value** is equal to or smaller than the value in **MAXINLINE.**

All primitives have been timed with and without inlining, and for some primitives inlining does not bring a speed-advantage. These have an inlining-value of 0x0 in the header, and will therefore never be inlined.

Inlining can be influenced by the user with 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 part of a program or not.

Inlining can also be influenced by the value **MAXINLINE.**

Inlining is most beneficial for short primitives. Some primitives are only 1 opcode long and execute in 1 or even less cycles. A branch to such a primitive and the branch back together take at least 3 cycles. This means that inlining reduces the execution time from 4c to 1c (or even to 0c!).

For the pi3, inlining a maximum of 8 opcodes is optimal. Inlining longer primitives does not make the program faster, indeed the program can even be slower. For the pi4, with a different ARM-processor, tests have shown that inlining is always beneficial.

The overall speed advantage of inlining can be quit large. In some cases code can run 300% faster. The sieve benchmark without inlining runs in ~2425us. With inlining it takes ~890us, a difference of ~270%.
The lowest improvement I have seen is a 0.2% faster read-speed with a driver I developed for flash-memory. Avoiding all optimisation reduced the size of the program by 6.5%, but memory is rarely an issue. And that is why inlining is switched on by default.

Inlining is fully transparent to the user, there are no risks associated with inlining. The only, small, disadvantage is that inlining increases the size of a program. To give you an indication: small programs can double in size, whereas larger programs rarely increase size by more than 15%.

Inlining can be halted temporarily with **HALTINLINE**, or forced temporarily with **FORCEINLINE**. **HALTINLINE** and **FORCEINLINE** remember the previous inline-settings, so that inlining can later be set back to the settings as they were by using **INLINEASBEFORE**.

HALTINLINE and **FORCEINLINE** are not nest-able. So if you use **HALTINLINE** or **FORCEINLINE** twice after each other, only the last value will be remembered.

INLINE (--): Immediate word. Starts inlining from that point onwards by setting the value **MAXINLINE** to -1 for the pi4 and 8 for the pi3.

NOINLINE (--): Immediate word. Stops inlining from that point onwards by setting the value **MAXINLINE** to 0x0. Can be used within the source-code to stop inlining from that point of the source-code.

HALTINLINE (--): halts inlining temporarily and remembers the previous inlining-settings. Not nestable.

FORCEINLINE (--): starts inlining and remembers the previous inlining-settings. Not nestable.

INLINEASBEFORE (--): reinstates the previous inlining-setting before **HALTINLINE** or **FORCEINLINE** were executed.

MAXINLINE (-- max_inline_length): value - defines the upper limit of the length of primitives which are inlined. For instance, if **MAXINLINE** contains the value 8, primitives with a length of 8 or less opcodes are inlined. Primitives with a length of 9 or longer are not inlined.
If **MAXINLINE** contains -1, inlining is done for all inlinable primitives. If **MAXINLINE** contains 0x0, inlining is switched off. For the pi3 a value of 8 is optimal, for the pi4 -1 is optimal.

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 this lucky, consider playing the lotto...). 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 and so not visible when listing words with **WORDS**. But, as with all hidden words, you can use them in a program if you want to.

COMPOUND (--): immediate - compounding is done from the location of **COMPOUND** in the source.

NOCOMPOUND (--): immediate - halts compounding from the location of **NOCOMPOUND** in the source.

.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 until 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 2420 words, using the classic way of trying to **FIND** a word, wabi-Forth on average needs ~25000 machine cycles to find a definition. Confirming that a given word does not exist in the dictionary takes ~49000 machine cycles.

A usual way of speeding up **FIND** is by logically 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.

A better way of speeding up the **FIND** is by using a hash-table. This is transparent to the user, it is just there and functions. Finding a 5 character word in a 2420 word dictionary with support of a hash-table is around a 550 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 2000 times faster.

Quickly finding a definition 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 2500 word dictionary, such a short definition can be confirmed to *not* exist in 31 ns. Which is around 850 times faster than using the classic way. This is the best case, the average improvement is closer to 400 times faster.

MEASURING THE BENEFITS OF THE HASH-TABLE

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

```
s" y" sliteral tsty
: tst0 (( tsty uncount find 2drop )) ; \ note: UNCOUNT !
```

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

```
usehtb
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 850* faster than without the hash-table.

```
31113 us -> 56.0 cycles
26632375 us -> 47938.3 cycles
```

A longer string takes a bit longer, but is still very much faster with a hash-table than without. Try it!

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

output:

```
48225 us -> 86.8 cycles ok
27956388 us -> 50321.5 cycles
```

5 MILLION DEFINITIONS

If you are curious how the numbers are for a directory with 5 million definitions, this is the code to test that:

```
s" y" sliteral tsty
: tst0 (( tsty uncount find 2drop )) ; \ note: UNCOUNT !
: tst1 tsty uncount find 2drop ;       \ note: UNCOUNT !

usehtb drop
tst0
nohtb
c[ tst1 ]c.
usehtb drop
```

These are the results:

With use of the hash-table:

```
49447 us -> 94.0 cycles
```

When the hash-table is switched of:

```
666025780 cycles
```

As such this is quit fast, only 124 cycles/word in the dictionary. But due to the large dictionary it is 7 million times slower than when using a hash-table. Which clearly shows that without a hash-table, or comparable functionality, big dictionaries are not feasible.

In case you are wondering how to get 5 million definitions in the dictionary: the chapter on overflow-protected string variables contains an example which does that.

ANOTHER EXAMPLE

demonstrates the effect of a hash-table on the speed of interpretation. It measures the shortest possible time in total to:

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

TST0 is normal compiled wabiPi4, while **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 ( << grab a coffee, this takes a while... )
usehtb
```

You will see that the difference in running time is large, the running time of the second run of TST1 is around 71 second. The slowest way is 32 thousand times slower than the fastest way. Using the hash-table is ~165* faster than not using the hash-table with around 2500 words in the dictionary.

TECHNICAL SETUP OF HASH-TABLE

The hash-table in the wabiPi4 system is a two-level system, with an entry-pool of 838889 hash-table rows and a link-pool (backup-area) containing 4753484 rows. In total this uses 64MB of memory.
Each row contains 3 32bit fields: the link-field, the CRC-field and the Execution Token.

Placing a new definition in the hash-table entails the following steps:

- a 32bit CRC is made of the name (case-insensitive) of the new definition, the length of the name and the wordlist identifier.
- This CRC is divided by 838889. The remainder of this division is used as index to the row in the entry-pool where the new definition will be placed.
- If this row is already in use by another definition, this pre-existing old entry will be moved into the next free row in the backup-area.
- If there was no pre-existing entry, the link field of the new entry is set to 0x0. Otherwise the link-field of the new entry is set to point to the moved old entry. This way both the old and new definition can be found.

The implementation with entry-pool and backup-area ensures that even with a very large dictionary, containing millions of definitions, the average number of checks to be done to find a word is only ~3.4. The average number of checks required to confirm that a word does not exist in the library is ~6.7. The 32 bit CRC-field ensures that in 99.9999999767% of cases only one, fast, comparison of two 32bit value has to be done for each entry.

User functionality: A special feature of the wabi-Forth hash-table is that the use of the hash-table by **FIND** can be switched off. The hash-table is then available for 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, hash-table functionality is available to speed up user-programs if so wanted.

Using a hash table is an advanced topic. The following words are available if you want to give it a try.

NOHTB (--): stops the use of the hash-table by wabi-system, and makes the hash table available for use by the user. Compilation still functions, but is slower.

USEHTB (--): Fully rehashes the dictionary into the hash-table and starts the use of the hash-table by the wabi-system. Rehashing is normally instantaneous. Rehashing a dictionary containing 5 million words takes 1.9 secs.

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. Used by **FORGET**.

HASHWORD (message, addr, len --): puts the 32bit message in the hash-table based on a hash of the data at the specified address and the length. Which meaning the data has, und thus which 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 wabiPi4 is using the hash-table, it stores the XT of a word in the message-field.

HTBFIND (addr_of_counted_string -- message): finds the corresponding message, based on the hashing of the counted string at address on stack.
Please note that the standard Forth word **FIND**, und thus also **HTBFIND**, expects a counted string as input.

Real time clock module

The RTC related words only function if a RTC-module is connected to the Raspberry. WabiPi4 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'.

ROM-modules

THE ROM-MODULE/FLASH-MEMORY SYSTEM IS UNDER ACTIVE DEVELOPMENT

WabiPi4 more or less emulates ROM-modules like the BBC-computer had. This is based on external FLASH-memory.

The BBC-computer contained sockets for 4 ROM-modules. A ROM-module would make an often-used program available directly. For instance a word-processor or a second programming language. These ROM-modules had to come from an external party though, programming a ROM required advanced technical knowledge.

The ROM-module system, analog to the BBC-computer, is intended as a system to make often used resources available to WabiPi4. Please note that it is not a full-blown file-system. For instance there is no directory-structure or renaming of files.

The obvious difference with the original ROM-modules is that wabiPi4 allows the user to put the files into ROM. And the maximum capacity is higher, megabytes instead of kilobytes.

Examples:

- regularly used programs, like an editor or a great game
- setup-routines and personal system-preferences
- drivers normally not available in the wabiPi4 system
- extensions to WabiForth

The Flash-system contains a 2-dimensional CRC based **error-correction** system of the stored data and **wear-levelling** of the individual sectors of the Flash memory. This to ensure reliable storage over longer terms.

Specifications:

maximum capacity	The system can handle Flash memory up to 42GB	This is a theoretical maximum as SPI-based FLASH memory is available upto 128MB.
error-correction method	2 dimensional CRC-based	the system can rapidly correct up to 250 bit-errors per 4 kB sector, more slowly upto 500 bit-errors, and very slowly upto 750 bit-errors.
wear-levelling	automatic balancing of sector erase/write cycles	sector erase/write cycles are balanced within 1% for 100.000 erase/write cycles. That is once any of the sectors reaches 100.000 erase/write cycles, the other sectors have at least 99.000 write/erase cycles.

maximum number of files	each MB of storage gives 256 sectors, each having 2700 bytes available to the user.	A file is at least 1 sector.

PRIMARY WORDS:

FILE>ROM (addr len <name_string> -- ROMid#): saves the file at addr with length len in the flash-memory with <name>. Returns a unique file-identifier under which the file is known in the file-system.

ROM>FILE (addr ROMid# -- len): gets file identified by ROMid# from ROM and moves it into the buffer starting at addr. It is up to the user to guarantee that there is enough space in the buffer to contain the file.

SOURCELOAD (addr len --): loads the source-code contained in the buffer specified by addr and len into the dictionary.

ROMLOAD (ROMid# -- len): loads the source-code contained in the ROM identified by ROMid# into dictionary.

REMOVEROM (ROMid# --): removes the rom-file identified by id from Flash-memory. Cannot be undone.

ROMSIZE (ROMid# -- size/-1): returns the size of the file contained in the ROM identified by ROMid#. Returns -1 if the specified ROM does not exist.

.ROMS (--): prints tabel with all known ROMs

FREEBYTES (-- freebytes): returns the available capacity of the flash-memory

OTHER USEFUL WORDS:

.PAGE (page# --): prints the specified page-number from flash.

MYFILE (-- buf_addr): constant, returns address of a 256KB large memory-buffer which can be used by the programmer as a scratch-area when testing WAF.

SECBUF (-- addr): constant, returns the address of the sector-buffer. This is the buffer used by WAF to construct or check individual sectors saved to or loaded from FLASH. Is typically used together with .SECBUF.

.SECBUF (buffer --): prints an overview of a sector from RAM. The typical use is to give the constant **SECBUF** as input:

Example: **secbuf .secbuf**

.FLSC (--): prints an overview of all flash-sectors

.SAT (--): prints sector-allocation table

The following definitions support loading text-files over the UART-interface into flash-memory. They are intended as a basis for developing your own words.

>|||> (--): Marks start of file

<|||< (--): Marks end of file

∗STARTFILE (-- start of loaded file)

∗ENDFILE (-- end of loaded file)

FUNCTIONALITY TO BE ADDED:

- **renaming of a file** – can presently only be done by saving a file under a new name
- **change the ID# of a file** – ditto
- **save to a specific ID#** – presently the system gives out the next ID#

•

Finding Yourself

Printing the name of a definition
If you want to know the execution token **(XT)** of a definition, you use **' (TICK),** an ANSI standard word**.** 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 the address is 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 in which **.MYSELF** is located**.**
.BYWHOM prints the calling definition**.**

```
: .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? ;
```

Execute **TST2** and you will see that it was called by **INTERPRET,** and **WHO?** will report it was called by **TST2.**
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 guide**.**

.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"**.**

.XTNAMECAPS (xt --)**:** given a valid **XT,** prints the name of a definition in capital characters**.** If there is a 0x0 on the stack, **.XTNAMECAPS** 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**.** This word works well together with **.XTNAME** and **.XTNAMECAPS.**

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.

ARM-v8 assembler

An integrated ARMv8 assembler was an important requirement for WabiForth. After a reboot or '**COLD**', the assembler is available directly. No need for messing around with vocabularies or loading of source-code.
You can use '**FORGET ASSEMBLER**' immediately after a reboot or '**COLD**' to remove the assembler from the dictionary. This saves a cool 0,0013% of total available memory.

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 is presently 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.** WabiPi4 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 than the optimiser.

Fourth reason: **access to parts of the ARM-system** which are not covered by wabiPi4 functionality. For instance wabiPi4 has no build in support for the debug-functionality of the ARM-processor, although it is switched on. 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 wabiPi4 assembler is that it is possible to mix Forth words and assembly code at will, and within any definition.

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 assemble anything

Assembler words in wabiPi4 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 square brackets ('**[**' and '**]**') around blocks of assembly words.

ASSEMBLER WORDS

Normal definitions and primitives
Opcodes are either part of a normal wabiPi4 definition or part of a primitive. The distinction is important. Primitives must follow certain rules to function correctly.

CODE: defines a primitive
In most Forths the colon, ‘:’, is for defining Forth-words and the word **CODE:** (or something similar) is used for definitions using assembly.
WabiPi4 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 at he start of the definition, nor does it restore the link-register at the end of the definition. And exactly 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,** can be used outside of a primitive, but their preservation is not assured after a call to another word.

WabiPi4 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 for primitives)
r12,  =    w,    (scratch for primitives)
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, -> the 8b immediate is 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 a subroutine with the destination 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

The assembler contains a simple label system. It has a capacity of a maximum of 512 individual labels. There are 12 pre-defined labels, named **label0:** to **label11:.** The word **NEWLBL** lets you create more labels.

The word **LBL[** is used to initiate the labels.

LBL[(--): resets the label-addresses. Normally used once in a program. Multiple wabiForth definitions can share labels. This makes it possible to branch from one definition to a label inside another definition.

NEWLBL (<name> --): using the name after the word NEWLBL, creates a new assembler-label.

Pre-defined labels:

label0:
label1:
label2:
label3:
label4:
label5:
label6:
label7:
label8:
label9:
label10:
label11:

Lessons from one opcode...

The simplest definition with assembly possible is a definition containing 1 opcode. The example here just 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+ ( n -- n+15 )   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 execution-times of the three definitions with the words **((** and **))**. 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  8334 us -> 15.0 cycles
tstw  7779 us -> 14.0 cycles
tstc  2224 us ->  4.0 cycles
```

Assembly is mostly used to raise performance, and the increase in speed is clearly noticeable.

But there is one more 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 )) ;
tsti  556 us -> 1.0 cycles
```

A very impressive result! The routine is now 15 times faster than the original routine. The main reason is that inlining avoids the linked branch to and the return from the called definition.

Lessons learned:

1. The rudimentary compiler of wabiPi4, coupled with the excellent Cortex-a72 processor, do a good job of creating and running efficient and fast code.
2. Assembly is faster than Forth.
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 only 3 opcodes, lower ASCII characters are converted to capitals. A version with **CODE:** will be faster, but it does function with a **COLON.**

```
: 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 wabiPi4 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** and **DROP** separately.
If **CODE:** and **2 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. On the Cortex-a72 the latter is the shorter but NOT the faster option. This in contrast to the Cortex-a53 processor of the pi3b+ where the second is much faster. This is a consequence of design-decisions of the ARM-corporation.

Example with **conditional branch** - 2.0 cycles execution time

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

Example with **conditional execution** - 2.0 cycles execution time

```
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
```

This example is not relevant for the RPi4b, as **2@** (**twofetch**) on the rpi4b does not need memory to be aligned. But it still is a good example how to make a **2@** (**twofetch**).

```
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 -- )
  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 32 bit immediate into a register. On the Cortex-a72, the processor in the RPi 4b, 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 wabiPi4, assembly and wabiPi4 definitions can be mixed at will. This obviates the need to call a wabiPi4 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 getting the content of a VALUE into a register. The example does nothing useful, but shows nicely how the content of a VALUE can be loaded into a register without using the stack.

```
0 value TESTVAL

: getwe ( -- val_of_testval )
[  top, dts, 4 i-]!, str,    \ dup
   r0, ' testval 20 + ldv32, \ put xt of TESTVAL in r0
   top, r0, ldr,             \ r0 is pointer to the value
] ;
```

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 wabiPi4:

```
variable seed
2345 seed !   \ fill the seed with a starting value

: forthrandom ( address_seed - rndm_val ) ( 44c )
  dup >r @ dup >r
  dup 5 lshift xor
  dup 9 rshift xor
  dup 7 lshift xor
  r> xor dup r> ! ;
```

The same routine, but now using assembly. The code for the Forth version is 37 opcodes long once compiled. The assembly-version is only 8 opcodes long (7 opcodes when inlining) and is 3.6 times faster.

```
variable seed
2345 seed !   \ fill the seed with a starting value

code: asmrandom ( address_seed - rndm_val ) ( 12c )
[ v, top, ldr,
  w, v, v, 5 lsl#, eor,
  w, w, w, 9 lsr#, eor,
  w, w, w, 7 lsl#, eor,
  w, w, v, eor,
   w, top, str,
  top, w, mov, ]
; 7 inlinable
```

The following example comes from an SHA-256 implementation, where the most frequently used functions are called SIG0 and SIG1.

In Forth these words look like this:

```
: SIG0 ( x -- n )
  dup >r  7 rotr r@ 18 rotr xor r>  3 rshift xor ; \ 30c
: SIG1 ( x -- n )
  dup >r 17 rotr r@ 19 rotr xor r> 10 rshift xor ; \ 30c
```

These words compile to 22 opcodes and each take 30 cycles to complete. In ARM32 opcodes, the same functionality can be done using just 3 opcodes. And completion only takes 2 cycles. Overall the SHA256 hash generation is 15% faster with just this little optimisation.

```
code: SIG0 ( x -- n ) \ 2c
[ w, top, 7 ror#, mov,
  w, w, top, 18 ror#, eor,
  top, w, top, 3 lsr#, eor, ] ; 3 inlinable

code: SIG1 ( x -- n ) \ 2c
[ w, top, 17 ror#, mov,
  w, w, top, 19 ror#, eor,
  top, w, top, 10 lsr#, eor, ] ; 3 inlinable
```

The next 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 3 cycles. In wabiPi4 it takes 28 cycles. When speed is essential, and it often is with SPI-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 )) ; \ -> 3.0c
```

The same coded in wabiPi4:

```
: 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 )) ; \ -> 28.0c
```

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 wabiPi4 assembler uses the two values separately rather than combine into the one resulting value. Any 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
```

Mixing forth and assembly – the BYTE-Sieve

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, using an optimizing Fortran compiler, 'did' this in 0.11 seconds. The Raspberry Pi 4b, using wabiPi4, takes around 890 micro seconds, around 124 times faster than the CRAY-1. The assembler version shown below takes around 665 microseconds, around 165 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 shows mixing assembly and normal wabiPi4 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, ;

\ the actual SIEVE -----
lbl[                          \ reset of labels

: asmsieve \ ( odd# -- #primes, us )
[ prologmax ]                 \ brackets needed!!

  mcs swap                    \ put microsecond on stack
  dup 8 +                     \ calculate needed size of array
  allocate                    \ array in high memory using

[  top, dts, 4 ]i+!, ldr,     \ drop flag from allocate

  r1, top, mov,               \ top in r1
  top, dts, 4 ]i+!, ldr,      \ =drop

  r6, top, mov,               \ top in r6
  top, dts, 4 ]i+!, ldr,      \ =drop

  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,     \ =dup
  top, w, mov, ]             \ push content of w on stack

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

[ r3, 0 movw,
  r4, 0 movw,

label3:
  r7, r1, r3, +], ldrb,      \ get r3'th element
  r7, 0 0 i#, cmp,
  beq, label4:

  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#
  bgt, label5:

  r4, r1, r8, +], strb,      \ flags[k]=0
  r8, r8, r7, add,
  b, label6:                 \ 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,
  blt, label3:               \ end sieve-loop

  r0, r0, 0 1 i#, subs,      \ 1 from r0=m
  bne, label1:   ]           \ end 10*-loop

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

Enter:

```
8190 asmsieve . .
```

and you will get something like 670 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...

Mixing forth and assembly – SHA256 hashing

This is a fast implementation of the SHA-256 algorithm. With an address and a length on stack as input, it calculates the SHA-256 hash for the data contained in that buffer. The SHA-256 is presently the most used hashing function, and there are no knowns weak spots of this algorithm.

The implementation shows how Forth and assembly can be mixed within definitions.

A copy of this routine, with slightly more comments, can be found in Project Forth Works. Copying that saves a lot of typing. In Project Forth Works you will also find a version coded exclusively in generic Forth. That version is 16 times slower than this assembly version, but it makes the function of the SHA-algorithm easier to understand. Especially as the names of the words follow the names used in the standard description.

```
\ SHA-256 mixed Forth/assembly version

\ ===== variables, arrays and helper-functions
\ h0-h7 form the actual hash
    0 value h0  0 value h1  0 value h2  0 value h3
    0 value h4  0 value h5  0 value h6  0 value h7

    0 value lenmess  0 value addrmess

\ message block = 16 words for message and...
\ ...48 words for temp storage of Wi for last 48 rounds

    256 allocate drop constant MBLCK    \ message block
     32 allocate drop constant WBLCK    \ variables wa-wh

: MESS! ( m n -- )    4* mblck + ! ;
: CLEARMESS 16 0 do 0 i mess! loop ;    \ clears message-block

: HASH->WVAR \ move hash to working variables
    h0 WBLCK  0 +  !
    h1 WBLCK  4 +  !
    h2 WBLCK  8 +  !
    h3 WBLCK 12 +  !
    h4 WBLCK 16 +  !
    h5 WBLCK 20 +  !
    h6 WBLCK 24 +  !
    h7 WBLCK 28 +  ! ;

: WVAR->HASH \ add working variables to hash
    WBLCK  0 + @  +to h0
    WBLCK  4 + @  +to h1
    WBLCK  8 + @  +to h2
    WBLCK 12 + @  +to h3
    WBLCK 16 + @  +to h4
    WBLCK 20 + @  +to h5
    WBLCK 24 + @  +to h6
    WBLCK 28 + @  +to h7 ;
```

```
hex create  KI[]  \ 64 SHA-256 constants
428a2f98 , 71374491 , b5c0fbcf , e9b5dba5 , 3956c25b ,
  59f111f1 , 923f82a4 , ab1c5ed5 ,
d807aa98 , 12835b01 , 243185be , 550c7dc3 , 72be5d74 ,
  80deb1fe , 9bdc06a7 , c19bf174 ,
e49b69c1 , efbe4786 , 0fc19dc6 , 240ca1cc , 2de92c6f ,
  4a7484aa , 5cb0a9dc , 76f988da ,
983e5152 , a831c66d , b00327c8 , bf597fc7 , c6e00bf3 ,
  d5a79147 , 06ca6351 , 14292967 ,
27b70a85 , 2e1b2138 , 4d2c6dfc , 53380d13 , 650a7354 ,
  766a0abb , 81c2c92e , 92722c85 ,
a2bfe8a1 , a81a664b , c24b8b70 , c76c51a3 , d192e819 ,
  d6990624 , f40e3585 , 106aa070 ,
19a4c116 , 1e376c08 , 2748774c , 34b0bcb5 , 391c0cb3 ,
  4ed8aa4a , 5b9cca4f , 682e6ff3 ,
748f82ee , 78a5636f , 84c87814 , 8cc70208 , 90befffa ,
  a4506ceb , bef9a3f7 , c67178f2 ,
decimal

KI[] constant KITBL


\ =====  assembly macros
: ldv32, ( reg n -- )           \ load value n into reg -
  creates 2 opcodes!
  2dup
  16 lshift 16 rshift movw,
  16 rshift movt, ;

: prologcust
    r13, {, r0, r4, r-r, r6, r8, r-r, v, w, }!, stmfd, ; \
  saving v and w critical
: nextcust
    r13, {, r0, r4, r-r, r6, r8, r-r, v, w, }!, ldmfd, ; \
  restoring v and w critical


\ =====  The actual HASH-routines
hex : INITHASH \ fill first hash with intial values
  6a09e667 to h0  bb67ae85 to h1  3c6ef372 to h2  a54ff53a to h3
  510e527f to h4  9b05688c to h5  1f83d9ab to h6  5be0cd19 to h7
; decimal

\ wa=r0  wb=r1  wc=r2  wd=r3  we=r4  wf=r6
\ wg=r7  wh=r8  t1=v   t2=w   top=scratch

code: SUBLOOP  ( message+ki -- ) \ 25c - 37 opcodes
[   prologcust

    \ get working variables into registers - w=T2=> here scratch
    w, WBLCK ldv32,             \ get address of work-regs in w
    w, {, r0, r4, r-r, r6, r8, r-r, }, ldmia,

    \ start of T1
```

```
    v, top, r8, add,            \ =t1_step1

    \ now CH => ( we wf and ) ( we invert wg and )  xor ;
    top, r4, r6, and,           \ =we wf and - top=scratch
    w, r7, r4, bic,
    w, w, top, eor,             \ w=eor result

    v, v, w, add,               \ v=t1_step2

    \ now SIGMA1 ( x -- xn )
    w, r4, 6 ror#, mov,         \ r4=we
    w, w, r4, 11 ror#, eor,
    w, w, r4, 25 ror#, eor,

    v, v, w, add,               \ v=t1_final

    \ start of T2
    r8, r0, r1, and,            \ r8=wa and wb
    top, r0, r2, and,           \ top=wa and wc
    r8, r8, top, eor,
    top, r1, r2, and,           \ top=wb and wc
    w, top, r8, eor,            \ w=t2_step1

    \ now SIGMA0 as part of T2 with wa=r0
    top, r0, 2 ror#, mov,
    top, top, r0, 13 ror#, eor,
    top, top, r0, 22 ror#, eor,

    w, w, top, add,             \ w=t2_final

    \ actual update of the 8 working regs
    r8, r7, mov,                \ wg to wh
    r7, r6, mov,                \ wf to wg
    r6, r4, mov,                \ cave: we=r4 - we to wf

    r4, v, r3, add,             \ wd t1 + to we

    r3, r2, mov,                \ wc to wd
    r2, r1, mov,                \ wb to wc
    r1, r0, mov,                \ wa to wb

    r0, v, w, add,              \ t1 t2 + to wa

    \ write wregs back to work-block
    w, wblck ldv32,             \ get address of work-regs in w
    w, {, r0, r4, r-r, r6, r8, r-r, }, stmia,
    top, dts, 4 ]i+!, ldr,      \ =drop

    nextcust ] ;

: HASH1BLOCK \ 3-4us
    [ r13, {, r0, r3, r-r, }!, stmfd, ]

    hash->wvar
```

```
  \ first digest 16*4 bytes message-data from message-block
16 0 do

  \ the next part saves 1c from 67.5 to 66.5c/subloop
    [ top, dts, 4 i-]!, str,      \ dup

      w, mblck ldv32,
      w, w, i, 2 lsl#, add,
      top, w, ldr,

      v, kitbl ldv32,
      w, v, i, 2 lsl#, add,
      v, w, ldr,

      top, top, v, add, ]

      subloop
  loop

  \ the data from the first 16 rounds is than scrambled

  64 16 do

  \ the following saves ~20c/subloop
    [ top, dts, 4 i-]!, str,          \ dup

  \ =====  line 1
      r0, mblck ldv32,
      r2, i, 0 2 i#, sub,             \ r2=i-2 - r2=temp
      top, r0, r2, 2 lsl#, +], ldr,

      r3, top, 17 ror#, mov,
      r3, r3, top, 19 ror#, eor,
      top, r3, top, 10 lsr#, eor,

  \ =====  line 2
      r2, i, 0 7 i#, sub,             \ r2=i-7 - temp
      r2, r0, r2, 2 lsl#, +], ldr,    \ r2
      top, top, r2, add,              \ keep top

  \ =====  line 3
      r2, i, 0 15 i#, sub,            \ r2=i-15 - temp
      r2, r0, r2, 2 lsl#, +], ldr,

      r3, r2, 7 ror#, mov,
      r3, r3, r2, 18 ror#, eor,
      r3, r3, r2, 3 lsr#, eor,
      top, top, r3, add,

  \ =====  line 4
      r2, i, 0 16 i#, sub,            \ r2=i-16 - temp
      r2, r0, r2, 2 lsl#, +], ldr,    \ r2
      top, top, r2, add,

      w, mblck ldv32,
      w, w, i, 2 lsl#, add,
```

```
        top, w, str,

        v, kitbl ldv32,
        w, v, i, 2 lsl#, add,
        v, w, ldr,

        top, top, v, add, ]

        subloop
    loop

    wvar->hash
    [ r13, {, r0, r3, r-r, }!, ldmfd, ]
;

: FILLMESS ( addr -- ) \ fills message block with reversed words
    [ r13, {, r0, r3, r-r, }!, stmfd, ]

    16 0 do
      [ r0, top, i, 2 lsl#, +], ldr,    \ r0=[top + i*4]
        r1, mblck ldv32,
        r0, r0, rev,                    \ reverse bytes in word
        r0, r1, i, 2 lsl#, +], str, ]   \ store in message-block
    loop drop

    [ r13, {, r0, r3, r-r, }!, ldmfd, ] ;

: MBLCKREVERSE  ( -- ) \ reverse bytes in words in message-block
  16 0 do mblck i 4* + dup @ bytes>< swap ! loop ;

: PUTLEN ( len -- ) \ len*8 for number of bits
    0 3 dlshift  14 mess!  15 mess! ;

: GENSHA256 ( addr len -- )
    to lenmess  to addrmess

    inithash

    \ do hashing of full block
    lenmess 6 rshift dup >r
    0 ?do
        addrmess i 64 * + fillmess  \ i = blocknumber
        hash1block
    loop

    \ do hashing of last block
    r> 64 * addrmess + to addrmess
    lenmess 63 and >r               \ bytes_left in r:
    clearmess                       \ clear mblck

    addrmess mblck r@ cmove         \ move last bytes to mblck
    mblck r@ + 128 swap c!          \ add bit after last byte
    mblckreverse

    r> 55 > if
        hash1block
```

```
        clearmess
    then

    lenmess putlen                  \ put len at correct place
    hash1block ;                    \ and last block

: sha256 ( addr len -- ) gensha256 ;



\ =====  PRINT and TEST routines

: .HAHDR
  ." ---h0--- ---h1--- ---h2--- ---h3--- ---h4--- ---h5--- ---
  h6--- ---h7---" ;
: .HASH    h0 .32hex  h1 .32hex  h2 .32hex  h3 .32hex
     h4 .32hex  h5 .32hex  h6 .32hex  h7 .32hex ;

: .sha256 ( addr len -- )
  sha256 cr space ." SHA-256: " cr space .hash ;

: proof ( -- )
  inithash
  t[ s" abc" gensha256 ]t.
  cr 9 spaces .hahdr cr ." must be: "
  ." ba7816bf 8f01cfea 414140de 5dae2223 b00361a3 96177a9c
  b410ff61 f20015ad"
  cr ." SHA-256: " .hash ;
```

Creating primitives with high level code

This chapter shows a principle that might be fun to play with, and sometimes could help you save a few CPU-cycles. It's clearly not an example of 'good programming practise', but might help in understanding how wabiForth functions internally. See the remarks at the end of this chapter for an alternative way.

Inlinable primitives normally are made using the assembler. Now and then though it is possible to make a definition with normal wabiPi4 words and still make a inlinable primitive.
This seems a contradiction as a primitive is a word which per definition does not call other words. But inlining sometimes makes it possible anyhow.

It works like this: wabiPi4 by default inlines wherever useful. 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, due to inlining, 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 8-10 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 definition into a primitive is only useful if the compiler is able to generate efficient code.

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  wid:FORTH  defid:WORD
   1 00068C50  E92D4000  _-@_     | fastprolog lr only
   2 00068C54  E599C000  ____     | copy 2nd to reg:w
   3 00068C58  E08CB005  ____     | reg:v = top + reg:w
   4 00068C5C  E589B000  ____     | copy reg:v to 2nd
   5 00068C60  E1A0500C  __P_     | copy reg:w to top
   6 00068C64  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

name:BOUNDSC  wid:FORTH  inline:4ops  defid:CODE
   1 00068CA0  E599C000  ____      | copy 2nd to reg:w
   2 00068CA4  E08CB005  ____      | reg:v = top + reg:w
   3 00068CA8  E589B000  ____      | copy reg:v to 2nd
   4 00068CAC  E1A0500C  __P_      | copy reg:w to top
   5 00068CB0  E12FFF1E  _/__      | codenext
```

The wabiPi4 version runs in 8 cycles, the version using CODE: runs in 4 cycles. The internal version, of course an optimised inlinable primitive, also runs in 4 cycles. So in this specific example the Forth compiler found the optimal solution. Please note that this is an exception. An internal primitive is usually faster than a version coded in hi-level wabiPi4.

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 that might be used when creating a mask for isolating bits from a value.

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

name:>BIT  wid:FORTH  defid:WORD
   1 00068CF0  E92D4000  _-@_      | fastprolog lr only
   2 00068CF4  E3A0C001  ____      | reg:w =0x1
   3 00068CF8  E1A0551C  __U_      | LSHIFT: top = reg:w shifted
                                     left with top
   4 00068CFC  E8BD8000  ____      | fastnext lr only
```

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

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

name:>BITC  wid:FORTH  inline:2ops  defid:CODE
   1 00068D30  E3A0C001  ____      | reg:w =0x1
   2 00068D34  E1A0551C  __U_      | LSHIFT: top = reg:w shifted
                                     left with top
   3 00068D38  E12FFF1E  _/__      | codenext
```

The word **>BIT** coded in hi-level wabiPi4 takes about 5c to run. The version using **CODE:** and **INLINABLE** runs in 1 cycle.

Please note:
The method above will only works if *inlining* is switched 'on'. Without inlining, hi-level Forth cannot create a primitive, and the primitives created using **CODE:** will not function anymore. And your program will crash. If inlining is switched off in your program, use **FORCEINLINE** and **INLINEASBEFORE** before and after the definition to avoid problems.

An alternative way to speed up things is to use **IMMEDIATE** and **POSTPONED** in a definition. This may results in a performance as fast as inlinable primitives made with the method above. And halting inlining will not stop your program from working.

An example:

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

Could also be written as:

```
: >BITp ( n -- 2^n )
postpone 1 postpone swap postpone lshift ; immediate
```

BUT every advantage has its disadvantage...

This version does not depend on the correct setting of INLINING, but cannot be used outside a definition. But the resulting code is as fast as the version with CODE:.

Thumb assembler

The CPU in the Raspberry pi4 is capable of running in thumb-execution-mode. Thumb-code is efficient and fast, and was released by the ARM-corporation in 1994 so that ARM processors could also be used in small systems with limited memory.
It is interesting to experiment a bit with THUMB, at least I think so. For this a THUMB assembler for wabiPi4 is available. The assembler supports THUMB-1 for **M0** processors of ARM. The assembler can be found at the wabiForth GitHub site under 'examples'.

The actual handling of the switch to thumb-execution mode and back is handled by wabiPi4 when it encounters the word **THUMB:**.

THUMB: (--): starts a definition using the thumb-opcode set. Please note that thumb cannot be mixed with hi-level forth words, as these are written in ARM32. A thumb primitive is ended with the normal **';' (SEMICOLON).** WabiPi4 handles the switch to thumb and switching back to ARM32 mode after the word has finished.

Experimental phase:
Everything with thumb is experimental and it might be that in future things are updated or changed. For instance presently the switch to and from thumb-mode incorporate two **ISB**'s (instruction barriers) to add extra stability. This works fine but consumes about 30 or so extra CPU-cycles. More efficient solutions might be found in future.

You have to **load the Thumb-assembler** before you can use thumb-opcodes.

Here a simple routine which adds the sum between 0 and n. So if n=10, the sum calculated is 1+2+3+4+5+6+7+8+9+10=55.

```
lbl[ thumb: intsum ( n -- total(0-n) )
\ calcs the total of number 0 to n on stack
[
    r0, top, movs,      \ mov top into r0 - set flags
    beq, label1:        \ skip to exit on top=0
    top, 0 movs,
label0:
    top, top, r0, adds, \ add r0 to top
    r0, r0, 1 subs,     \ sub 1 from r0
    bne, label0:        \ jump on not zero to label0:
label1:
] ;
```

It shows the use of labels, including clearing the label-table; the use of THUMB:, the format of the opcodes and conditional branching.

If you do not have the thumb-assembler running, the following will write the opcodes as hex codes and function exactly the same.

```
hex
thumb: intsum ( n -- total(n-n) )
[  0028 16bopcode,
   D003 16bopcode,
   2500 16bopcode,
   182D 16bopcode,
   1E40 16bopcode,
   D1FC 16bopcode, ]
; decimal
```

5CH code

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 assembly 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 wabiPi4. So definitions which use a mix of Forth and assembly usually 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
```

Interrupts

The raspberry pi4b uses the GIC-400 (global interrupt controller=GIC) from the ARM-corporation to aid in handling interrupts. The GIC-400 has the reputation of being complicated in use. But this is limited to the initial setup, which is done by the wabiPi4 system.
Once running, it is relatively easy to handle interrupts with the GIC-400. Learning how is not trivial and takes a bit of perseverance, but entirely possible. I did! ARM has excellent documentation for the GIC-400 in which the functionality is described in detail.

The wabiPi4 system contains constants pointing to the addresses of the most important GIC-registers which control the use of the GIC-400 functionality. As wabiPi4 runs in (secure) supervisor operating-mode, the full functionality of the GIC-400 is available to the user.

THE INTERRUPTS OF THE WABIPI4 SYSTEM

There is only 1 interrupt active in the wabiPi4 system, and that is an user-requested **ABORT**. If the user wants to abort a running program this can be done by pressing 'fn', 'ctrl' and 'backspace' at the same time. This is recognised by the input-handler running on core2. If recognised, core2 issues a Software Generated Interrupt (SGI) to the GIC with core0 as destination. The resulting interrupt is forwarded to the FIQ-handler of core0. The FIQ-handler checks what the origin is of the interrupt, and if the ABORT is user-requested, an **ABORT** is issued.

Thus core0 is running without any interruption until the user requests an interrupt.

This has several advantages:
- execution of a program is as fast as possible
- core0 is always available to perform time-critical tasks. For instance checking a GPIO-port or anything else which has to be checked regularly
- measurements of time-periods are as reliable as technically possible

But sometimes having a regular interrupt available can be beneficial. For instance if you want to use WFI (Wait For Interrupt) for an energy-conserving pause of execution. For this reason it is possible to switch on an interrupt, by default at 500 Hz. Presently, the only use of this interrupt is for WFI. But in future might be used to regularly call a wabiPi4 routine.

The words used for this functionality are **FIQDIRECT** or **FIQBYPASS** (the names refer to terminology of the GIC-400 interrupt controller of the pi4). Calling **FIQBYPASS** switches on a 500 Hz interrupt, calling **FIQDIRECT** disables the interrupt.

FIQCOUNTER points to a variable which counts up with each call of the FIQhandler of core0. It can be used to check if interrupts are reaching the FIQhandler.

FIQBYPASS (--): switches on a 500 Hz interrupt, handled by the FIQhandler of core0. It is a slow word and takes around 13700 cycles to complete.

FIQDIRECT (--): switches the interrupt off again. It is a slow word and takes around 13700 cycles to complete. No interrupts of core0 is the default situation.

FIQCOUNTER (-- addr): points to an address which counts up with each call of the FIQhandler of core0.

Example:

Executing the following will print out the frequency of calls to the FIQhandler of core0. It returns 0 after a call to **FIQDIRECT**, and 500 after **FIQBYPASS** is called.

```
fiqcounter @ wait fiqcounter @ swap - .
```

AN ENERGY-SAVING WAIT-LOOP USING WFI OR WFE

The ARM opcode **WFI** waits for a next interrupt (hence the name: **W**ait **F**or **I**nterrupt) to occur in a system (any interrupt is okay) and then simply continues with the next opcode. The interesting feature of **WFI** is that it switches off the CPU-core while waiting but retains memory and status.

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 an 2-core ARM-processor running @ 1800 MHz, to run for 20 hours on a small battery. **WFI** ensures that for most of the time the CPU actually isn't running at all.

In wabiPi4 WFI normally does not function. The reason is easy to understand: waiting for an interrupt takes forever on wabiPi4 as core0 runs interrupt-free. One solution is using **WFE. WFE** does exactly the same as WFI, only it waits for an event, rather than an interrupt.
What an event is, is defined by the programmer. In wabiPi4 there is a global event-stream defined with 44.1 Khz events. This is the basis for sound-synthesis in core2. And because it is global, it can be used to activate **WFE.**

The following routine does the same as the example in the **DO...LOOP** - **FOR...NEXT** chapter. But by adding WFE or WFI it becomes energy-efficient.

A solution using WFE:

```
FOR_PRE
: KEY_TIMEOUT
  441000 for
     wfe
     key? 0= while
  next
           ." time-out"
     else
           key emit unloop
     then
;
```

A solution using WFI:

Another possibility is to switch on interrupts for core0 with **FIQBYPASS** (see above for an explanation). Interrupts are switched off again using **FIQDIRECT.**

Please note that the interrupts must be switched on when using the word containing **WFI,** or the program will wait forever. A simple user-interrupt will alleviate the situation if this happens. The status of the interrupts during compilation is irrelevant.

```
FOR_PRE
: KEY_TIMEOUT fiqbypass
  5000 for
  wfi
     key? 0= while
  next
           ." time-out"
     else
           key emit unloop
     then
fiqdirect ;
```

This example switches on the interrupts for core0 and starts waiting for input. After input or time-out it switches of the interrupts for core0 again.

Cache-handling

CACHE-HANDLING

One of the specific characteristics of the Forth programming language is that programming extends the language itself, rather than create a separate program. For a compiling Forth, like wabiPi4, this means that the act of programming changes a running program. This can only be done reliably if the handling of the caches is done very carefully and very conservative. All cache-handling is taken care of by wabiPi4. 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**, wabiPi4 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 wabiPi4 usually focusses on performance, in case of the setup of the cache-system reliability was prime. Therefore it was decided to run wabiPi4 with full coherence between the different cores. As such this is a bit slower than without coherence. But the big advantage is that wabiPi4 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 (--): The word **CLEANCACHE** makes cache invalidation easy. It 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.

"There are only two hard things in computer science: cache invalidation, naming things and off_by_one errors"

after Phil Karlton

The benefit of the **CLEANCACHE** is that **it creates a known starting-condition**. For instance 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 and memory is created.

ARM-VideoCore mailbox

A **Raspberry Pi 4b has 2 main processing units**, the ARM-processor and the Video-Core. These two communicate with each other through a feature called 'MAILBOX'.
This mailbox is not to be confused with the mailboxes used for communication between the 4 cores of the ARM-processor. The original Raspberry Pi 1 had a single core processor, and so there was no confusion.
But from the Pi 2 onwards, with multiple cores, the same term is used for two different functionalities. In this chapter we are dealing with the mailbox for communication between the ARM-processor and the Video-Core.

The mailbox-system technically supports communication via 16 bi-directional channels between the ARM-processor and the video-core. However, on the Raspberry Pi only the ARM-processor initiates communication, and always on channel 8. The ARM-processor does a 'process-request', usually accompanied by some data. The VideoCore then reacts to these 'process-requests' by performing certain actions, or giving information back or both.

The communication is done via TAGs. For the Raspberry 4b there are 68 documented valid TAGs, and at least 10 undocumented ones.

Each TAG has a specific function. These cover topics like firmware- and board-version, available DMA-channels, clocks, temperature and LEDs. In addition you can change video-settings and control the cursor.
Not all **TAGs** are created equal, most return useful information or do useful tasks, but several do not. And the documentation of the TAGs is far from perfect. Missing, wrong and outdated details fill the Raspberry data-sheets when looking at TAGs. WabiPi4 obviously supports all known **TAGs**, useful or not.

There are lies, damned lies and data-sheets...

after Mark Twain

WabiPi4 presently has no build-in support for multiple **TAGs** in a single process-request. If you want to, you can develop your own routine for this. It can be done using WabiPi4.

Documentation from the Raspberry-organisation is extensive, but slightly confusing and not always in concordance with what a TAG actually seems to do. If you are interested, get your hands on the documentation and study it. For help you can consult the help-forums of the Raspberry organisation.

The tags have the tag-number as name. These tag-numbers are the same as the numbers used in the Raspberry documentation.

TAG00001 (-- firmware_revision): returns firmware revision of board wabiPi4 is running on.

TAG10001 (-- board_model): returns model-number of board

TAG10002 (-- board_revision): returns board-revision code

TAG10003 (-- mac_addr=6*char): returns mac-address of board as 6 bytes with the most significant on top of the stack

TAG10004 (-- serial_number): returns a 64bit serial number, the serial_number is unique for each individual board

TAG10005 (-- base_addr, size): returns the base address and size of the memory ***below*** the video-core.

TAG10006 (-- base_addr, size): returns the base address and size of the video-core memory.

TAG10007 (-- 12*(parent_clock, clock)): returns the clocks in the Raspberry with their parent clock. Not very useful, see for yourself.

TAG50001 (-- addr, len): returns the address and length of the command-line. TYPE can be used to print the string.

TAG60001 (-- mask): returns a mask specifying which DMA channels (0-15) are available and which are in use. The bits 15:0 each represent a DMA-channel. A 0x0 means that the corresponding channel is not available for use by the programmer.

TAG20001 (device_id -- device_id, power-state): returns the power-state for the given device.

Unique device_IDs (according to the Raspberry documentation):
  0x00000000: SD Card
  0x00000001: UART0
  0x00000002: UART1
  0x00000003: USB HCD
  0x00000004: I2C0
  0x00000005: I2C1
  0x00000006: I2C2
  0x00000007: SPI
  0x00000008: CCP2TX
  0x00000009: Unknown (RPi4)
  0x0000000A: Unknown (RPi4)

This is one of the TAGs which suffers from incomplete documentation. For instance for many other device_IDs it is returned that a powered or unpowered device exists with that device_ID. Which seems logical as the pi4b has far more devices than the pi3b+. It is presently not documented which of these unspecified devices is what.

Power-state returned:
  bit 0: 0=off, 1=on
  bit 1: 0=device exists, 1=device does not exist

The returned power-state is also documented incompletely. For instance: for the UART1 (the UART used for wabiPi4 communication) it is returned that it is unpowered. Also during actual use of that UART.

TAG28001 (device_id, wanted_state -- device_id, returned_state): For the given device switch the power on or off. Returns the device_id and the result.

wanted state:
  bit 0: 0=off, 1=on
  bit 1: 0=do not wait, 1=wait for the power to become stable

returned state:
  bit 0: 0=off, 1=on
  bit 1: 0=device exists, 1=device does not exist

TAG20002 (device_id -- device_id, enable_waittime): For the given device, returns the required wait-time in microseconds for the power-supply to be stable after enabling the power. If a 0x0 is returned it signifies that the device does not exist. Usually a time of 1 is returned, for UART0 a time of 4000 microseconds is returned.

TAG30001 (clock_id -- clock_id, returned_state): For the given clock_ID returns the present state of that clock

unique clock IDs:
- 0x000000000: reserved
- 0x000000001: EMMC
- 0x000000002: UART
- 0x000000003: ARM
- 0x000000004: CORE
- 0x000000005: V3D
- 0x000000006: H264
- 0x000000007: ISP
- 0x000000008: SDRAM
- 0x000000009: PIXEL
- 0x00000000A: PWM
- 0x00000000B: HEVC
- 0x00000000C: EMMC2
- 0x00000000D: M2MC
- 0x00000000E: PIXEL_BVB

state:
- bit 0: 0=off, 1=on
- bit 1: 0=clock exists, 1=clock does not exist

TAG38001 (clock_id, state -- clock_id, returned_state): For the given clock_ID sets the state of that clock. The wanted state is 0=off/1=on, the returned state is as for **TAG30001**.

TAG30002 (clock_id -- clock_id, rate): for the given clock_ID returns the set rate, in Hz, for that clock. Returns 0x0 for a non-existent clock. The clock_rate is also returned for disabled clocks.

The following 3 LED-related **TAGS** function differently than documented. This might be because I use an old version of the firmware. The description below is as close as I can get to the actual way the 3 TAGs work.

TAG30041 (-- powerLED_state, statusLED_state): returns the state of the two onboard LEDs: red power-LED and green status-LED.

pin_state:
- bit 0: 0=on, 1=off << please note the reverse logic

This is different from the documentation, which states that the pin-number of the LED is returned with the status. At least on the pi4b this TAG does not return the pin-numbers.

TAG34041 (wanted_state, pin-number -- pin-number, state): checks if the wanted state is possible for the two onboard LEDs: red power-LED and green status-LED.

This **TAG** functions differently than described in the documentation. The states are

pin_state:

```
  for 130=red power LED:    0=on, 1=off
  for 42=green status LED:  0=off, 1=on
```

TAG38041 (wanted_state, pin-number -- 0x0, state): sets the specified pin-number to the requested state. Returns a 0x0 and the resulting state. Please note that the logic for both LEDs differs!

pin-number:

```
   42: green status LED
  130: red power LED
```

pin_state:

```
  for 130=red power LED:    0=on, 1=off
  for 42=green status LED:  0=off, 1=on
```

TAG30047 (clock_id -- clock_id, measured_rate_Hz): takes a measurement of the specified clock_id and returns the measurement in Hz. See previously for an overview of available clock_id's.

TAG38002 (clock_id, wanted_rate, turbo_flag -- clock_id, set_rate): For the specified clock_id set the new rate to the wanted_rate. Setting the turbo_flag to 1 means that nothing else happens than the rate. Setting the flag to 0x0 means that the CPU can change other things like voltage as a result of the new rate. It is good to notice that internal limits and other implementation issues mean that the clock can stay unchanged. For instance setting the ARM-clock does not function. But changing the rate of the CORE-clock does function.

TAG30004 (clock_id -- clock_id, max_rate): for the given clock_id, return the maximum rate allowed by the system.

TAG30007 (clock_id -- clock_id, min_rate): for the given clock_id, return the minimum rate allowed by the system.

TAG30009 (0x0 -- 0x0, turbo_state): return the state of the turbo_flag. 0=non_turbo/1=turbo

TAG38009 (0x0, wanted_turbo_state -- 0x0, turbo_state): set the state of the turbo_flag, where 0=non_turbo/1=turbo. Again a TAG where the actual function and the documentation do not match. Under certain circumstances the TAG returns 0x8000000 instead of the turbo-state.

TAG30003 (voltage_id -- voltage_id, voltage): for the given voltage_id returns the set voltage in microvolt. A voltage of -1 is returned if a voltage_id does not exist. For voltage_id=0, 0x80000000 is returned.

Unique voltage IDs:

```
  0x0: reserved
  0x1: core
  0x2: SDRAM_C
  0x3: SDRAM_P
  0x4: SDRAM_I
```

TAG38003 (voltage_id, wanted_voltage -- voltage_id, voltage): for the given voltage_Id set the voltage to the wanted voltage.

The wanted voltage can be specified in three ways, depending on the value:

- **value <= 16**: relative to platform-specific typical voltage, in 25mV steps -> for instance a value=10 raises the voltage by 250mV above the typical voltage.
- **16 < value < 500000**: relative to platform-specific typical voltage, in microvolts -> for instance a value=100000 raises the voltage by 0.1V above the default voltage
- **value >= 500000**: absolute voltage in microvolts. -> for instance a value of 1000000 sets the voltage to 1.0V.

TAG30005 (voltage_id -- voltage_id, max_voltage): returns the maximum voltage in microvolt for the given voltage_id.

TAG30008 (voltage_id -- voltage_id, min_voltage): returns the minimum voltage in microvolt for the given voltage_id.

TAG30006 (0x0 - 0x0, temperature): return the actual temperature of the SOC. The returned temperature is in mili-degrees Celsius, but is always rounded to whole degrees Celsius...

TAG3000A (0x0 - 0x0, max_temperature): returns the maximum temperature of the SOC. The returned temperature is in mili-degrees Celsius but is always rounded to whole degrees. Overclocking may be disabled above this temperature by the SOC.

TAG3000C (size, alignment, flags -- handle): Allocates a contiguous block of memory on the GPU. Size and alignment are in bytes. A non-zero value for the handle signifies success.

Flags contain:

- **bit0: MEM_FLAG_DISCARDABLE:** if true then can be resized to 0x0 at any time. Intended for cached data.
- **bit1:** unused
- **bit2-3:**
  - 0x0: **MEM_FLAG_NORMAL** -> normal allocating memory - do not use from ARM
  - 0x1: **MEM_FLAG_DIRECT** -> 0xC alias -> uncached
  - 0x2: **MEM_FLAG_COHERENT** -> 0x8 alias -> non-allocating in L2-cache but coherent
  - 0x3: **MEM_FLAG_L1_NONALLOCATING** -> allocating in L2, not allocating in L1
- **bit4: MEM_FLAG_ZERO** -> if true than initialise buffer to all zero's
- **bit5: MEM_FLAG_NO_INIT** -> if true -> don't initialise, otherwise initialise to all one's
- **bit6: MEM_FLAG_HINT_PERMALOCK** -> if true hint to MMU of VC that buffer is likely to be locked for long periods of time.
- **other bits:** unused

TAG3000D (handle -- bus_address): lock buffer in place, and return a bus address. Must be done before memory can be accessed. Bus-address <> 0 signals success.

TAG3000E (handle -- status): unlock buffer. The contents is retained, but may be moved. The buffer must be locked before next use. A returned status = 0x0 signals success.

TAG3000F (handle -- status): release the memory buffer. A returned status = 0x0 signals success.

TAG30010 (bus_addr, r0, r1, r2, r3, r4, r5 -- r0): calls the function at given bus_address with the values moved to the registers r0-r5. It blocks until the call completes. The (GPU) instruction cache is flushed. Setting bit:0 of the bus_address will suppress the instruction_cache flush. This is safe if the buffer hasn't changed since last execution.
This must be one of the most exiting TAGs in existence! An extra processor, which can be called in parallel to the ARM-processor. And if I tell you that this processor is a vector processor which can process 16 data-fields in parallel, than maybe you understand my excitement. Disadvantage is that the architecture of this processor is not documented, but fortunately most of the specs have been reverse engineered.

TAG30014 (disp_manx_handle -- status:0=success, mem_handle): Gets the disp_mem_handle associated with a created disp_manx resource. This resource can be locked and the memory directly written from the ARM to avoid having to copy image data into the GPU. This TAG basically allows a block of memory outside of the GPU to be part of the operations of the GPU. Thus avoiding the copy of a block of memory to the GPU a lot of times per second. The details of this TAG are somewhat nebulous.

TAG30020 (block_number -- block_number, addr, length): Get EDID-block - returns the block_number, an address and the length of the data at the address which is always 132 bytes. The first 4 bytes are the status, the rest are the 128 bytes of the EDID-block. Status=0x0 is success. Keep getting blocks (starting at 0x0) till the status<>0x0.

SCREEN SETUP WITH TAGS

All TAGs related to the setup of a screen in the GPU (frame_buffers etc.) have to be used in 1 operation. This is a major complication of the previous situation where one could use single TAGs till the screen was initiated. The main problem is that the error-reporting is much more unclear then previously. If you get an error back, there is no way of knowing which of the TAGs causes the error.
WabiPi4 handles the single operation with several TAGs for the screen setup for the user. It does presently not directly support combining several TAGs into 1 action from WabiPi4. If you need that functionality, you have to program it yourself, which obviously is entirely possible.

TAG40001 (alignment_bytes -- base_addr, frame_size) allocates a frame buffer in the GPU memory. The size of the buffer is calculated by the GPU based on the size of the virtual and physical screens in pixel height and width, and the colour-depth (ie 1, 2, 4, 8, 16 or 32 bits). This is why the setup has to be done in 1 operation, as other TAGs are used for that information.

TAG48001 (--): release the frame-buffer

TAG40002 (0=off/1=on -- 0=off/1=on): blank the screen

TAG40003 (-- width, height): get physical width and height of the display. Please note that the physical width and height can differ from the size of the screen send to the monitor.

TAG44003 (width, height -- width, height): test if the requested width and height for the display are possible. If possible, the requested width and height are returned. This TAG tests, but does not change anything.

TAG48003 (width, height -- width, height): set physical width and height for the display.

TAG40004 (-- width, height): get virtual width and height of the display. Please note that the virtual buffer size is the portion of buffer that is sent to the display device, not the resolution of the buffer itself. The virtual buffer size may be smaller than the allocated buffer size in order to implement efficient scrolling and panning.

TAG44004 (width, height -- width, height): tests setting virtual width and height of the display. If the requested width and height are possible, these are returned. This TAG tests, but does not change anything.

TAG48004 (width, height -- width, height): set virtual width and height for the display.

TAG40005 (-- depth): get depth in bits per pixel. Can be 1, 2, 4, 8, 16 or 32.

TAG44005 (depth -- depth): test setting the depth in bits per pixel. Can be 1, 2, 4, 8, 16 or 32. If the returned depth is the same as the input than the test was successful. This TAG does not change any settings.

TAG48005 (depth -- depth): set depth in bits per pixel. Can be 1, 2, 4, 8, 16 or 32. The returned depth might be different from the requested depth.

TAG40006 (-- pixel-order): get pixel-order. 0x0=BGR/0x1=RGB.

TAG44006 (pixel-order -- pixel-order): test pixel-order. If the returned pixel-order is the same as the input then the test was successful. This TAG does not change any settings.

TAG48006 (pixel-order -- pixel-order) set pixel-order. The returned pixel-order might differ from the requested pixel-order.

TAG40007 (-- alpha-state): get alpha-state.

Alpha-state:
- 0x0: alpha channel enabled (0 = fully opaque)
- 0x1: alpha channel reversed (0 = fully transparent)
- 0x2: alpha channel ignored

TAG44007 (alpha-state -- alpha-state): test alpha-state. If the returned alpha-state is the same as the requested state then the test was successful. This TAG does not change any settings.

TAG48007 (alpha-state -- alpha-state): set the alpha-state. The returned alpha-state might differ from the requested alpha-state.

TAG40008 (-- pitch): get pitch in bytes per line.

TAG40009 (-- x_pixels, y_pixels): get virtual offset in pixels. Can be a positive or negative number.

TAG44009 (x_pixels, y_pixels -- x_pixels, y_pixels): test virtual offset in pixels. Can be a positive or negative number. If the returned values are the same as the requested values then the test was successful.

TAG48009 (x_pixels, y_pixels -- x_pixels, y_pixels): set virtual offset in pixels. Can be a positive or negative number. The returned values can differ from the requested values. Useful for panning and scrolling.

TAG4000A (-- top, bottom, left, right): get overscan, all in pixels.

TAG4400A (top, bottom, left, right -- top, bottom, left, right): test overscan, all in pixels. If the returned values are equal to the requested values then the test was successful.

TAG4800A (top, bottom, left, right -- top, bottom, left, right): set overscan, all in pixels. The returned values can differ from the requested values.

TAG4000B (-- 256*32bit_RGB_value): get palet. If no palte is defined, 2 32bit values will appear on stack, without any special meaning.

Cortex-a72

Giving the programmer **access to the processor**, the Cortex-a72, is a specific focus of wabiPi4. WabiPi4 contains a selection of words to help with this.

For more information see the ARMv8 and Cortex-a72 documentation.

MEMORY ATTRIBUTES

The Cortex-a72 processor can only function efficiently if it knows how to handle the different parts of the memory. Attributes like 'cachable' or not, mixed memory or data-only memory or device-memory etc. This is done by filling a page-table with the attributes for each memory-block.

With the word **.PAGETABLE** you can print out the page-table as it is presently in use. WabiPi4 uses the simplest possible model compatible with efficient execution of Forth.
Also see **CPU_PARWRITE** and **CPU_PARRREAD**, which show the attributes for a specific memory-address.

.PAGETABLE (--): prints an overview of the page-table as in use by the Cortex-a72 processor.

ARM-PROCESSOR ID, STATUS AND INFO REGISTERS

The ARM-processor contains a wide selection of registers which return information describing the specific features of processor or statuses of functional blocks of the processor. A selection of these registers are directly supported by wabiPi4.

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 - which gives top level information about the debug system. The debug system is not directly supported by wabiPi4, but is enabled.

CPU_CTR (-- n): returns content of CPU-register CTR = cache-type-register
CPU_L2CTLR (-- n): returns content of L2 Control Register

CPU_CCSIDR (0, 1 or 2 -- n): returns content of the Cache Size ID Register. The input specifies the cache: 0x0=L1 data-cache, 0x1=L1 instruction-cache, 0x2=L2 unified cache. Other input-values result in a return of true. See the ARM-documentation for interpretation of the returned values.

CPU_CBAR (-- n): returns content of Configuration Base Address Register which is base-address of the GIC-400 registers. Returns 0xFF84000.

CPU_PMCR (-- n): returns content of CPU-register PMCR = performance monitors control register.
CPU_PMCEID1 (-- n): returns content of CPU-register PMCEID0 - aka the Performance Monitors Common Event Identification register 0. It describes which performance monitors are available in the CPU.
CPU_PMCEID1 (-- n): ditto for PMCEID1
CPU_PMCCFILTR (-- n) returns content of CPU-register PCCFILTR

CPU_CNTFRQ (-- 5644800): returns content of CPU-register CNTFRQ which holds the actual frequency of the physical, virtual and core counters - returns 5644800. This register has a pure informational character, changing the content does *not* change the frequency.

CPU_CPSR (-- n): returns content of CPU-register CPSR = current processor state register
CPU_SCTLR (-- n): returns content of CPU-register SCTLR = system control register
CPU_SCR (-- n): returns content of CPU-register SCR = security configuration register
CPU_SDCR (-- n): returns content of CPU-register SDCR = secure debug config registers

CPU_RMR32 (--): warm-reset request register, resets the CPU and goes to Aarch32 mode.
CPU_RMR64 (--): warm-reset request register, resets the CPU and goes to Aarch64 mode.
Theoretically, this word allows the development of a system running in Aarch64 mode. For this to function the programmer points the vector at address 0x0 to the start of a valid Aarch64 routine. **This word is intended for experimentation and testing. (**You know: messing around with a processor and trying to get something to run. What larks!!)

CPU_FPSID (-- n): returns content of CPU-register FPSID which describes the floating point processor features
CPU_FPSCR (-- n): returns content of CPU-register FPSCR = Floating-Point Status and Control Register - among other useful information, bit3 and bit2 give feedback on the over- or underflow of the last executed floating point instruction.

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

CPU_MAIR0 (-- n): returns content of CPU-register MAIR0 = memory attribute indirection register 0. This register is the same as the PRRR register which is used by wabiPi4. See the ARM documentation for more details.

CPU_MAIR1 (-- n): returns content of CPU-register MAIR1 = memory attribute indirection register 1. In wabiPi4 this register is set but not in actual use.

The ID_ISARn registers of the ARM-processor describe the available features of the specific processor wabiPi4 is running on. 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

ANSI-required documentation

The ANSI-standard specifies which implementation defined items must be documented. See section 4.1.1 of the ANSI-standard.

REQUIRED CATEGORY OF DOCUMENTATION	IMPLEMENTATION
aligned address requirements (**3.1.3.3 Addresses**)	the system does not rely on alignment for integer data, but does so for floating point data and ARM-opcodes - alignment is assured during normal operation.
behavior of **6.1.1320 EMIT** for non-graphic characters	**EMIT** handles 8 bit characters for the screen and 7 bits for the UART. Characters below 32 are handled in accordance with the ASCII standard. A bell sound is not sounded.
character editing of **6.1.0695 ACCEPT** and **6.2.1390 EXPECT**	**ACCEPT** follows the standard. In addition entry can be stopped by pressing 'ESC'. In that case a string with a length of 1 and 27 as first character is returned. **EXPECT** follows the standard.
character set (3.1.2 Character types, 6.1.1320 **EMIT**, 6.1.1750 **KEY**)	The system uses ASCII for characters below 128. For the screen only, character values between 128 and 255 are printed in accordance with the definition of the character in the character table. The user can change any value in the character table.
character-aligned address requirements (**3.1.3.3 Addresses**)	none
character-set-extensions matching characteristics (**3.4.2 Finding definition names**)	Character-set-extensions are not supported. Finding definitions is exclusively done using ASCII characters.
conditions under which control characters match a space delimiter (**3.4.1.1 Delimiters**)	The control-characters 9, 10, 11, 12 and 13 match a space delimiter during parsing.
format of the control-flow stack (**3.2.3.2 Control-flow stack**)	The return-stack is used as control-flow stack. Addresses and optionally flags denoting certain kinds of control-flow are used.
conversion of digits larger than thirty-five **(3.2.1.2 Digit conversion)**	Setting **BASE** to a value larger than 35 results in an **ABORT**, an error-message, and **BASE** is set to decimal.
display after input terminates in 6.1.0695 **ACCEPT** and 6.2.1390 **EXPECT**	**ACCEPT** and **EXPECT** show the characters as entered by the user. After termination of input the characters are kept as is.
exception abort sequence (as in 6.1.0680 **ABORT"**)	**ABORT"** prints a message and then aborts. The user is put back into interpreting mode in **QUIT**.
input line terminator (**3.2.4.1 User input device**)	Input line terminator: **CR**, **LF** and **CR/LF** are all accepted during input. **LF** is used during output.

REQUIRED CATEGORY OF DOCUMENTATION	IMPLEMENTATION
maximum size of a counted string, in characters (**3.1.3.4 Counted strings, 6.1.2450 WORD**)	4294967295 characters (this is longer than the available memory)
maximum size of a parsed string (**3.4.1 Parsing**)	1024 characters.
maximum size of a definition name, in characters (**3.3.1.2 Definition names**)	32 characters.
maximum string length for **6.1.1345 ENVIRONMENT?**, in characters	1024 characters.
method of selecting **3.2.4.1 User input device**	only 1 device supported
method of selecting **3.2.4.2 User output device**	by default the UART and the screen are used in parallel for output. The UART can be switched on/off.
methods of dictionary compilation (**3.3 The Forth dictionary**)	Dictionary compilation method: subroutine threaded, with inlining of primitives where beneficial, and compounding.
number of bits in one address unit (**3.1.3.3 Addresses**	32 bits
number representation and arithmetic (**3.2.1.1 Internal number representation**)	Numbers are internally represented with the least significant byte first, numbers can be 32 or 64 bits long.
ranges for *n*, *+n*, *u*, *d*, *+d*, and *ud* (**3.1.3 Single-cell types**, **3.1.4 Cell-pair types**)	**n:** −2147483648 to 2147483647 **+n:** 0 to 2147483647 **u:** 0 to 4294967295 **d:** −9223372036854775808 to 9223372036854775807 **+d:** 0 to 9223372036854775807 **ud:** 0 to 18446744073709551615
read-only data-space regions (**3.3.3 Data space**)	All data-space regions have read/write characteristics
size of buffer at **6.1.2450 WORD (3.3.3.6 Other transient regions)**	1024 characters
size of one cell in address units (**3.1.3 Single-cell types**)	4
size of one character in address units (**3.1.2 Character types**)	1
size of the keyboard terminal input buffer (**3.3.3.5 Input buffers**)	1023 characters – the UART has a circular receive buffer of 2097152 characters
size of the pictured numeric output string buffer (**3.3.3.6 Other transient regions**)	255 characters
size of the scratch area whose address is returned by **6.2.2000 PAD** (**3.3.3.6 Other transient regions**)	4095 characters

REQUIRED CATEGORY OF DOCUMENTATION	IMPLEMENTATION
system case-sensitivity characteristics (**3.4.2 Finding definition names**)	WabiPi4 is case-insensitive.
system prompt (**3.4 The Forth text interpreter, 6.1.2050 QUIT**)	The depth of each of the three user-stacks (data, float, return) is printed after each linefeed in QUIT. like this: **(0 0 0) ok** As support for efficient debugging, some negative depth of the stacks is possible.
type of division rounding (**3.2.2.1 Integer division, 6.1.0100 */, 6.1.0110 */MOD, 6.1.0230 /, 6.1.0240 /MOD, 6.1.1890 MOD)**	Division is symmetric with truncation as rounding-method
values of **6.1.2250 STATE** when true	**1**
values returned after arithmetic overflow (**3.2.2.2 Other integer operations**)	After an overflow, usually the lower 32 bits of the calculation are returned.
Whether the current definition can be found after **6.1.1250 DOES>** (**6.1.0450 :**)	A definition can only be found after successful execution of a semicolon.

AMBIGUOUS CONDITIONS – SYSTEM ACTION

REQUIRED CATEGORY OF DOCUMENTATION	SYSTEM ACTION
a name is neither a valid definition name nor a valid number during text interpretation (3.4 The Forth text interpreter)	The system aborts with a message specifying which word was unknown.
a definition name exceeded the maximum length allowed (3.3.1.2 Definition names)	The system aborts with a message specifying which name was too long
addressing a region not listed in **3.3.3 Data Space**	All data-regions in the system are available to the user. If non-existent memory is addressed, a hardware-generated exception occurs and a message is printed specifying the illegal address causing the exception.
argument type incompatible with specified input parameter, e.g., passing a flag to a word expecting an n (3.1 Data types)	WabiPi4 does not check for type-compatibility of input-arguments with the following exceptions: 1. for **TO** & **+TO** wabiPi4 checks if the name after **TO/+TO** is a **VALUE** or a **2VALUE** 2. for **$VAR** related words, wabiPi4 checks if the specified string is actually defined as a **$VAR.**

REQUIRED CATEGORY OF DOCUMENTATION	SYSTEM ACTION
attempting to obtain the execution token, (e.g., with 6.1.0070 ', 6.1.1550 **FIND**, etc.) of a definition with undefined interpretation semantics	Execution-tokens for any definition can be obtained.
dividing by zero	The system aborts with a message specifying that a division by zero occured.
insufficient data-stack space or return-stack space (stack overflow)	The system allows for a limited amount of overflow. No checks are done.
insufficient space for loop-control parameters	No checks are done by the system. The stacks have space for 2048 cells each.
insufficient space in the dictionary	The maximum size of the dictionary is 983 MB. If there is no space left, the system will abort when trying to create a new definition. In addition there is 2863 MB data-space, addressed by **ALLOCATE** and **TEMPALLOCATE**. Both words do not allocate memory if the space is insufficient for the request and return a **true** as warning.
interpretating a word with undefined interpretation semantics	No checks are done by the system. Some words with undefined interpretation semantics run without adverse effects in interpreting mode.
modifying the contents of the input buffer or a string literal (3.3.3.4 Text-literal regions, 3.3.3.5 Input buffers);	By design, the system has read/write privileges to any memory-region.
overflow of a pictured numeric output string	This overflow will not happen under normal circumstances in the wabiPi4 system. No checks are done.
parsed string overflow	The system aborts with a message specifying the cause of the abort.
producing a result out of range, e.g., multiplication (using ∗) results in a value too big to be represented by a single-cell integer	No checks are done by the system. The result is most likely the lower 32 bits of the returned result, but this is not guaranteed.
reading from an empty data stack or return stack (stack underflow)	The system allows for (a limited amount of) negative depths with the system continuing to function. No specific checks are done by the wabiPi4-system.
unexpected end of input buffer, resulting in an attempt to use a zero-length string as a name	The system aborts with a message specifying that a name was expected.

SPECIFIC AMBIGUOUS CONDITIONS – SYSTEM ACTION

REQUIRED CATEGORY OF DOCUMENTATION	IMPLEMENTATION
>IN greater than size of input buffer (3.4.1 Parsing)	Under normal circumstances **>IN** cannot become larger than the buffer, and no checks are done by the system.
RECURSE appears after 6.1.1250 **DOES>**	The system will abort with a clarifying message.
Argument input source different than current input source for 6.2.2148 **RESTORE-INPUT**	**RESTORE-INPUT** returns a true flag on failure, no **ABORT** occurs.
data space containing definitions is de-allocated (3.3.3.2 Contiguous regions)	No checks are done by the system.
data space read/write with incorrect alignment (3.3.3.1 Address alignment)	Address-alignment is mandatory for floating-point related words and for ARM-opcodes. The system will abort with an exception if problems with these occur. Address-alignment is technically not needed for other values. Normal functioning of the system ensures alignment automatically.
data-space pointer not properly aligned (6.1.0150 ,, 6.1.0860 **C,**)	**COMMA** can handle unaligned addresses. However ARM-opcodes must be aligned, so if **COMMA** is used to write opcodes on unaligned addresses, executing the resulting code will result in an unrecognised opcode with a hardware generated prefetch-exception. The system will abort and print a message which address caused the exception.
less than u+2 stack items (6.2.2030 PICK, 6.2.2150 ROLL)	No checks on this are done by the system. Items below the lowest stack-position can be **PICK**'d or **ROLL**'d, and no error-message will be returned.
loop-control parameters not available	No checks on this are done by the system.
most recent definition does not have a name	Using a zero length string for a name result in an **ABORT** with a clarifying message.
name not defined by 6.2.2405 **VALUE** used by 6.2.2295 **TO**	The system aborts with a message specifying the offending name.
name not found (6.1.0070 ', 6.1.2033 **POSTPONE**, 6.1.2510 **[']**	The system aborts with a message specifying which is the offending name.
parameters are not of the same type (6.1.1240 **DO**, 6.2.0620 **?DO**, 6.2.2440 **WITHIN**)	No checks on this are done by the system.
6.1.2033 **POSTPONE** or 6.2.2530 **[COMPILE]** applied to 6.2.2295 **TO**	In wabiPi4 **TO** can be postponed and the containing definition can be used to put a value into a **VALUE**. **[COMPILE]** is not implemented in wabiPi4.

REQUIRED CATEGORY OF DOCUMENTATION	IMPLEMENTATION
string longer than a counted string returned by 6.1.2450 **WORD**	Technically and logically impossible in wabiPi4 as a counted string has a 32b index.
u greater than or equal to the number of bits in a cell (6.1.1805 **LSHIFT**, 6.1.2162 **RSHIFT**)	0x0 is returned
word not defined via 6.1.1000 **CREATE** (6.1.0550 **>BODY**, 6.1.1250 **DOES>**)	No checks on this are done by the system. In wabiPi4 the words **CREATE** and **DOES>** must be within the same definition.
words improperly used outside 6.1.0490 **<#** and 6.1.0040 **#>** (6.1.0030 **#**, 6.1.0050 **#S**, 6.1.1670 **HOLD**, 6.1.2210 **SIGN**)	The words **#**, **#S** and **HOLD** can be used outside of **<#** and **#>** without problems occurring. No spurious filling of the buffer occurs, but there are stack-effects.

OTHER SYSTEM DOCUMENTATION

REQUIRED CATEGORY OF DOCUMENTATION	IMPLEMENTATION
list of non-standard words using 6.2.2000 **PAD** (3.3.3.6 Other transient regions)	-- **PAD** is not used by any word in the system dictionary of wabiPi4
operator’s terminal facilities available	--
program data space available, in address units	**1'031'158'476** bytes
return stack space available, in cells	2 times 2048 cells - Wabipi4 technically separates the return-stack into 2 stacks, one for the CPU and one for the user, to optimize the branch-prediction system of the CPU. Both return-stacks are 2048 cells deep.
stack space available, in cells	2048 cells
system dictionary space required, in address units	The image of **290'432** bytes is expanded during the boot-process to a system dictionary with a size of **448'384** bytes. In addition the system also allocates **176'422'912** bytes for system functionality. For details see the chapter on the **memory-map** of the system.

Air Traffic Controller

The following game is largely complete but not yet a finished product. But parts might be helpful for topics like the use of PIXEL-windows, that's why I include it.

```
(*
 Air Traffic Controller
 Based on a game by Dave Mannering for Creative Computing 1978-1981

 This wabiPi4 version (C) 2024 - J.J. Hoekstra
 Requires wabiPi4 4.1.0 or higher
 32316 bytes 197 definitions
 The original version ran in 16K
*)

word# unused >|||>
forget atcshield : atcshield ;

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

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

12 constant origplane
 4 constant origcol

\ values ***********************************
50 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
 0 value score

\ 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
    transparent 2 >wincanvas
                2 wincls
          white 2 >winink
           true 2 winvisible                \ transparent

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

              2 0 win#>task#

    \ windows 1 - copyright etc
          false 1 winvisible

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

    \ window 3 - list of active planes
        230 496 3 makewin
          794 0 3 >winorig
       vlyellow 3 >wincanvas
                3 wincls
              4 3 >winrand
    transparent 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
    transparent 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
    transparent 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! ;               \ 0=prop/1=jet

: 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!   ( start p -- ) 10 16plane! ;

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

: #%flag@   ( p -- flag ) 15 plane@ ;           \ = p#%_flag
: #%flag!   ( n p -- ) 15 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

create pointstbl \ table for scoring-system - 82
     25 ,  26 ,  27 ,  28 ,  29 ,  30 ,  31 ,  32 ,  34 ,
     35 ,  37 ,  39 ,  41 ,  43 ,  45 ,  47 ,  49 ,  51 ,
     53 ,  55 ,  57 ,  59 ,  61 ,  63 ,  65 ,  67 ,  69 ,
     71 ,  73 ,  75 ,  78 ,  81 ,  84 ,  87 ,  90 ,  93 ,
     96 ,  99 , 102 , 105 , 109 , 113 , 117 , 121 , 125 ,
    129 , 134 , 139 , 144 , 150 , 156 , 162 , 169 , 176 ,
    184 , 192 , 201 , 211 , 221 , 232 , 244 , 256 , 270 ,
    285 , 301 , 319 , 338 , 358 , 380 , 404 , 430 , 458 ,
    489 , 523 , 560 , 600 , 645 , 692 , 745 , 810 , 885 ,
    975 ,

: [points] #steps pointstbl + @ ;

: (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 ; \ 5cval: 5 ascii > 32b value
: setseeds $>seed
    dup -2026418266 = if                    \ is '?????'
        drop randomize
    else
        dup rndmseed
    then ;
: origtbl@ ( orig ocol -- n ) swap origcol * [orig] + + sc@ ; \ sc@: signed c@

: (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
    transparent 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
    >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 0-5 for opdracht -> goto 3 - bs -> goto 1
\  3 -> 2/3 chars received - do opdracht, cr, goto 1 - bs -> goto 2

\ *****  ORDERS  *****
\ 1=A0-A5: height
\ 2=L1-L4: turn left
\ 3=R1-R4: turn right
\ 4=H: into holding-pattern at next nav point
\ 5=M: Maintain - stop climbing/descending and continue at present height
\ 6=P: Proceed - stop all ongoing course-changes and orders for turn at navaid and
        continue at present course
\ 7=S: show data on plane - especially for fuel
\ 8=[: on next nav turn to # << set #flag
\ 9=]: on next nav turn to % << set %flag

\ ********************

0 value is=
: input_handler ( key -- ) \ sets is= to 4 on jump to next

    dup 10 = is= 0 = and if
        drop 4 to is= exit                  \ is= = flag for jump to next 15s
    then

    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 newhght!
            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!
        2 rndm 0= if 1 i prop! then         \ =50% jets - used to be 25 and 33% jets
                     6 i height!
                     6 i newhght!
                       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
    0 to score
    ;
: .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" 6 spaces
    #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@ newhght@ 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" space  \ 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!                 \ << #TBD: put a zero there but than gives
         aborted landing message
            then
        then

        i stat@ 4 = if                      \ 4=permission to start prop_plane
            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                   \ no 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@ 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
    [points] 11 * +to score
;

: planedone? ( p -- f\t )                   \ checks for landing and for correct exit
         of plane
    >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
            [points] 5 * +to score exit
        else drop
            r@ deactivateplane
            ." BOUNDARY ERROR plane " r> .plchar cr false exit
        then
    then
    rdrop true
    ;
: checkplanes ( -- f/t )                    \ check for ??
    #plane 1 do
        i planeactive? if
            i planedone? 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
            \   i \ << huh??
                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
\   29 -> circle 45 left - goto 29 << = holding at navaid
\   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
    ;
: [@navbeaconl] ( x y -- t/f )              \ position 5, 8
    8 = swap 5 = and if true
    else false then ;

: [@navbeaconr] ( x y -- t/f )              \ position 18, 8
    8 = swap 18 = and if true
    else false then ;

: @navbeaconr? ( p -- )                     \ state=30
    >r r@ xy@ [@navbeaconr] if
        r@ leftdir                          \ start circling
        29 r@ stat!                         \ state engine to 29
    then rdrop ;

: @navbeaconl? ( p -- )                     \ state=30
    >r r@ xy@ [@navbeaconl] if
        r@ leftdir                          \ start circling
        29 r@ stat!                         \ state engine to 29
    then rdrop ;

: gotofield#froml ( p -- )
    >r r@ xy@ [@navbeaconl] if
        r@ #%flag@ 1 =
        if
            2 r@ direc!                     \ goto airfield%
            1 r@ stat!                      \ = simple proceed
            0 r@ #%flag!                    \ reset goto on navaid
        then
    then rdrop ;

: gotofield#fromr ( p -- )
    >r r@ xy@ [@navbeaconr] if
        r@ #%flag@ 1 =
        if
```

```
            6 r@ direc!                     \ goto airfield%
            1 r@ stat!                      \ = simple proceed
            0 r@ #%flag!                    \ reset goto on navaid
        then
    then rdrop ;

: gotofield%fromr ( p -- )
    >r r@ xy@ [@navbeaconr] if
        r@ #%flag@ 2 =
        if
            7 r@ direc!                     \ goto airfield%
            1 r@ stat!                      \ = simple proceed
            0 r@ #%flag!                    \ reset goto on navaid
        then
    then rdrop ;

: turnto#%? ( p -- ) \ checks for and does a turn at nav-beacon if #%flag set and at
        any beacon
    dup gotofield#froml
    dup gotofield#fromr
        gotofield%fromr ;

: 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@ @navbeaconr?
        r> @navbeaconl?             exit then \ start circling when at either *
    dup 29 = if drop r> leftdir     exit then \ continue circling
    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 turnto#%?                 \ check for turn at nav-beacon
                i adaptheight               \ #checked
            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
    setseeds
    gameinit win6
    ;
: .status ( p -- )
    cr s" status plane: " . cr
    ;
: 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 10 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 ." UNABLE - 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 IN PROGRESS" win6
        false exit
    then
```

```
    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               \ possibly print ROGER later
    true ;

: doopdracht ( -- )
    opd4plane stat@ 40 = if                 \ if landing -> no more changes possible
        exit
    then

    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                         \ turn left
        getal 1 5 within if
            getal 9 + planestat! then
        getal 0 = if
            30 planestat! then              \ hold at nav-beacon
        getal 5 = if                        \ at nav-beacon turn to airfield #
            1 opd4plane #%flag! then
        exit
    then

    opdracht 3 = if
        getal 1 5 within if
            getal 13 + planestat! then
        getal 0 = if
            1 planestat! then               \ proceed
        getal 5 = if                        \ at nav-beacon turn to airfield %
            2 opd4plane #%flag! then
        exit
    then

    opdracht 4 = if
        30 planestat! then                  \ hold=start circling at nav-beacon

    opdracht 5 7 within if
        1 planestat!                        \ proceed in present direction
        0 opd4plane #%flag!                 \ reset 'goto' airfield at nav
        opd4plane dup height@ swap newhght! \ stop changing height
    then

    opdracht 7 = if opd4plane .status then  \ print status

    opdracht 8 = if 1 opd4plane #%flag! then \ at nav goto #
```

```
    opdracht 9 = if 2 opd4plane #%flag! then \ at nav goto % - only works for right nav
    ;

: entrymodule ( -- )                        \  catches keystrokes
    key? if                                 \ true to jump to next 15s
        key >caps
        input_handler
        is= 3 = if
            checkopdracht if doopdracht then
            win4cr
            0 to is=
        then
    \   is= 4 = if                          \ catch next 15s
            \ true
            \ cr ." jump"                   \ << debug - functions
    \   then

    then
    \ false
    ;
: gameloop
    gameinit
    startmodule if exit then
    begin
        userinput
        0 to prop_phase
        uartclear cr
        #steps 0 do
            i ." step: " .
            ." score: " score 25 / .

            ." dp1: " depth .
            i to presentstep

            prepplanes                      \ activate plane & update destinations
            .upcommingplanes
            .activeplanes
            checkplanes                     \ << a -1 every big loop

            30 0 do                         \ 30 microsteps of ~500ms
                i to microstep
                .planesfield
                resettimer begin
                    entrymodule             \ entrymodule false or true (=jump to next
         15s )
                    gettimer 50 >=
                until                       \ leave on true
                switchscreens
                is= 4 = if 0 to is= leave then
            loop

            updateplanes                    \ destination to source
            updatestateengine
            switchpropphase
            [points] +to score

            cr
        loop
    endmodule                               \ f/t
    until                                   \ exit on
    backtonormal
    ;

\ end game *********************************
\ *****************************************
```

```
: .pl \ show planes - first do GAMEINIT!! - #debug
    #plane 1 do
        cr 8 spaces i .planel
        2 spaces
        i stat@ .
        i pstart@ .
        i xnew@ .
        i ynew@ .
        i height@ .
        i newhght@ .
    loop ;

: btn backtonormal ;

<|||<

unused -      . ." bytes"
word# swap -  . ." definitions"
```

– end of text –