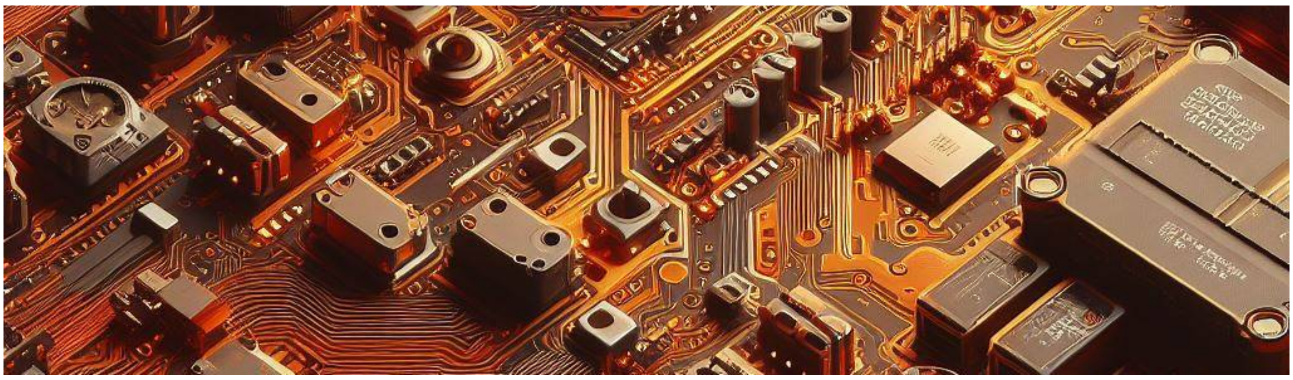
The great book for eFORTH Linux

version 1.3 - 7 décembre 2023



Author

- Marc PETREMANN

Contents

Introduction

Since 2019, I manage several websites dedicated to FORTH language development for ARDUINO and ESP32 boards, as well as the eForth web version:

- ARDUINO : https://arduino-forth.com/
- ESP32 : https://esp32.arduino-forth.com/
- eForth web : https://eforth.arduino-forth.com/

These sites are available in two languages, French and English. Every year I pay for hosting the main site **arduino-forth.com**.

It will happen sooner or later – and as late as possible – that I will no longer be able to ensure the sustainability of these sites. The consequence will be that the information disseminated by these sites disappears.

This book is the compilation of content from my websites. It is distributed freely from a Github repository. This method of distribution will allow greater sustainability than websites.

Incidentally, if some readers of these pages wish to contribute, they are welcome:

- to suggest chapters ;
- to report errors or suggest changes;
- to help with the translation...

Translation help

Google Translate allows you to translate texts easily, but with errors. So I'm asking for help to correct the translations.

In practice, I provide the chapters already translated in the LibreOffice format. If you want to help with these translations, your role will simply be to correct and return these translations.

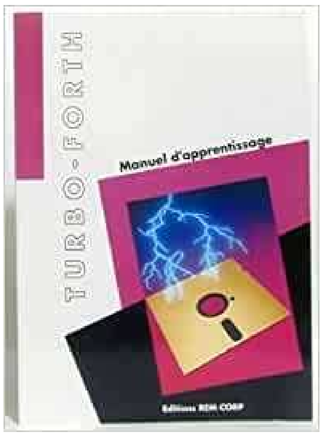
Correcting a chapter takes little time, from one to a few hours.

To contact me : petremann@arduino-forth.com

Why program in FORTH language on eForth Linux?

Preamble

I have been programming in FORTH since 1983. I stopped programming in FORTH in 1996. But I have never stopped monitoring the evolution of this language. I resumed programming in 2019 on ARDUINO with FlashForth then ESP32forth.



I am co-author of several books concerning the FORTH langage :

- Introduction au ZX-FORTH (ed Eyrolles - 1984 - ASIN:B0014IGOXO)
- Tours de FORTH (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH pour CP/M et MSDOS (ed Loisitech - 1986)
- TURBO-Forth, manuel d'apprentissage (ed Rem CORP - 1990)
- TURBO-Forth, guide de référence (ed Rem CORP - 1991)

Programming in the FORTH language was always a hobby until 1992 when the manager of a company working as a subcontractor for the automobile industry contacted me. They had a concern for software development in C language. They needed to order an industrial automaton.

The two software designers of this company programmed in C language: TURBO-C from Borland to be precise. And their code couldn't be compact and fast enough to fit into the 64 kilobytes of RAM memory. It was 1992 and flash memory type expansions did not exist. In these 64 KB of RAM, we had to fit MS-DOS 3.0 and the application!

For a month, C language developers had been twisting the problem in all directions, even reverse engineering with SOURCER (a disassembler) to eliminate non-essential parts of executable code.

I analyzed the problem that was presented to me. Starting from scratch, I created, alone, in a week, a perfectly operational prototype that met the specifications. For three years, from 1992 to 1995, I created numerous versions of this application which was used on the assembly lines of several automobile manufacturers.

Boundaries between language and application

All programming languages are shared like this :

- an interpreter and executable source code: BASIC, PHP, MySQL, JavaScript, etc... The application is contained in one or more files which will be interpreted whenever necessary. The system must permanently host the interpreter running the source code;
- a compiler and/or assembler: C, Java, etc. Some compilers generate native code, that is to say executable specifically on a system. Others, like Java, compile executable code on a virtual Java machine.

The FORTH language is an exception. It integrates :

- an interpreter capable of executing any word in the FORTH language
- a compiler capable of extending the dictionary of FORTH words

What is a FORTH word?

A FORTH word designates any dictionary expression composed of ASCII characters and usable in interpretation and/or compilation: words allows you to list all the words in the FORTH dictionary.

Certain FORTH words can only be used in compilation: `if else then` for example.

With the FORTH language, the essential principle is that we do not create an application. In FORTH, we extend the dictionary! Each new word you define will be as much a part of the FORTH dictionary as all the words pre-defined when FORTH starts. Example:

```
: typeToLoRa ( -- )
    0 echo !     \ disable display echo from terminal
    ['] serial2-type is type
  ;
: typeToTerm ( -- )
    ['] default-type is type
    -1 echo !    \ enable display echo from terminal
  ;
```

We create two new words: `typeToLoRa` and `typeToTerm` which will complete the dictionary of pre-defined words.

A word is a function?

Yes and no. In fact, a word can be a constant, a variable, a function... Here, in our example, the following sequence :

```
: typeToLoRa ...code... ;
```

would have its equivalent in C langage :

```
void typeToLoRa() { ...code... }
```

In FORTH language, there is no limit between language and application.

In FORTH, as in C language, you can use any word already defined in the definition of a new word.

Yes, but then why FORTH rather than C?

I was expecting this question.

In C language, a function can only be accessed through the main function `main()`. If this function integrates several additional functions, it becomes difficult to find a parameter error in the event of a malfunction of the program.

On the contrary, with FORTH it is possible to execute - via the interpreter - any word pre-defined or defined by you, without having to go through the main word of the program.

The FORTH interpreter is immediately accessible on eForth Linux.

The compilation of programs written in FORTH language is carried out in the ESP32 card and not on the PC. There is no edit→Link→compile→Run cycle. Example:

```
: >gray ( n -- n' )
    dup 2/ xor      \ n' = n xor ( 1 time right shift logic )
  ;
```

This definition is executable immediatly. The FORTH interpreter/compiler will parse the stream and compile the new word `>gray`.

In the definition of `>gray`, we see the sequence `dup 2/ xor`. To test this sequence, simply type it in the terminal. To execute `>gray`, simply type this word in the terminal, preceded by the number to transform.

FORTH language compared to C language

This is my least favorite part. I don't like to compare the FORTH language to the C language. But as almost all developers use the C language, I'm going to try the exercise.

Here is a test with `if()` in C language:

```
if(j > 13){                 // If all bits are received
    rc5_ok = 1;             // Decoding process is OK
    detachInterrupt(0);     // Disable external interrupt (INT0)
    return;
}
```

Test with if in FORTH language (code snippet) :

```
var-j @ 13 >        \ If all bits are received
    if
        1 rc5_ok !  \ Decoding process is OK
        di          \ Disable external interrupt (INT0)
        exit
    then
```

Here is the initialization of registers in C langage :

```
void setup() {
  // Timer1 module configuration
  TCCR1A = 0;
  TCCR1B = 0;          // Disable Timer1 module
  TCNT1  = 0;          // Set Timer1 preload value to 0 (reset)
  TIMSK1 = 1;          // enable Timer1 overflow interrupt
}
```

The same definition in FORTH langage :

```
: setup
  \ Timer1 module configuration
  0 TCCR1A !
  0 TCCR1B !     \ Disable Timer1 module
  0 TCNT1  !     \ Set Timer1 preload value to 0 (reset)
  1 TIMSK1 !     \ enable Timer1 overflow interrupt
;
```

What FORTH allows you to do compared to the C language

We understand that FORTH immediately gives access to all the words in the dictionary, but not only that. Via the interpreter, we also access the entire memory of eForth Linux :

```
hex here 100 dump
```

You should find this on the terminal screen :

```
3FFEE964                         DF DF 29 27 6F 59 2B 42 FA CF 9B 84
3FFEE970            39 4E 35 F7 78 FB D2 2C A0 AD 5A AF 7C 14 E3 52
3FFEE980            77 0C 67 CE 53 DE E9 9F 9A 49 AB F7 BC 64 AE E6
3FFEE990            3A DF 1C BB FE B7 C2 73 18 A6 A5 3F A4 68 B5 69
3FFEE9A0            F9 54 68 D9 4D 7C 96 4D 66 9A 02 BF 33 46 46 45
3FFEE9B0            45 39 33 33 2F 0D 08 18 BF 95 AF 87 AC D0 C7 5D
3FFEE9C0            F2 99 B6 43 DF 19 C9 74 10 BD 8C AE 5A 7F 13 F1
3FFEE9D0            9E 00 3D 6F 7F 74 2A 2B 52 2D F4 01 2D 7D B5 1C
3FFEE9E0            4A 88 88 B5 2D BE B1 38 57 79 B2 66 11 2D A1 76
3FFEE9F0            F6 68 1F 71 37 9E C1 82 43 A6 A4 9A 57 5D AC 9A
3FFEEA00            4C AD 03 F1 F8 AF 2E 1A B4 67 9C 71 25 98 E1 A0
3FFEEA10            E6 29 EE 2D EF 6F C7 06 10 E0 33 4A E1 57 58 60
3FFEEA20            08 74 C6 70 BD 70 FE 01 5D 9D 00 9E F7 B7 E0 CA
3FFEEA30            72 6E 49 16 0E 7C 3F 23 11 8D 66 55 EC F6 18 01
3FFEEA40            20 E7 48 63 D1 FB 56 77 3E 9A 53 7D B6 A7 A5 AB
3FFEEA50            EA 65 F8 21 3D BA 54 10 06 16 E6 9E 23 CA 87 25
3FFEEA60            E7 D7 C4 45
```

This corresponds to the contents of memory.

And the C language couldn't do that?

Yes, but not as simple and interactive as in FORTH language.

But why a stack rather than variables?

The stack is a mechanism implemented on almost all microcontrollers and microprocessors. Even the C language leverages a stack, but you don't have access to it.

Only the FORTH language gives full access to the data stack. For example, to make an addition, we stack two values, we execute the addition, we display the result: `2 5 + .` displays 7.

It's a little destabilizing, but when you understand the mechanism of the data stack, you greatly appreciate its formidable efficiency.

The data stack allows data to be passed between FORTH words much more quickly than by processing variables as in C language or any other language using variables.

Are you convinced?

Personally, I doubt that this single chapter will irremediably convert you to programming in the FORTH language. When trying to master Linux, you have two options :

- program in C language and use the numerous libraries available. But you will remain locked into the capabilities of these libraries. Adapting codes to C language requires real knowledge of programming in C language and mastering the architecture of LINUX. Developing complex programs will always be a problem.
- try the FORTH adventure and explore a new and exciting world. Of course, it won't be easy. You will need to understand the architecture of LINUX, libarries, network... In return, you will have access to programming perfectly suited to your projects.

Are there any professional applications written in FORTH?

Oh yes! Starting with the HUBBLE space telescope, certain components of which were written in FORTH language.

The German TGV ICE (Intercity Express) uses RTX2000 processors to control motors via power semiconductors. The machine language of the RTX2000 processor is the FORTH language.

This same RTX2000 processor was used for the Philae probe which attempted to land on a comet.

The choice of the FORTH language for professional applications turns out to be interesting if we consider each word as a black box. Each word must be simple, therefore have a fairly short definition and depend on few parameters.

During the debugging phase, it becomes easy to test all the possible values processed by this word. Once made perfectly reliable, this word becomes a black box, that is to say a function in which we have absolute confidence in its proper functioning. From word to word, it is easier to make a complex program reliable in FORTH than in any other programming language.

But if we lack rigor, if we build gas plants, it is also very easy to get an application that works poorly, or even to completely crash FORTH!

Good programming.

Install eForth on Linux

eForth Linux is a very powerful version for Linux system. eForth Linux works on all recent versions of Linux, including in a Linux virtual environment.

Prerequisites

You must have a working Linux system:

- installed on a computer using Linux as the only operating system;
- installed in a virtual environment.

If you only have a computer running Windows 10 or 11, you can install Linux in the WSL[1] subsystem.

Windows Subsystem for Linux allows developers to run a GNU/Linux environment (including most utilities, applications, and command-line tools) directly on Windows, without modification and without overloading a machine, traditional virtual or a dual-boot configuration.

The advantage of installing a Linux distribution in **WSL** allows you to have a Linux version available in command mode in a few seconds. Here, **Ubuntu** is accessible from the Windows file system and launches with a single click :

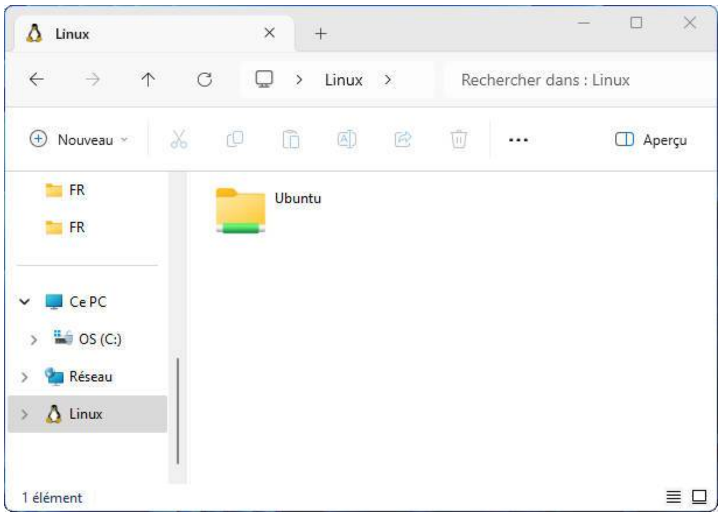


Figure 1: Ubuntu accessible in one click from WSL under Windows

All instructions for installing **WSL2** and then the Linux distribution of your choice are available here:

https://learn.microsoft.com/en-us/windows/wsl/install

By default, WSL2 offers to install the **Ubuntu Linux distribution**.

1 WSL = Windows Subsystem Linux

Install eForth Linux on Linux

If you launch Ubuntu (or any other version of Linux), you will find yourself in your user directory by default. We start by accessing to **usr/bin** directory :

```
cd /usr/bin
```

We will now download the version of the ueForth Linux binary file :

- either from the home page of Brad NELSON's ESP32forth site: https://esp32forth.appspot.com/ESP32forth.html
- either from the eforth Google storage repository: https://eforth.storage.googleapis.com/releases/archive.html

In the list of proposed files, copy the web link mentioning Linux:

```
https://eforth.storage.googleapis.com/releases/ueforth-7.0.7.15.linux
```

On Linux, type the `wget command` :

```
sudo wget https://eforth.storage.googleapis.com/releases/ueforth-7.0.7.15.linux
```

The download will automatically drop the file into the previously selected folder. If you took the link above, you end up with a file named **ueforth-7.0.7.15.linux** in this folder.

We rename this file with the `mv` command :

```
mv ueforth-7.0.7.15.linux ueforth
```

We check that everything went well with a simple `dir ue*` command.

We still have one last manipulation to perform, making this file executable by the Linux system :

```
sudo chmod 755 ueforth
```

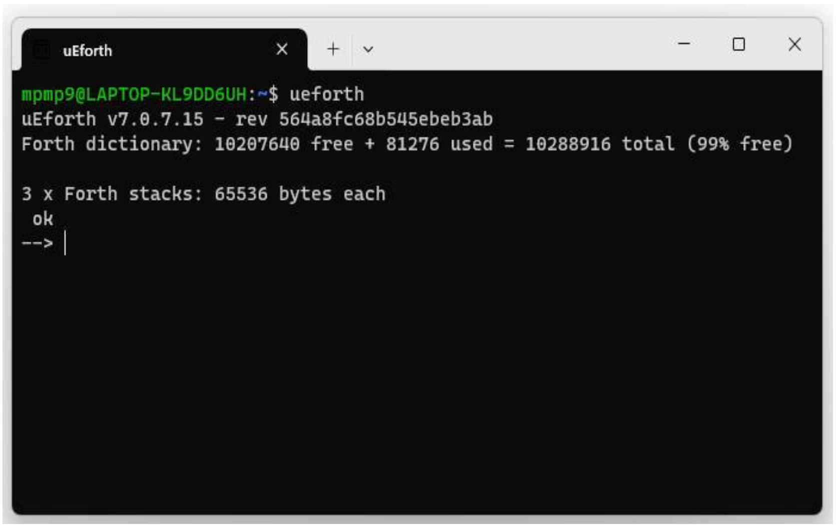
And it's done ! eForth Linux can now be used from any Linux directory.

Launch eForth Linux

To launch **eForth when Linux** boots :

```
cd ueforth
./ueforth.bin
```

eForth Linux starts immediately:



```
mpmp9@LAPTOP-KL9DD6UH:~$ ueforth
uEforth v7.0.7.15 - rev 564a8fc68b545ebeb3ab
Forth dictionary: 10207640 free + 81276 used = 10288916 total (99% free)

3 x Forth stacks: 65536 bytes each
 ok
-->
```

Figure 2: eForth Linux is active

You can now test eForth and program your first applications in FORTH language.

PLEASE NOTE : this eForth version handles integers in 64-bit format. It's easy to check:

```
cell. \ display: 8
```

Or a dimension of 8 bytes for integers. This warning is essential if you are using FORTH code written for 16 or 32 bit versions.

Good programming.

A real 64-bit FORTH with eForth Linux

Eforth Linux is a real 64-bit FORTH. What does it mean?

The FORTH language favors the manipulation of integer values. These values can be literal values, memory addresses, register contents, etc.

Values on the data stack

When Eforth Linux starts, the FORTH interpreter is available. If you enter any number, it will be dropped onto the stack as a 64-bit integer :

```
35
```

If we stack another value, it will also be stacked. The previous value will be pushed down one position :

```
45
```

To add these two values, we use a word, here `+`:

```
+
```

Our two 64-bit integer values are added together and the result is dropped onto the stack. To display this result, we will use the word `.`:

```
. \ display 80
```

In FORTH language, we can concentrate all these operations in a single line :

```
35 45 + .  \ display 80
```

Unlike the C language, we do not define an **int8** or **int16** or **int32** or **int64** type.

With Eforth Linux, an ASCII character will be designated by a 64-bit integer, but whose value will be bounded [32..255]. Example :

```
67 emit  \ display C
```

Values in memory

eForth Linux allows you to define constants and variables. Their content will always be in 64-bit format. But there are situations where that doesn't necessarily suit us. Let's take a simple example, defining a Morse code alphabet. We only need a few bytes :

- one to define number of marks in Morse code character
- one or more bytes for Morse code marks

```
create mA ( -- addr )
    2 c,
    char . c,   char - c,
create mB ( -- addr )
    4 c,
    char - c,   char . c,   char . c,   char . c,

create mC ( -- addr )
    4 c,
    char - c,   char . c,   char - c,   char . c,
```

Here we define only 3 words, `mA`, `mB` and `mC`. In each word, several bytes are stored. The question is: how will we retrieve the information in these words ?

The execution of one of these words deposits a 64-bit value, a value which corresponds to the memory address where we stored our Morse code information. It is the word `c@` that we will use to extract the Morse code from each letter :

```
mA c@ .   \ display 2
mB c@ .   \ display 4
```

The first byte placed on the stack will be used to manage a loop to display the code of a character in Morse code :

```
: .morse ( addr -- )
    dup 1+ swap c@ 0 do
        dup i + c@ emit
    loop
    drop
  ;
mA .morse   \ display .-
mB .morse   \ display -...
mC .morse   \ display -.-.
```

There are plenty of certainly more elegant examples. Here we show a way to manipulate 8-bit values, our bytes, while operating these bytes on a 64-bit stack.

Word processing depending on data size or type

In all other languages, we have a generic word, like `echo` (in PHP) which displays any type of data. Whether integer, real, string, we always use the same word. Example in PHP language:

```
$bread = "Baked bread";
$price = 2.30;
echo $bread . " : " . $price;
// display   Baked bread: 2.30
```

For all programmers, this way of doing things is THE STANDARD! So how would FORTH do this example in PHP?

```
: bread s" Baked bread" ;
: price s" 2.30" ;
bread type   s" : " type    price type
```

```
\ display   Baked bread: 2.30
```

Here, the word `type` tells us that we have just processed a character string.

Where PHP (or any other language) has a generic function and a parser, FORTH compensates with a single data type, but adapted processing methods which inform us about the nature of the data processed.

Here is an absolutely trivial case for FORTH, displaying a number of seconds in HH:MM:SS format:

```
: :##
    #   6 base !
    #   decimal
    [char] : hold
  ;
: .hms  ( n -- )
    <#  :##  :##  #  #  #>   type
  ;
4225 .hms   \ display: 01:10:25
```

I love this example because, to date, **NO OTHER PROGRAMMING LANGUAGE** is capable of achieving this HH:MM:SS conversion so elegantly and concisely.

You have understood, the secret of FORTH is in its vocabulary.

Conclusion

FORTH has no data typing. All data passes through a data stack. Each position in the stack is ALWAYS a 64-bit integer!

That's all there is to know.

Purists of hyper-structured and verbose languages, such as C or Java, will certainly cry heresy. And here, I will allow myself to answer them : why do you need to type your data ?

Because it is in this simplicity that the power of FORTH lies : a single stack of data with an untyped format and very simple operations.

And I'm going to show you what many other programming languages can't do, define new definition words :

```
: morse: ( comp: c -- | exec -- )
    create
        c,
    does>
        dup 1+ swap c@ 0 do
            dup i + c@ emit
        loop
        drop space
```

```
  ;
2 morse: mA     char . c,   char - c,
4 morse: mB     char - c,   char . c,   char . c,   char . c,
4 morse: mC     char - c,   char . c,   char - c,   char . c,
mA mB mC    \ display   .- -... -.-.
```

Here, the word `morse:` has become a definition word, in the same way as constant or variable...

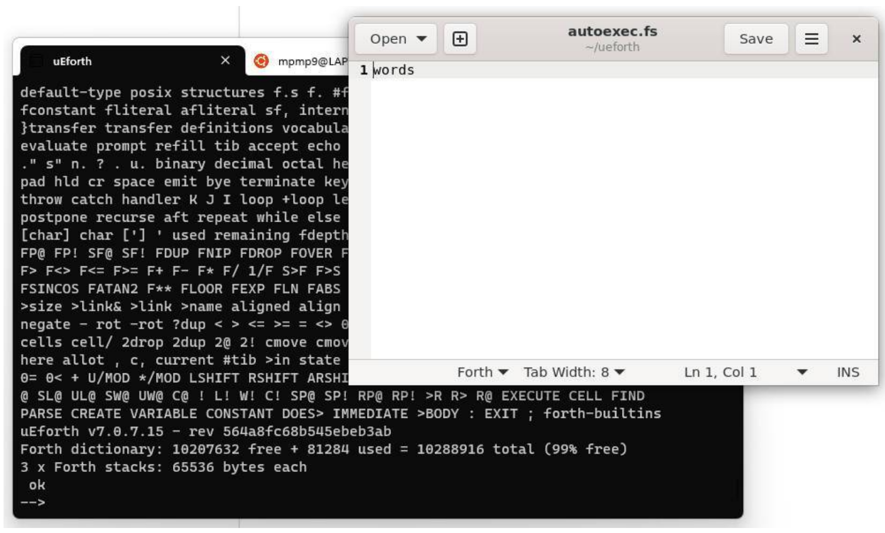
Because FORTH is more than a programming language. It is a meta-language, that is to say a language to build your own programming language....

Editing and managing source files for eForth Linux

As with the vast majority of programming languages, source files written in FORTH language are in simple text format. The extension of files in FORTH language is free:

- **txt** generic extension for all text files;
- **forth** used by some FORTH programmers;
- **fth** compressed form for FORTH;
- **4th** other compressed form for FORTH;
- **fs** our favorite extension…

Text file editors



```
words
```

```
default-type posix structures f.s f. #f
fconstant fliteral afliteral sf, intern
}transfer transfer definitions vocabula
evaluate prompt refill tib accept echo
." s" n. ? . u. binary decimal octal he
pad hld cr space emit bye terminate key
throw catch handler K J I loop +loop le
postpone recurse aft repeat while else
[char] char ['] ' used remaining fdepth
FP@ FP! SF@ SF! FDUP FNIP FDROP FOVER F
F> F<> F<= F>= F+ F- F* F/ 1/F S>F F>S
FSINCOS FATAN2 F** FLOOR FEXP FLN FABS
>size >link& >link >name aligned align
negate - rot -rot ?dup < > <= >= = <> 0
cells cell/ 2drop 2dup 2@ 2! cmove cmov
here allot , c, current #tib >in state
0= 0< + U/MOD */MOD LSHIFT RSHIFT ARSHI
@ SL@ UL@ SW@ UW@ C@ ! L! W! C! SP@ SP! RP@ RP! >R R> R@ EXECUTE CELL FIND
PARSE CREATE VARIABLE CONSTANT DOES> IMMEDIATE >BODY : EXIT ; forth-builtins
uEforth v7.0.7.15 - rev 564a8fc68b545ebeb3ab
Forth dictionary: 10207632 free + 81284 used = 10288916 total (99% free)
3 x Forth stacks: 65536 bytes each
 ok
-->
```

Figure 3: editing autoexec.fs file with gedit on Linux

gedit file editor is the simplest:

If you use a custom file extension, such as **fs** , for your FORTH language source files, Linux will recognize these files as plain text.

Storage on GitHub

GitHub [2]website is, along with **SourceForge** [3], one of the best places to store source files.

On GitHub, you can share a

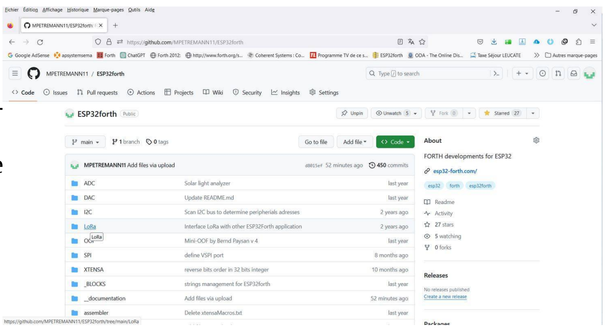


Figure 4: files storage on Github

2 https://github.com/
3 https://sourceforge.ne

working folder with other developers and manage complex projects. The Netbeans editor can connect to the project and allows you to pass or retrieve file changes.

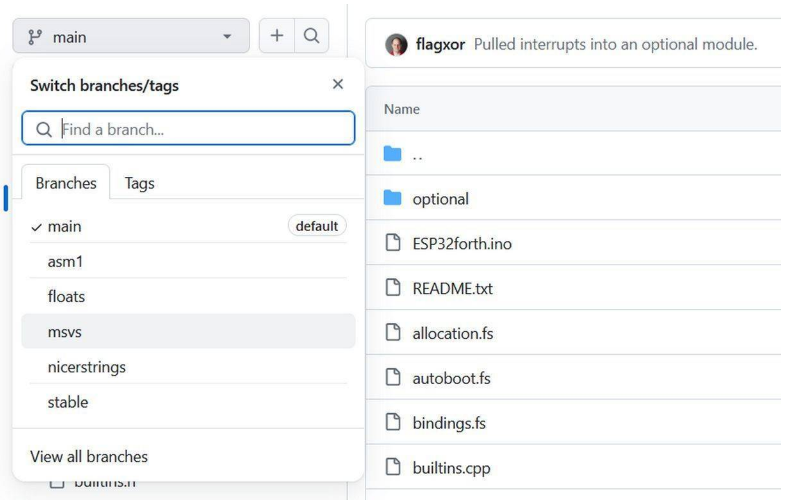


Figure 5: access to a branch in a project

On **GitHub** , you can manage project *forks* . You can also make certain parts of your projects confidential. Above are the branches in the flagxor/ueforth projects:

Edit files for eForth Linux from Windows

If you have installed a Linux version that runs in the WSL2 environment, it is perfectly possible to edit Linux source files from Windows:

- launch Ubuntu from Windows
- Once Ubuntu is active, move the mouse pointer out of the WSL window. You return to the Windows environment. Open Windows File Manager.
- in the left pane, click *Linux* ;

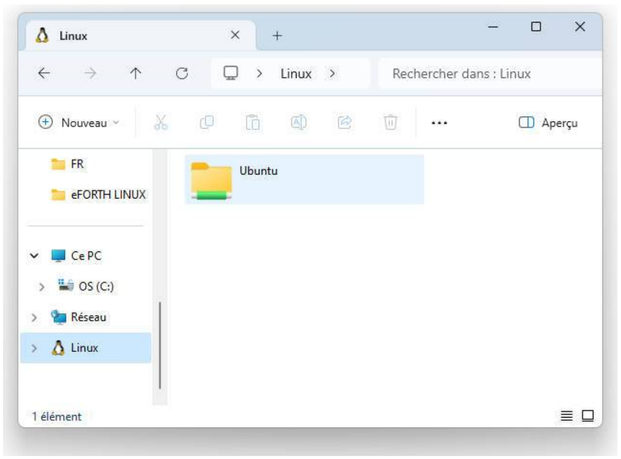
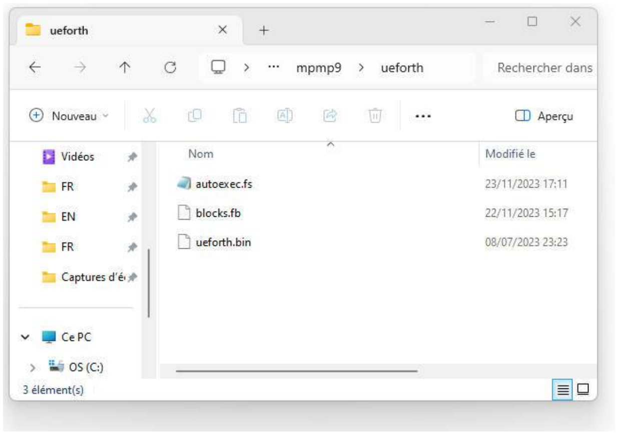


Figure 6: accessing Linux files from Windows

- in the main pane, click on the Linux version, here *Ubuntu* ;



Nom	Modifié le
autoexec.fs	23/11/2023 17:11
blocks.fb	22/11/2023 15:17
ueforth.bin	08/07/2023 23:23

Figure 7: Linux files visible from Windows

- navigate to the eForth folder: home folder → User → ueforth →
- select the file to edit. For the example, we will open **autoexec.fs** ;

If you use an IDE, like Netbeans, here is how to configure this IDE to integrate your eForth Linux development projects.

Creation and management of FORTH projects with Netbeans

As a prerequisite, you must install Netbeans. Link for download and installation: https://netbeans.apache.org/front/main/

Netbeans can be installed on Windows or Linux. For my part, having already installed Netbeans under Windows, I am not going to overload my machine by installing a Linux version. Consequently, the following explanations concern the management of an eForth Linux project via WSL2 from Windows.

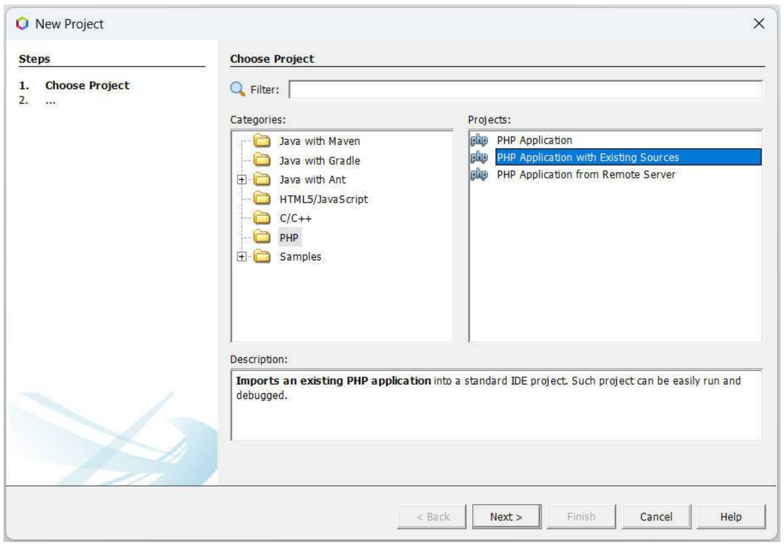
Create an eForth project with Netbeans

There, also a prerequisite:

- ueforth Linux is installed in Linux via WSL2 Windows.
- The source files are in a Windows folder:
  `Linux → Ubuntu → home → userName → ueforth`
  where ***`userName`*** is the username defined during the Linux installation
- all eForth Linux source files are saved in the **`ueforth directory`**

Launch Netbeans. To create a new Netbeans project:

- click *File* → select *New Project...*
- **New Project** window , select Categories: *PHP* and in Projects: *PHP Application with Existing Sources*



Imports an existing PHP application into a standard IDE project. Such project can be easily run and debugged.

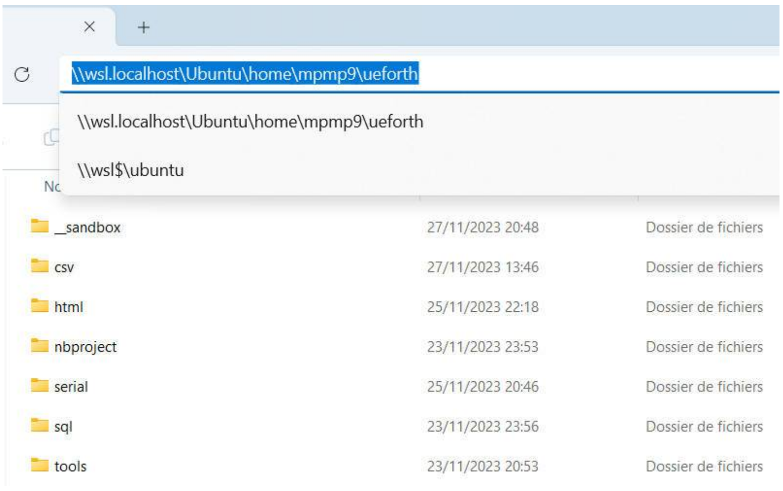
Figure 8: création projet PHP

- click *Next >*
- In the **Name and Location** → *Sources Folder field* , enter the path to the eForth Linux source files

ueforth folder from Windows, launch File Explorer. At the bottom right, click **Ubuntu** . Then click on the folders:

`home → userName → ueforth`

In the navigation bar, at the top, you must find the path to the ueforth folder. Place the mouse pointer in this banner. Copy the path:

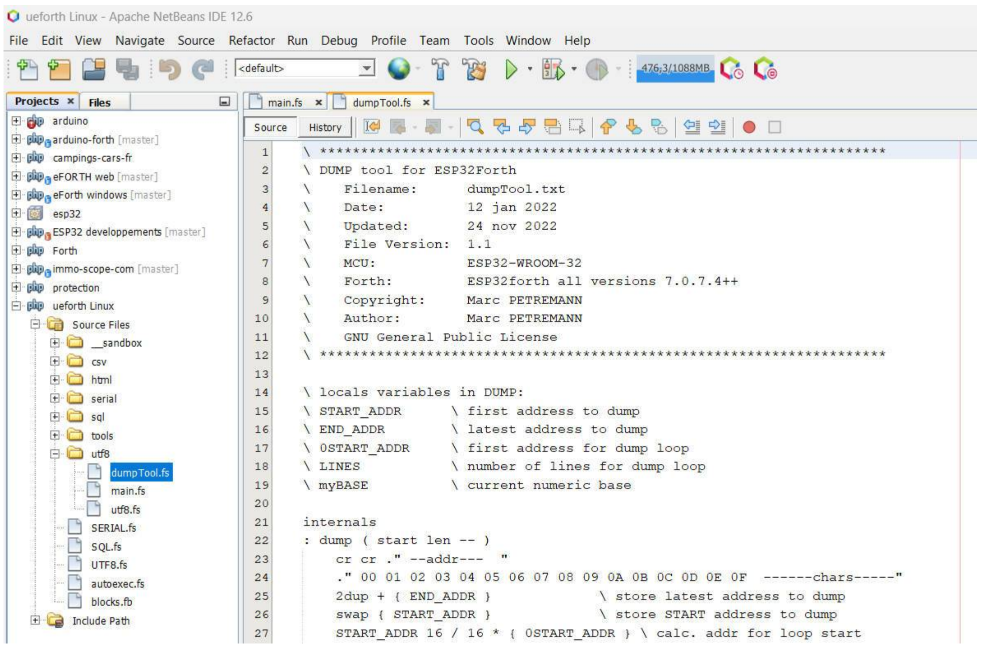


```
\\wsl.localhost\Ubuntu\home\mpmp9\ueforth
\\wsl.localhost\Ubuntu\home\mpmp9\ueforth
\\wsl$\ubuntu
```

_sandbox	27/11/2023 20:48	Dossier de fichiers
csv	27/11/2023 13:46	Dossier de fichiers
html	25/11/2023 22:18	Dossier de fichiers
nbproject	23/11/2023 23:53	Dossier de fichiers
serial	25/11/2023 20:46	Dossier de fichiers
sql	23/11/2023 23:56	Dossier de fichiers
tools	23/11/2023 20:53	Dossier de fichiers

Figure 9: copy path to ueforth on Linux

Paste this path into the Netbeans field described above. Finish creating the new project in Netbeans. You can now find all the files of your project in Netbeans:



```
\ **********************************************************************
\ DUMP tool for ESP32Forth
\    Filename:      dumpTool.txt
\    Date:          12 jan 2022
\    Updated:       24 nov 2022
\    File Version:  1.1
\    MCU:           ESP32-WROOM-32
\    Forth:         ESP32forth all versions 7.0.7.4++
\    Copyright:     Marc PETREMANN
\    Author:        Marc PETREMANN
\    GNU General Public License
\ **********************************************************************

\ locals variables in DUMP:
\ START_ADDR      \ first address to dump
\ END_ADDR        \ latest address to dump
\ 0START_ADDR     \ first address for dump loop
\ LINES           \ number of lines for dump loop
\ myBASE          \ current numeric base

internals
: dump ( start len -- )
    cr cr ." --addr---  "
    ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----"
    2dup + { END_ADDR }             \ store latest address to dump
    swap { START_ADDR }             \ store START address to dump
    START_ADDR 16 / 16 * { 0START_ADDR } \ calc. addr for loop start
```

Figure 10: the new project is operational

Now any editing, creating, modifying or deleting a file from Netbeans is immediately reflected in your **ueforth** project folder on Linux.

Some good practices

The first good practice is to name your working files and folders correctly. You are developing for eforth, so stay in the folder named **ueforth** .

For various tests, create a **sandbox subfolder in this folder** .

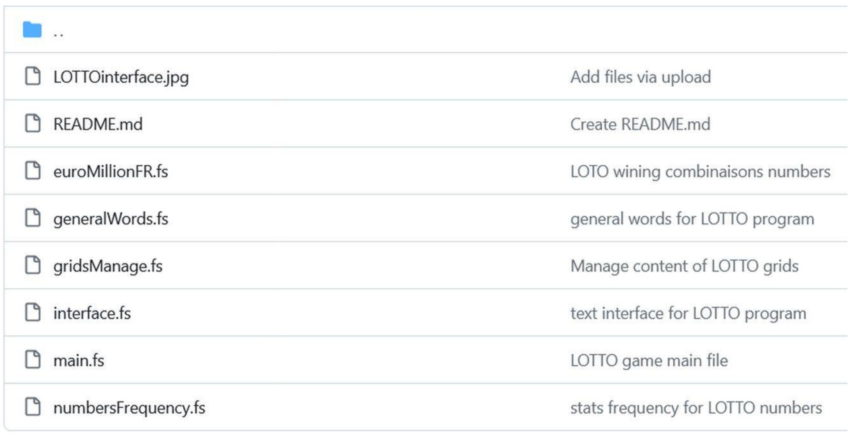
For well-constructed projects, create a folder per project. For example, you want to develop a game, create a **myGame** subfolder .

tools subfolder . If you are using a file from this **tools folder** in a project, copy and paste that file into that project's folder. This will prevent a modification of a file in **tools** from subsequently disrupting your project.

For FORTH tests without a specific purpose, put them in a **__sandbox folder** .

The second best practice is to distribute the source code of a project into several files:

- **config.fs** to store project settings;
- **documentation** directory to store files in the format of your choice, related to the project documentation;



..	
LOTTOinterface.jpg	Add files via upload
README.md	Create README.md
euroMillionFR.fs	LOTO wining combinaisons numbers
generalWords.fs	general words for LOTTO program
gridsManage.fs	Manage content of LOTTO grids
interface.fs	text interface for LOTTO program
main.fs	LOTTO game main file
numbersFrequency.fs	stats frequency for LOTTO numbers

Figure 11: exemple de nommage de fichiers source Forth

- **myApp.fs** for your project definitions. Choose a fairly explicit file name. For example, to manage your game, take the name **game-commands.fs** .

Executing the contents of a file by eForth Linux

From eForth Linux, executing the contents of a source file is done very simply by using the word `include` followed by the file name:

```
include autoexec.fs
```

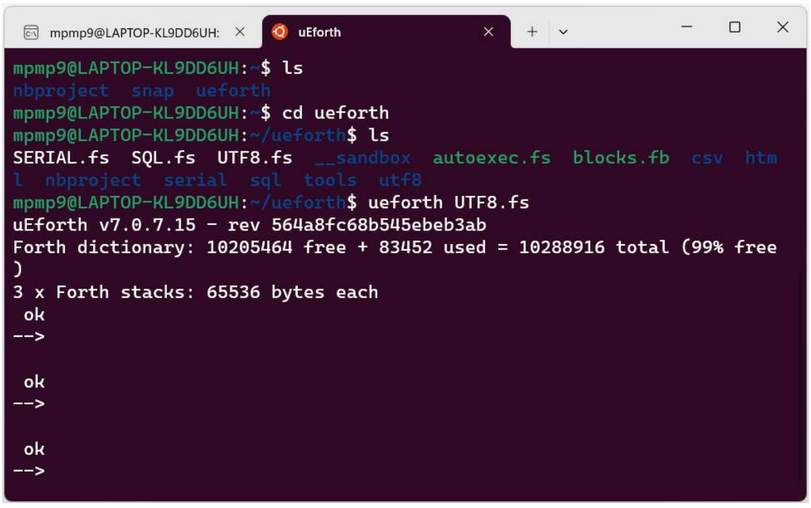
executes the contents of the **autoexec.fs file** .

If the file to read is in a subfolder, the file name will be preceded by the folder name. Example to launch **main.fs in the myGame** subfolder :

```
cd mygame
include main.fs
```

If you have correctly installed **ueforth** , its launch may be followed by the name of the source file to execute. On Linux:

```
cd ueforth
ueforth UTF8.fs
```



```
mpmp9@LAPTOP-KL9DD6UH:~$ ls
nbproject  snap  ueforth
mpmp9@LAPTOP-KL9DD6UH:~$ cd ueforth
mpmp9@LAPTOP-KL9DD6UH:~/ueforth$ ls
SERIAL.fs  SQL.fs  UTF8.fs  __sandbox  autoexec.fs  blocks.fb  csv  htm
l  nbproject  serial  sql  tools  utf8
mpmp9@LAPTOP-KL9DD6UH:~/ueforth$ ueforth UTF8.fs
uEforth v7.0.7.15 - rev 564a8fc68b545ebeb3ab
Forth dictionary: 10205464 free + 83452 used = 10288916 total (99% free
)
3 x Forth stacks: 65536 bytes each
 ok
-->

 ok
-->

 ok
-->
```

Figure 12: executing a file when launching ueforth

Linux logs all system commands, even after shutting down the PC and restarting it. It is therefore very easy to restart project processing with just a few presses of *the up arrow key* .

In summary, provided you have a Linux version accessible from **Windows WSL2** , you edit the source files with Netbeans from Windows. And you process project files from Linux.

If you are in a *full Linux environment* , the manipulations are not very different. To launch ueforth, you will need to open a command window in Linux.

The Linux file system

eForth Linux integrates the essential components for accessing Linux system files.

To compile the contents of a source file, here the **dumpTool.fs file in the tools** folder , edited by **gedit** , type:

```
include /tools/dumpTool.fs
```

The word `include` is an eForth dictionary word.

To see the list of Linux files , use the word `ls` :

```
ls    \ display :
.
..
autoexec.fs
blocks.fb
ueforth.bin
tools
 ok
```

Here we see the **tools folder** . Eforth Linux does not use syntax highlighting like Linux does. To see the contents of this **tools** subfolder , type:

```
ls tools\display:
ls tools
.
..
dumpTool.fs
```

There is no option to filter file names or pseudo directories.

Handling files

To completely delete a file, use the word `rm` followed by the name of the file to be deleted. Here we want to delete the myTest.fs file which was created and is no longer used:

```
rm myTest.fs\display:
  ok
```

To rename a file, use the word `mv` . For example, we want to rename a **myTest.txt file** :

```
mv myTest.txt myTest.fs
ls\display:
.
..
autoexec.fs
blocks.fb
```

```
myTest.fs
tools
```

To copy a file, use the word **cp** :

```
cp myTest.fs testColors.fs
ls\display:
.
..
autoexec.fs
blocks.fb
myTest.fs
testColors.fs
tools
```

To see the contents of a file, use the word **cat** :

```
cat autoexec.fs
\ displays contents of autoexec.fs
```

To save the contents of a string to a file, save the contents of the string with **dump-file** :

```
r| ." Insert my text into myTest" | s"myTest.fs" dump-file
```

We will not dwell on these manipulations which can also be carried out from Linux or a source text editor.

Organize and compile your files with eForth Linux

We will see how to manage files for an application being developed with eForth Linux.

It is agreed that all files used are in ASCII text format.

The following explanations are given as advice only. They come from a certain experience and aim to facilitate the development of large applications with eForth Linux.

All source files for your project are on your computer in the Linux environment. It is advisable to have a subfolder dedicated to this project. For example, you are working on a game named rubik, so you create a directory named **rubik** .

Regarding file name extensions, we recommend using the **fs** extension .

Editing files on a computer is carried out with any text file editor, **gedit** under Linux.

In these source files, do not use any characters not included in the ASCII code characters. Some extended codes can disrupt program compilation.

Organize your files

In the following, all our files will have the extension **fs** .

Let's start from our **rubik** directory on our computer.

The first file we will create in this directory will be the **main.fs file** . This file will contain the calls to load all the other files of our application under development.

Example of content of our **main.fs file** :

```
\ RUBIK game main file
s" config.fs" included
```

In the development phase, the contents of this **main.fs file** will be loaded from a **RUBIK.fs** file placed in the same folder as eForth and containing this:

```
cd rubik
s" main.fs" included
```

This causes the contents of our **main.fs file to be executed** . Loading of other files will be executed from this **main.fs file** . Here we load the **config.fs file** of which here is an extract:

```
0 value MAX_DEPTH
3 constant CUBE_SIZE
```

config.fs file we will put all the constant values and various global parameters used by the other files.

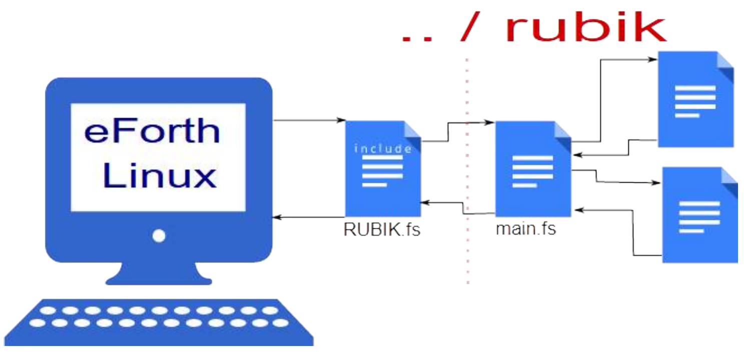


Figure 13: sequence of RUBIK project files

It is advisable to put all the files of the same project in the folder of this project, here **rubik** for our example.

Chaining of files

Each file can call a file with the word `included` . Here is an example of a file hierarchy included in this way:

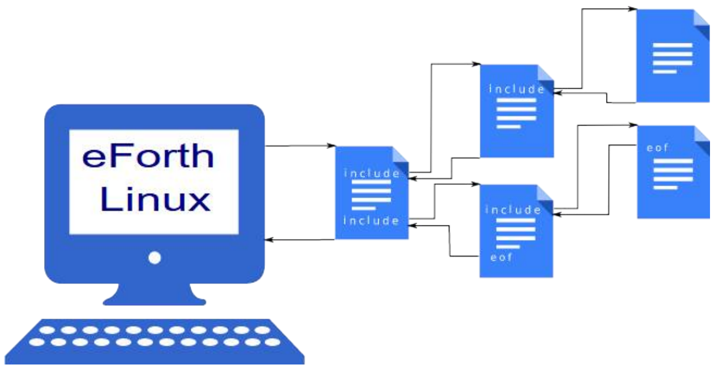


Figure 14: enchaînement de fichiers

Here, eForth calls a first file. Even if it is feasible, it is not recommended to create cascade sequences. Prefer a succession of loading files from **main.fs.** Example :

```
DEFINED? --tempusFugit [if] forget --tempusFugit  [then]
create --tempusFugit
s" strings.fs"          included
s" RTClock.fs"          included
s" clepsydra.fs"        included
s" config.fs"           included
s" dispTools.fs"        included
```

In this succession of files, we use the **strings.fs** file . This is a so-called *tool file* . It is the copy of a fairly general use file whose content extends the FORTH dictionary.

By working with a copy of the original file, you can make corrections or improvements without risking altering the operation of the code in the original file. If these modifications are consolidated, we can transfer them to the original file.

For each FORTH source code file, date the versions. This will allow you to find the chronology of code modifications.

Conclusion

Files saved in Linux system are available permanently. If you access a Linux version in a WSL2 management system from Windows, these files will also be accessible to the Windows file system.

Comments and debugging

There is no IDE[4] to manage and present code written in FORTH language in a structured way. At worst you use an ASCII text editor, at best a real IDE and text files:

- **edit** or **wordpad** on Windows
- **edit** under Linux
- **PsPad** under windows
- **Netbeans** under Windows or Linux...

Here is a code snippet that could be written by a beginner:

```
: inGrid? { n gridPos -- fl } 0 { fl } gridPos getGridAddr for aft
getNumber n = if -1 to fl then then next drop  fl ;
```

This code will be perfectly compiled by eForth Linux. But will it remain understandable in the future if it needs to be modified or reused in another application?

Write readable FORTH code

Let's start with the name of the word to be defined, here `inGrid?`. eForth Linux allows you to write very long word names. The size of the defined words has no influence on the performance of the final application. We therefore have a certain freedom to write these words :

- like object programming in JavaScript: `rid.test.number`
- the Camel wayCoding `gridTestNumber`
- for programmers wanting very understandable code `is-number-in-the-grid`
- programmer who likes concise code: `gtn?`

There is no rule. The main thing is that you can easily reread your FORTH code. However, computer programmers in FORTH language have certain habits:

- constants in uppercase characters `LOTTO_NUMBERS_IN_GRID`
- word defining other words `lottoNumber:` , i.e. word followed by a colon;
- address transformation word `>date` , here the address parameter is incremented by a certain value to point to the appropriate data;
- memory storage word `date@` or `date!`
- Data display word `.date`

4 Integrated Development Environment = Integrated Development Environment

And what about naming FORTH words in a language other than English? Here again, only one rule: **total freedom** ! Be careful though, eForth Linux does not accept names written in alphabets other than the Latin alphabet. However, you can use these alphabets for comments:

```
: .date     \ Плакат сегодняшней даты
    ….coded… ;
```

Or

```
: .date     \海報今天的日期
    ….coded… ;
```

Source code indentation

Whether the code is two lines, ten lines or more has no effect on the performance of the code once compiled. So, you might as well indent your code in a structured way:

- one line per word of control structure `if else then` , `begin while repeat…` For the word if, we can precede it with the logical test that it will process;
- a line by execution of a predefined word, preceded if necessary by the parameters of this word.

Example :

```
: inGrid? { n gridPos -- fl }
    0 { fl }
    gridPos getGridAddr
    for
        aft
            getNumber n =
            if
                -1 to fl
            then
        then
    next
    drop
    fl
  ;
```

If the code processed in a control structure is sparse, the FORTH code can be compacted:

```
: inGrid? { n gridPos -- fl }
    0 { fl }   gridPos getGridAddr
    for aft
            getNumber n =
            if -1 to fl  then
        then
    next
    drop  fl
  ;
```

This is often the case with `case of endof endcase` structures ;

```
: socketError ( -- )
    errno dup
    case
        2 of      ." No such file "                 endof
        5 of      ." I/O error "                    endof
        9 of      ." Bad file number "              endof
       22 of      ." Invalid argument "             endof
    endcase
    . quit
  ;
```

Comments

Like any programming language, the FORTH language allows the addition of comments in the source code. Adding comments has no impact on the performance of the application after compiling the source code.

In FORTH language, we have two words to delimit comments:

- the word `(` must be followed by at least one space character. This comment is completed by the character `)` ;
- the word `\` must be followed by at least one space character. This word is followed by a comment of any size between this word and the end of the line.

The word `(` is widely used for stack comments. Examples:

```
dup   ( n -- nn)
swap  ( n1 n2 -- n2 n1)
drop  ( n --)
emit  ( c -- )
```

Stack comments

As we have just seen, they are marked by `(` and `)` . Their content has no effect on the FORTH code during compilation or execution. So we can put anything between `(` and `)` . As for the stack comments, we will remain very concise. The `--` sign symbolizes the action of a FORTH word. The indications before `--` correspond to the data placed on the data stack before the execution of the word. The indications after `--` correspond to the data left on the data stack after execution of the word. Examples :

- `words ( -- )` means that this word does not process any data on the data stack;
- `emit ( c -- )` means that this word processes data as input and leaves nothing on the data stack ;
- `bl ( -- 32 )` means that this word does not process any input data and leaves the decimal value 32 on the data stack;

There is no limitation on the amount of data processed before or after execution of the word. As a reminder, the indications between `(` and `)` are only there for information.

Meaning of stack parameters in comments

To begin with, a small but very important clarification is necessary. This is the size of the data on stack. With eForth Linux, the stack data takes up 8 bytes. So these are integers in 64-bit format. So what do we put on the data stack? With eForth Linux, it will **ALWAYS be 64 BIT DATA** ! An example with the `c word!` :

```
create myDelemiter
    0 c,
64 myDelimiter c!   ( c addr -- )
```

Here, the parameter `c` indicates that we stack an integer value in 64-bit format, but whose value will always be included in the interval [0..255].

The standard parameter is always `n` . If there are several integers, we will number them: `n1 n2 n3` , etc.

We could therefore have written the previous example like this :

```
create myDelemiter
    0 c,
64 myDelimiter c!   ( n1 n2 -- )
```

But it is much less explicit than the previous version. Here are some symbols that you will see throughout the source codes:

- **addr** indicates a literal memory address or delivered by a variable;
- **c** indicates an 8-bit value in the interval [0..255]
- **d** indicates a double precision value.
  Not used with eForth Linux which is already in 64-bit format;
- **fl** indicates a Boolean value, 0 or non-zero;
- **n** indicates an integer. 64-bit signed integer for eForth Linux;
- **str** indicates a character string. Equivalent to `addr len --`
- **u** indicates an unsigned integer

Nothing prevents us from being a little more explicit:

```
: SQUARE ( n -- n-exp2 )
    dup *
  ;
```

Word Definition Word Comments

Definition words use `create` and `does>` . For these words, it is advisable to write stack comments like this:

```
\ define a command or data stream for SSD1306
: streamCreate: ( comp: <name> | exec: -- addr len )
    create
```

```
        here     \ leave current dictionnary pointer on stack
        0 c,     \ initial lenght data is 0
    does>
        dup 1+ swap c@
        \ send a data array to SSD1306 connected via I2C bus
        sendDatasToSSD1306
  ;
```

Here, the comment is split into two parts by the character `|` :

- on the left, the action part when the definition word is executed, prefixed by `comp:`
- on the right the action part of the word that will be defined, prefixed with `exec:`

At the risk of insisting, this is not a standard. These are only recommendations.

Textual comments

They are indicated by the word `\` followed by at least one space character and explanatory text:

```
\ store at <WORD> addr length of datas compiled beetween
\ <WORD> and here
: ;endStream ( addr-var len ---)
    dup 1+ here
    swap -      \ calculate cdata length
    \ store c in first byte of word defined by streamCreate:
    swap c!
  ;
```

These comments can be written in any alphabet supported by your source code editor:

```
\ 儲存在 <WORD> addr 之間編譯的資料長度
\ <WORD> 和這裡
: ;endStream ( addr-var len ---)
    dup 1+ here
    swap -      \ 計算 cdata 長度
    \ 將 c 儲存在由 StreamCreate 定義的字的第一個位元組中:
    swap c!
  ;
```

Comment at the beginning of the source code

With intensive programming practice, you quickly find yourself with hundreds or even thousands of source files. To avoid file choice errors, it is strongly recommended to mark the start of each source file with a comment:

```
\ *************************************
\ Manage commands for OLED SSD1306 128x32 display
\    Filename:      SSD10306commands.fs
\    Date:          21 may 2023
\    Updated:       21 may 2023
\    File Version:  1.0
\    MCU:           ESP32-WROOM-32
\    Forth:         ESP32forth all versions 7.x++
\    Copyright:      Marc PETREMANN
\    Author:         Marc PETREMANN
\    GNU General Public License
\ ***************************************
```

All this information is at your discretion. They can become very useful when you come back to the contents of a file months or years later.

To conclude, do not hesitate to comment and indent your source files in FORTH language.

Diagnostic and tuning tools

The first tool concerns the compilation or interpretation alert:

```
3 5 25 --> : TEST ( ---)
 ok
3 5 25 -->     [ HEX ] ASCII A DDUP    \ DDUP don't exist
```

Here, the word **DDUP** does not exist. Any compilation after this error will fail.

The decompiler

In a conventional compiler, the source code is transformed into executable code containing the reference addresses to a library equipping the compiler. To have executable code, you must link the object code. At no time can the programmer have access to the executable code contained in his library with the resources of the compiler alone.

With eForth Linux, the developer can decompile their definitions. To decompile a word, simply type **see** followed by the word to decompile:

```
: C>F ( øC --- øF) \ Conversion Celsius in Fahrenheit
    9 5 */ 32 +
  ;
see c>f
\ display:
: C>F
    9 5 */ 32 +
;
```

Many words in eForth's Linux FORTH dictionary can be decompiled.

Decompiling your words allows you to detect possible compilation errors.

Memory dump

Sometimes it is desirable to be able to see the values that are in memory. The word **dump** accepts two parameters: the starting address in memory and the number of bytes to display:

```
create myDATAS 01 c, 02 c, 03 c, 04 c,
hex
myDATAS 4 dump     \ displays :
3FFEE4EC                                                 01 02 03 04
```

Data stack monitor

The contents of the data stack can be displayed at any time using the word `.s` . Here is the definition of the word `.DEBUG` which exploits `.s` :

```
variable debugStack

: debugOn ( -- )
    -1 debugStack !
  ;

: debugOff ( -- )
    0 debugStack !
  ;

: .DEBUG
    debugStack @
    if
        cr ." STACK: " .s
        key drop
    then
  ;
```

To use `.DEBUG`, simply insert it in a strategic place in the word to be debugged:

```
\ example of use:
: myTEST
    128 32 do
        i  .DEBUG
        emit
    loop
  ;
```

Here, we will display the contents of the data stack after execution of word i in our `do loop` . We activate the focus and run `myTEST` :

```
debugOn
myTest
\ displays:
\ STACK: <1> 32
\ 2
\ STACK: <1> 33
\ 3
\ STACK: <1> 34
\ 4
\ STACK: <1> 35
\ 5
\ STACK: <1> 36
\ 6
\ STACK: <1> 37
\ 7
\ STACK: <1> 38
```

When debugging is enabled by `debugOn` , each display of the contents of the datastack pauses our `do loop`. Run `debugOff` so that the `myTEST word` executes normally.

Perform unit tests

eForth has the word assert allowing you to carry out tests. The best place to use this word is in a **tests.fs** file. Example :

```
$1234 100div  nip  $34  = assert
$1234 100div  drop $12  = assert
```

Here, we test the word `100div` which leaves the quotient and the remainder of the division by 256 (100 in hexadecimal) on the stack. The test should leave a true or false value on the stack. If the test returns a null value, `assert` generates an **ERROR** message.

Here is another example using `assert` :

```
$0080 bytesToUTF8 $c280   = assert
$0544 bytesToUTF8 $d584   = assert
$a894 bytesToUTF8 $eaa294 = assert
```

Here, we test the word `bytesToUTF8`. This word comes from code currently under development. The values to test come from the online UTF8 documentation. These three lines allow you to instantly test `bytesToUTF8` with several typical cases. If the word does not generate the expected result, `assert` will report that there is a test error.

Creating and using assert(

The word `assert` has a major drawback. If we carry out a lot of tests on different words, in a file, here **tests.fs**, we only get an error report, but no information on the test line which generated this error.

It turns out that the *gForth* version has the word `assert(`, the usage syntax of which is :

```
assert( 0 >gray  0 = )
assert( 1 >gray  1 = )
assert( 2 >gray  3 = )
```

The *gForth* code has been adapted to display the incorrect content. Here is the source code for this version:

```
-1 value ASSERT_LEVEL

variable assert-start

: assert(   ( -- )
    tib >in @ + assert-start !
    ASSERT_LEVEL 0= if
        POSTPONE (
    then
  ; immediate

: ) ( fl -- )
```

```
    0= if
        cr ." ASSERT : "
        assert-start @
        tib >in @ + over - 1- type
         -1 throw
    then
  ; immediate
```

In this code, we have an `ASSERT_LEVEL` value. If this value is set to zero, `assert(` behaves like the word `(`.

Next, we have an `assert-start` variable. This variable is used to store the location of `assert(` in the interpretation chain processed by eForth.

The word `)` tests the Boolean flag. If it is zero, it generates an error message and displays the code after `assert(` which is causing the test error.

If you are on a development project, here is an example of typical file chaining in **main.fs** :

```
s" gray.fs"     included
s" assert.fs"   included
s" tests.fs"    included
```

The g**ray.fs** file contains FORTH code currently being developed and fine-tuned. The **assert.fs** file contains the `assert(` code. Finally, our **tests.fs** file contains the battery of tests to be performed on the definitions currently being developed.

So, with a simple **main.fs** include sequence, the code under development is compiled, then it is instantly tested through the unit tests written in **tests.fs**.

The word `assert(` was written to display nothing if the tests were executed successfully.

This development strategy with unit testing allows you to quickly detect code errors if you modify a definition that is subject to unit testing.

Dictionary / Stack / Variables / Constants

Expand Dictionary

Forth belongs to the class of woven interpretive languages. This means that it can interpret commands typed on the console, as well as compile new subroutines and programs.

The Forth compiler is part of the language and special words are used to create new dictionary entries (i.e. words). The most important are `:` (start a new definition) and `;` (finishes the definition). Let's try this by typing :

```
: *+ * + ;
```

What happened? The action of `:` is to create a new dictionary entry named `*+` and switch from interpretation mode to compilation mode. In compile mode, the interpreter searches for words and, rather than executing them, installs pointers to their code. If the text is a number, instead of pushing it onto the stack, eFORTH Linux constructs the number in the dictionary space allocated for the new word, following special code that puts the stored number on the stack each time the word is executed. The execution action of `*+` is therefore to sequentially execute the previously defined words `*` and `+`.

Word `;` is special. It is an immediate word and it is always executed, even if the system is in compile mode. Which makes `;` is twofold. First, it installs code that returns control to the next external level of the interpreter, and second, it returns from compilation mode to interpretation mode.

Now let's try this new word :

```
decimal 5 6 7 *+ . \ display 47 ok<#,ram>
```

This example illustrates two main work activities in Forth : adding a new word to the dictionary, and trying it as soon as it has been defined.

Dictionary management

The word `forget` followed by the word to delete will remove all dictionary entries you have made since that word :

```
: test1 ;
: test2 ;
: test3 ;
forget test2  \ delete test2 and test3 in dictionnary
```

Stacks and reverse Polish notation

Forth has an explicitly visible stack that is used to pass numbers between words (commands). Using Forth effectively forces you to think in terms of the stack. This can be difficult at first, but as with anything, it gets much easier with practice.

In FORTH, The pile is analogous to a pile of cards with numbers written on them. Numbers are always added to the top of the stack and removed from the top of the stack. Eforth Linux integrates two stacks: the parameter stack and the return stack, each consisting of a number of cells that can hold 64-bit numbers.

The FORTH input line :

```
decimal 2 5 73 -16
```

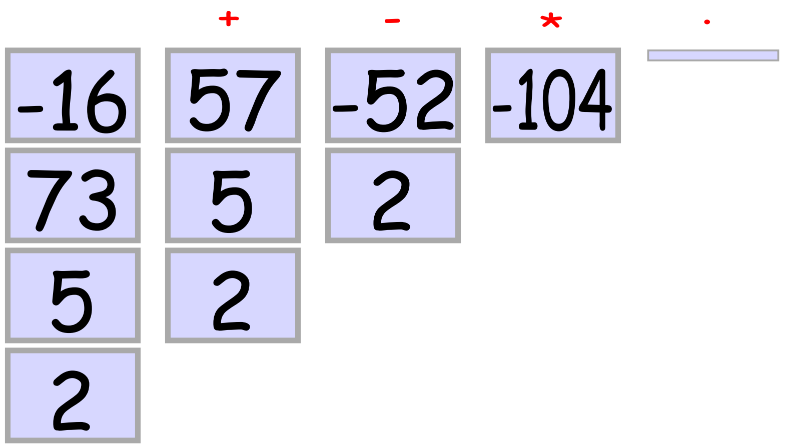
leaves the parameter stack as it is

Cell	Content	comment
0	-16	(TOS) Top of stack
1	73	(NOS) Next in stack
2	5	
3	2	

We will typically use zero-based relative numbering in Forth data structures such as stacks, arrays, and tables. Note that when a sequence of numbers is entered like this, the rightmost number becomes TOS and the leftmost number is at the bottom of the stack.

Let's continue with this:

```
+ - * .
```



The operations would produce successive stack operations :

After the two lines, the console displays :

```
decimal 2 5 73 -16  \ display: 2 5 73 -16 ok
+ - * .             \ display: -104 ok
```

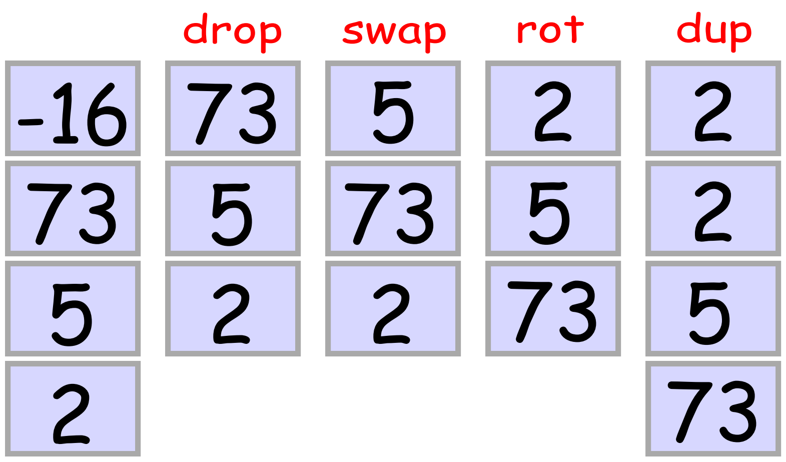
Note that eForth Linux conveniently displays the stack elements when interpreting each line and that the value of `-16` is displayed as a 64-bit unsigned integer. Furthermore, the word `.` consumes data value `-104`, leaving the stack empty. If we execute `.` on the now empty stack, the external interpreter aborts with a stack pointer error STACK UNDERFLOW ERROR.

The programming notation where the operands appear first, followed by the operator(s) is called Reverse Polish Notation (RPN).

Handling the parameter stack

Being a stack-based system, eForth Linux must provide ways to put numbers on the stack, remove them and rearrange their order. We have already seen that we can put numbers on the stack simply by typing them. We can also integrate numbers into the definition of a FORTH word.

The word `drop` removes a number from the top of the stack thus putting the next one on top. The word `swap` exchanges the first 2 numbers. `dup` copies the number at the top, pushing all other numbers down. `rot` rotates the first 3 numbers. These actions are



presented below.

The Return Stack and Its Uses

When compiling a new word, eForth Linux establishes links between the calling word and previously defined words that are to be invoked by the execution of the new word. This linking mechanism, at runtime, uses the return stack. The address of the next word to be invoked is placed on the back stack so that when the current word has finished executing, the system knows where to move to the next word. Since words can be nested, there must be a stack of these return addresses.

In addition to serving as a reservoir of return addresses, the user can also store and retrieve from the return stack, but this must be done carefully because the return stack is essential to program execution. If you use the return stack for temporary storage, you must return it to its original state, otherwise you will likely crash eForth Linux. Despite the danger, there are times when using return stack as temporary storage can make your code less complex.

To store on the return stack, use `>r` to move the top of the parameter stack to the top of the return stack. To retrieve a value, `r>` moves the top value from the return stack back to the top of the parameter stack. To simply remove a value from the top of the return stack, there is the word `rdrop`. The word `r@` copies the top of the return stack back into the parameter stack.

Memory usage

In eForth Linux, 64-bit numbers are fetched from memory to the stack by the word `@` (fetch) and stored from the top to memory by the word `!` (store). `@` expects an address on the stack and replaces the address with its contents. `!` expects a number and an address to store it. It places the number in the memory location referenced by the address, consuming both parameters in the process.

Unsigned numbers that represent 8-bit (byte) values can be placed in character-sized characters. memory cells using `c@` and `c!`.

```
create testVar
    cell allot
$f7 testVar c!
testVar c@ .    \ display 247
```

Variables

A variable is a named location in memory that can store a number, such as the intermediate result of a calculation, off the stack. For example :

```
variable x
```

creates a storage location named `x`, which executes leaving the address of its storage location at the top of the stack :

```
x .    \ display address
```

We can then retrieve or store at this address :

```
variable x
3 x !
x @ .   \ display: 3
```

Constants

A constant is a number that you would not want to change while a program is running. The result of executing the word associated with a constant is the value of the data remaining on the stack.

```
\ define VSPI pins
19 constant VSPI_MISO
23 constant VSPI_MOSI
18 constant VSPI_SCLK
```

```
05 constant VSPI_CS

\ define SPI frequency port
4000000 constant SPI_FREQ

\ select SPI vocabulary
only FORTH  SPI also

\ initialize the SPI port
: init.VSPI ( -- )
    VSPI_CS OUTPUT pinMode
    VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin
    SPI_FREQ SPI.setFrequency
  ;
```

Pseudo-constant values

A value defined with `value` is a hybrid type of `variable` and `constant`. We set and initialize a value and it is invoked as we would a constant. We can also change a value like we can change a variable.

```
decimal
13 value thirteen
thirteen .      \ display: 13
47 to thirteen
thirteen .      \ display: 47
```

The word `to` also works in word definitions, replacing the value following it with whatever is currently at the top of the stack. You need to be careful that `to` is followed by a value defined by value and not something else.

Basic tools for memory allocation

The words `create` and `allot` are the basic tools for reserving memory space and attaching a label to it. For example, the following transcription shows a new dictionary entry `graphic-array` :

```
create graphic-array ( --- addr )
    %00000000 c,
    %00000010 c,
    %00000100 c,
    %00001000 c,
    %00010000 c,
    %00100000 c,
    %01000000 c,
    %10000000 c,
```

When executed, the word `graphic-array` stacks the address of the first entry.

We can now access the memory allocated to `graphic-array` using the fetch and store words explained earlier. To calculate the address of the third byte assigned to `graphic-array` we can write `graphic-array 2 +`, remembering that the indices start at 0.

```
30 graphic-array  2 + c!
graphic-array  2 + c@ .     \ display 30
```

Local variables with eForth Linux

Introduction

The FORTH language processes data primarily through the data stack. This very simple mechanism offers unrivaled performance. Conversely, following the flow of data can quickly become complex. Local variables offer an interesting alternative.

The fake stack comment

If you follow the different FORTH examples, you will have noticed the stack comments framed by (and `)` . Example:

```
\ addition two unsigned values, leaves sum and carry on the stack
: um+ ( u1 u2 -- sum carry )
    \ here the definition
  ;
```

Here, the comment `( u1 u2 -- sum carry )` has absolutely no action on the rest of the FORTH code. This is pure commentary.

When preparing a complex definition, the solution is to use local variables framed by `{` and `}` . Example :

```
: 2OVER { a b c d }
    a b c d a b
  ;
```

We define four local variables `a b c` and `d`.

The words `{` and `}` are similar to the words `(` and `)` but do not have the same effect at all. Codes placed between `{` and `}` are local variables. The only constraint: do not use variable names that could be FORTH words from the FORTH dictionary. We might as well have written our example like this :

```
: 2OVER { varA varB varC varD }
    varA varB varC varD varA varB
  ;
```

Each variable will take the value of the data stack in the order of their deposit on the data stack. here, 1 goes into `varA`, 2 into `varB`, etc.:

```
--> 1 2 3 4
 ok
1 2 3 4 --> 2over
 ok
1 2 3 4 1 2 -->
```

Our fake stack comment can be completed like this :

```
: 2OVER { varA varB varC varD -- varA varB varC varD varA varB }
```

The characters following `--` have no effect. The only point is to make our fake comment look like a real stack comment.

Action on local variables

Local variables act exactly like pseudo-variables defined by `value`. Example :

```
: 3x+1 { var -- sum }
    var 3 * 1 +
  ;
```

Has the same effect as :

```
0 value var
: 3x+1  ( var -- sum )
    to var
    var 3 * 1 +
  ;
```

In this example, `var` is defined explicitly by `value`.

We assign a value to a local variable with the word `to` or `+to` to increment the content of a local variable. In this example, we add a local variable `result` initialized to zero in the code of our word:

```
: a+bEXP2 { varA varB -- (a+b)EXP2 }
    0 { result }
    varA varA *      to result
    varB varB *     +to result
    varA varB * 2 * +to result
    result
  ;
```

Isn't it more readable than this?

```
: a+bEXP2 ( varA varB -- result )
    2dup
    * 2 * >r
    dup *
    swap dup * +
    r> +
  ;
```

Here is a final example, the definition of the word `um+` which adds two unsigned integers and leaves the sum and the overflow value of this sum on the data stack:

```
\ add two unsigned integers, leaves sum and carry on the stack
: um+ { u1 u2 -- sum carry }
    0 { sum }
    cell for
        aft
            u1 $100 /mod to u1
            u2 $100 /mod to u2
            +
            cell 1- i - 8 * lshift  +to sum
        then
    next
    sum
    u1 u2 + abs
  ;
```

Here is a more complex example, rewriting **DUMP** using local variables:

```
\ local variables in DUMP:
\ START_ADDR       \ first address for dump
\ END_ADDR         \ last address for dump
\ 0START_ADDR      \ first address for loop in dump
\ LINES            \ number of lines for dump loop
\ myBASE           \ current numerical base
internals
: dump ( start len -- )
    cr cr ." --addr---   "
    ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----"
    2dup + { END_ADDR }                 \ store latest address to dump
    swap { START_ADDR }                 \ store START address to dump
    START_ADDR 16 / 16 * { 0START_ADDR } \ calc. addr for loop start
    16 / 1+ { LINES }
    base @ { myBASE }                   \ save current base
    hex
    \ outer loop
    LINES 0 do
        0START_ADDR i 16 * +            \ calc start address for current line
        cr <# # # # #  [char] - hold # # # # #> type
        space space     \ and display address
        \ first inner loop, display bytes
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            ca@ <# # # #> type space \ display byte in format: NN
        loop
        space
        \ second inner loop, display chars
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            \ display char if code in interval 32-127
            ca@     dup 32 < over 127 > or
            if      drop [char] . emit
            else    emit
            then
        loop
    loop
    myBASE base !                  \ restore current base
    cr cr
  ;
forth
```

The use of local variables greatly simplifies data manipulation on stacks. The code is more readable. Note that it is not necessary to pre-declare these local variables, it is enough to designate them when using them, for example: **`base @ { myBASE }`**.

WARNING: if you use local variables in a definition, no longer use the words **`>r`** and **`r>`**, otherwise you risk disrupting the management of local variables. Just look at the decompilation of this version of **`DUMP`** to understand the reason for this warning:

```
: dump  cr cr s" --addr---  " type
    s" 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----" type
    2dup + >R SWAP >R -4 local@ 16 / 16 * >R 16 / 1+ >R base @ >R
    hex -8 local@ 0 (do) -20 local@ R@ 16 * + cr
    <# # # # # 45 hold # # # # #> type space space
    16 0 (do) -28 local@ j 16 * R@ + + CA@ <# # # #> type space 1 (+loop)
    0BRANCH rdrop rdrop space 16 0 (do) -28 local@ j 16 * R@ + + CA@ DUP 32 < OVER 127 > OR
    0BRANCH DROP 46 emit BRANCH emit 1 (+loop) 0BRANCH rdrop rdrop 1 (+loop)
    0BRANCH rdrop rdrop -4 local@ base ! cr cr rdrop rdrop rdrop rdrop rdrop ;
```

Data structures for eForth Linux

Preamble

Eforth Linux is a 64-bit version of the FORTH language. Those who have practiced FORTH since its beginnings have programmed with 16-bit versions. This data size is determined by the size of the elements deposited on the data stack. To find out the size in bytes of the elements, you must execute the word `cell`. Running this word for ESP32forth :

```
cell .  \ display 8
```

The value 8 means that the size of the elements placed on the data stack is 8 bytes, or 8x8 bits = 64 bits.

With a 16-bit FORTH version, cell will stack the value 2. Likewise, if you use a 32-bit version, cell will stack the value 4.

Tables in FORTH

Let's start with fairly simple structures : tables. We will only discuss one- or two-dimensional arrays.

One-dimensional 64-bit data array

This is the simplest type of table. To create a table of this type, we use the word `create` followed by the name of the table to create :

```
create temperatures
    34 ,    37 ,    42 ,    36 ,    25 ,    12 ,
temperatures        \ push addr on stack
    0 cell *        \ calculate offset 0
    +               \ add offset to addr
    @ .             \ display 34

temperatures        \ push addr on stack
    1 cell *        \ calculate offset 0
    +               \ add offset to addr
    @ .             \ display 37
```

We can factor the access code to the desired value by defining a word which will calculate this address :

```
: temp@ ( index --  value )
    cell * temperatures + @
  ;
0 temp@ .   \ display 34
2 temp@ .   \ display 42
```

You will notice that for n values stored in this table, here 6 values, the access index must always be in the interval [0..n-1].

Words for table definitions

Here's how to create a word definition of one-dimensional integer arrays :

```
: array ( comp: --  | exec: index  -- addr )
    create
    does>
        swap cell * +
  ;
array myTemps
    21 ,    32 ,    45 ,    44 ,    28 ,    12 ,
0 myTemps @ .   \ display 21
5 myTemps @ .   \ display 12
```

In our example, we store 6 values between 0 and 255. It is easy to create a variant of `array` to manage our data in a more compact way :

```
: arrayC ( comp: --  | exec: index  -- addr )
    create
    does>
        +
  ;
arrayC myCTemps
    21 c,   32 c,   45 c,   44 c,   28 c,   12 c,
0 myCTemps c@ .     \ display 21
5 myCTemps c@ .     \ display 12
```

With this variant, the same values are stored in four times less memory space.

Read and write in a table

It is entirely possible to create an empty array of n elements and write and read values in this array :

```
arrayC myCTemps
    6 allot             \ allocate 6 bytes
    0 myCTemps 6 0 fill \ fill this 6 bytes with value 0
32 0 myCTemps c!        \ store 32 in myCTemps[0]
25 5 myCTemps c!        \ store 25 in myCTemps[5]
0 myCTemps c@ .         \ display 32
```

In our example, the array contains 6 elements. With eForth Linux, there is enough memory space to process much larger arrays, with 1,000 or 10,000 elements for example. It's easy to create multi-dimensional tables. Example of a two-dimensional array :

```
63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
    SCR_WIDTH SCR_HEIGHT * allot            \ allocate 63 * 16 bytes
    mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill \ fill this memory with 'space'
```

Here, we define a two-dimensional table named `mySCREEN` which will be a virtual screen of 16 rows and 63 columns.

Simply reserve a memory space which is the product of the dimensions X and Y of the table to use.

Management of complex structures

ESP32forth has the `structures` vocabulary. The content of this vocabulary makes it possible to define complex data structures.

Here is simple example of structure :

```
structures
struct YMDHMS
    ptr field >year
    ptr field >month
    ptr field >day
    ptr field >hour
    ptr field >min
    ptr field >sec
```

Here, we define the `YMDHMS` structure. This structure manages the `>year` `>month` `>day` `>hour` `>min` and `>sec` pointers.

The sole purpose of the `YMDHMS` word is to initialize and group the pointers in the complex structure. Here is how these pointers are used :

```
create DateTime
    YMDHMS allot

2022 DateTime >year  !
  03 DateTime >month !
  21 DateTime >day   !
  22 DateTime >hour  !
  36 DateTime >min   !
  15 DateTime >sec   !

: .date ( date -- )     \ date is address of structure
    >r
    ."  YEAR: " r@ >year    @ . cr
    ." MONTH: " r@ >month   @ . cr
    ."   DAY: " r@ >day     @ . cr
    ."    HH: " r@ >hour    @ . cr
    ."    MM: " r@ >min     @ . cr
    ."    SS: " r@ >sec     @ . cr
    r> drop
  ;
```

```
DateTime .date
```

We defined word `DateTime` as simple table of 6 consecutive cells each 32 bits. Access to each cell is with specific pointer. We can redefine our structure `YMDHMS` with `i8` pointers to bytes.

```
structures
struct cYMDHMS
    ptr field >year
    i8  field >month
    i8  field >day
    i8  field >hour
    i8  field >min
    i8  field >sec

create cDateTime
    cYMDHMS allot

2022 cDateTime >year   !
  03 cDateTime >month c!
  21 cDateTime >day   c!
  22 cDateTime >hour  c!
  36 cDateTime >min   c!
  15 cDateTime >sec   c!

: .cDate ( date -- )
    >r
    ."  YEAR: " r@ >year     @ . cr
    ." MONTH: " r@ >month   c@ . cr
    ."   DAY: " r@ >day     c@ . cr
    ."    HH: " r@ >hour    c@ . cr
    ."    MM: " r@ >min     c@ . cr
    ."    SS: " r@ >sec     c@ . cr
    r> drop
  ;
cDateTime .cDate    \ displays:
\  YEAR: 2022
\ MONTH: 3
\   DAY: 21
\    HH: 22
\    MM: 36
\    SS: 15
```

In this `cYMDHMS` structure, we kept the year in 32-bit format and reduced all other values to 8-bit integers. We see, in the `.cDate` code, that the use of pointers allows easy access to each element of our complex structure....

Real numbers with eForth Linux

If we test the operation `1 3 /` in FORTH language, the result will be 0.

It's not surprising. Basically, eForth Linux only uses 64-bit integers via the data stack. Integers offer certain advantages:

- speed of processing;
- result of calculations without risk of drift in the event of iterations;
- suitable for almost all situations.

Even in trigonometric calculations, we can use a table of integers. Simply create a table with 90 values, where each value corresponds to the sine of an angle, multiplied by 1000.

But integers also have limits:

- impossible results for simple division calculations, like our 1/3 example;
- requires complex manipulations to apply physics formulas.

Since version 7.0.6.5, ESP32forth includes operators dealing with real numbers.

Real numbers are also called floating point numbers.

The real ones with eForth Linux

In order to distinguish real numbers, they must end with the letter "e":

```
3           \ push 3 on the normal stack
3e          \ push 3 on the real stack
5.21e f.    \ display 5.210000
```

It's the word `f.` which allows you to display a real number located at the top of the reals stack.

Real number accuracy with eForth Linux

The word `set-precision` allows you to indicate the number of decimal places to display after the decimal point. Let's see this with the constant `pi` :

```
pi f.       \ display 3.141592
4 set-precision
pi f.       \ display 3.1415
```

The limit precision for processing real numbers with eForth Linux is six decimal places :

```
12 set-precision
1.987654321e f.       \ display 1.987654668777
```

If we reduce the display precision of real numbers below 6, the calculations will still be carried out with a precision to 6 decimal places.

Real constants and variables

A real constant is defined with the word `fconstant` :

```
0.693147e fconstant ln2    \ natural logarithm of 2
```

A real variable is defined with the word `fvariable` :

```
fvariable intensity
170e 12e F/ intensity SF!   \ I=P/U   ---   P=170w   U=12V
intensity SF@ f.            \ display 14.166669
```

ATTENTION: all real numbers pass through the **real number stack** . In the case of a real variable, only the address pointing to the real value passes through the data stack.

The word `SF!` stores a real value at the address or variable pointed to by its memory address. Executing a real variable places the memory address on the classic data stack.

The word `SF@` stacks the real value pointed to by its memory address.

Arithmetic operators on real numbers

eForth Linux has four arithmetic operators `F+ F- F* F/` :

```
1.23e 4.56e F+ f.   \ display 5.790000      1.23-4.56
1.23e 4.56e F- f.   \ display -3.330000     1.23-4.56
1.23e 4.56e F* f.   \ display 5.608800      1.23*4.56
1.23e 4.56e F/ f.   \ display 0.269736      1.23/4.56
```

ESP32forth also has these words:

- `1/F` calculates the inverse of a real number;
- `fsqrt` calculates the square root of a real number.

```
5e 1/F f.       \ display 0.200000      1/5
5e fsqrt f.     \ display 2.236068      sqrt(5)
```

Mathematical operators on real numbers

eForth Linux has several mathematical operators:

- `F**` raises a real r_val to the power r_exp
- `FATAN2` calculates the angle in radian from the tangent.

- `FCOS` (r1 -- r2) Calculates the cosine of an angle expressed in radians.
- `FEXP` (ln-r -- r) calculates the real corresponding to e EXP r
- `FLN` (r -- ln-r) calculates the natural logarithm of a real number.
- `FSIN` (r1 -- r2) calculates the sine of an angle expressed in radians.
- `FSINCOS` (r1 -- rcos rsin) calculates the cosine and sine of an angle expressed in radians.

Some examples :

```
  2e 3e f** f.    \ display 8.000000
  2e 4e f** f.    \ display 16.000000
 10e 1.5e f** f.  \ display 31.622776

4.605170e FEXP F.    \ display 100.000018

pi 4e f/
FSINCOS f. f.   \ display 0.707106 0.707106
pi 2e f/
FSINCOS f. f.   \ display 0.000000 1.000000
```

Logical operators on real numbers

eForth Linux also allows you to perform logic tests on real data:

- `F0<` (r -- fl) tests if a real number is less than zero.
- `F0=` (r -- fl) indicates true if the real is zero.
- `f<` (r1 r2 -- fl) fl is true if r1 < r2.
- `f<=` (r1 r2 -- fl) fl is true if r1 <= r2.
- `f<>` (r1 r2 -- fl) fl is true if r1 <> r2.
- `f=` (r1 r2 -- fl) fl is true if r1 = r2.
- `f>` (r1 r2 -- fl) fl is true if r1 > r2.
- `f>=` (r1 r2 -- fl) fl is true if r1 >= r2.

Integer ↔ real transformations

eForth Linux has two words to transform integers into reals and vice versa:

- `F>S` (r -- n) converts a real to an integer. Leave the integer part on the data stack if the real has decimal parts.
- `S>F` (n -- r: r) converts an integer to a real number and transfers this real number to the reals stack.

Example :

```
35 S>F
F.   \ display 35.000000

3.5e F>S .   \ display 3
```

Displaying numbers and character strings

Change of numerical base

FORTH does not process just any numbers. The ones you used when trying the previous examples are single-precision signed integers. These numbers can be processed in any number base, with all number bases between 2 and 36 being valid :

```
255 HEX. DECIMAL    \displays    FF
```

You can choose an even larger numerical base, but the available symbols will fall outside the alpha-numeric set [0..9,A..Z] and risk becoming inconsistent.

The current numerical base is controlled by a variable named **BASE** and whose content can be modified. So, to switch to binary, simply store the value **2** in **BASE** . Example:

```
2 BASE !
```

and type **DECIMAL** to return to the decimal numeric base.

ESP32forth has two pre-defined words allowing you to select different numerical bases:

- **DECIMAL** to select the decimal numeric base. This is the numerical base taken by default when starting ESP32forth;
- **HEX** to select the hexadecimal numeric base.
- **BINARY** to select the binary numeric base.

Upon selection of one of these numerical bases, the literal numbers will be interpreted, displayed or processed in this base. Any number previously entered in a number base other than the current number base is automatically converted to the current number base. Example :

```
DECIMAL       \ base to decimal
255           \ stacks 255
HEX           \ selects hexadecimal base
1+            \ increments 255 becomes 256
.             \ displays 100
```

One can define one's own numerical base by defining the appropriate word or by storing this base in **BASE**. Example :

```
: BINARY ( ---)         \ selects the binary number base
    2 BASE ! ;
DECIMAL 255 BINARY .    \ displays 11111111
```

The contents of **BASE** can be stacked like the contents of any other variable :

```
VARIABLE RANGE_BASE       \ RANGE-BASE variable definition
BASE @ RANGE_BASE !       \ storage BASE contents in RANGE-BASE
HEX FF 10 + .             \ displays 10F
RANGE_BASE @ BASE !       \ restores BASE with contents of RANGE-BASE
```

In a definition `:` , the contents of `BASE` can pass through the return stack :

```
: OPERATION ( ---)
    BASE @ >R         \ stores BASE on back stack
    HEX FF 10 + .     \ operation of the previous example
    R> BASE ! ;       \ restores initial BASE value
```

WARNING : the words `>R` and `R>` cannot be used in interpreted mode. You can only use these words in a definition that will be compiled.

Definition of new display formats

Forth has primitives allowing you to adapt the display of a number to any format. With ESP32forth, these primitives deal with integers numbers :

- `<#` begins a format definition sequence;
- `#` inserts a digit into a format definition sequence;
- `#S` is equivalent to a succession of `#` ;
- `HOLD` inserts a character into a format definition;
- `#>` completes a format definition and leaves on the stack the address and length of the string containing the number to display.

These words can only be used within a definition. Example, either to display a number expressing an amount denominated in euros with the comma as a decimal separator :

```
: .EUROS ( n ---)
  <# # # [char] , hold #S #>
  type space ." EUR" ;
1245 .euros
```

Execution examples:

```
35   .EUROS               \ displays     0,35 EUR
3575 .EUROS               \ displays     35,75 EUR
1015 3575 + .EUROS        \ displays     45,90 EUR
```

In the `.EUROS` definition, the word `<#` begins the display format definition sequence. The two words `#` place the ones and tens digits in the character string. The word `HOLD` places the character `,` (comma) following the two digits on the right, the word `#S` completes the display format with the non-zero digits following `,` . The word `#>` closes the format definition and places on the stack the address and the length of the string containing the digits of the number to display. The word `TYPE` displays this character string.

At runtime, a display format sequence deals exclusively with signed or unsigned 32-bit integers. The concatenation of the different elements of the string is done from right to left, i.e. starting with the least significant digits.

The processing of a number by a display format sequence is executed based on the current numeric base. The numerical base can be modified between two digits.

Here is a more complex example demonstrating the compactness of FORTH. This involves writing a program converting any number of seconds into HH:MM:SS format:

```
 :00 ( ---)
    DECIMAL #            \ insert digit unit in decimal
    6 BASE !             \ base 6 selection
    #                    \ insert digit ten
    [char] : HOLD        \ insertion character :
    DECIMAL ;            \ return decimal base
: HMS ( n ---)           \ displays number seconds format HH:MM:SS
    <# :00 :00 #S #> TYPE SPACE ;
```

Execution examples :

```
59 HMS     \ displays     0:00:59
60 HMS     \ displays     0:01:00
4500 HMS   \ displays     1:15:00
```

Explanation: The system for displaying seconds and minutes is called the sexagesimal system. Units are expressed in decimal numerical base, **tens** are expressed in base six. The word `:00` manages the conversion of units and tens in these two bases for formatting the numbers corresponding to seconds and minutes. For times, the numbers are all decimal.

Another example, to define a program converting a single precision decimal integer into binary and displaying it in the format bbbb bbbb bbbb bbbb:

```
: FOUR-DIGITS ( ---)
    # # # # 32 HOLD ;
: AFB ( n ---)               \ format 4 digits and a space
    BASE @ >R                \ Current database backup
    2 BASE !                 \ Binary digital base selection
    <#
    4 0 DO                   \ Format Loop
        FOUR-DIGITS
    LOOP
    #> TYPE SPACE            \ Binary display
    R> BASE ! ;              \ Initial digital base restoration
```

Execution example :

```
DECIMAL 12 AFB     \ displays     0000 0000 0000 0110
HEX 3FC5 AFB       \ displays     0011 1111 1100 0101
```

Another example is to create a telephone diary where one or more telephone numbers are associated with a surname. We define a word by surname :

```
: .## ( ---)
    # # [char] . HOLD ;
: .TEL ( d ---)
    CR <# .## .## .## .## # # #> TYPE CR ;
: WACHOWSKI ( ---)
    0618051254 .TEL ;
WACHOWSKI    \ displays: 06.18.05.12.54
```

This calendar, which can be compiled from a source file, is easily editable, and although the names are not classified, the search is extremely fast.

Displaying characters and character strings

A character is displayed using the word `EMIT` :

```
65 EMIT            \ displays A
```

The displayable characters are in the range 32..255. Codes between 0 and 31 will also be displayed, subject to certain characters being executed as control codes. Here is a definition showing the entire character set of the ASCII table:

```
variable #out
: #out+! ( n -- )
    #out +!                  \ increment #out
  ;
: (.)   ( n -- a l )
  DUP ABS <# #S ROT SIGN #>
;
: .R   ( n l -- )
  >R (.) R> OVER - SPACES TYPE
;
: ASCII-SET ( ---)
    cr 0 #out !
    128 32
    DO
        I 3 .R SPACE         \ displays character code
        4 #out+!
        I EMIT 2 SPACES      \ displays character
        3 #out+!
        #out @ 77 =
        IF
            CR   0 #out !
        THEN
    LOOP ;
```

Running **ASCII-SET** displays the ASCII codes and characters whose code is between 32 and 127. To display the equivalent table with the ASCII codes in hexadecimal, type **HEX ASCII-SET**:

```
hex  ASCII-SET
 20      21 !   22 "   23 #   24 $   25 %   26 &   27 '   28 (   29 )   2A *
 2B +   2C ,   2D -   2E .   2F /   30 0   31 1   32 2   33 3   34 4   35 5
 36 6   37 7   38 8   39 9   3A :   3B ;   3C <   3D =   3E >   3F ?   40 @
 41 A   42 B   43 C   44 D   45 E   46 F   47 G   48 H   49 I   4A J   4B K
 4C L   4D M   4E N   4F O   50 P   51 Q   52 R   53 S   54 T   55 U   56 V
 57 W   58 X   59 Y   5A Z   5B [   5C \   5D ]   5E ^   5F _   60 `   61 a
 62 b   63 c   64 d   65 e   66 f   67 g   68 h   69 i   6A j   6B k   6C l
 6D m   6E n   6F o   70 p   71 q   72 r   73 s   74 t   75 u   76 v   77 w
 78 x   79 y   7A z   7B {   7C |   7D }   7E ~   7F     ok
```

Character strings are displayed in various ways. The first, usable in compilation only, displays a character string delimited by the character " (quote mark):

```
: TITLE ." GENERAL MENU";
    TITLE     \ displays     GENERAL MENU
```

The string is separated from the word **."** by at least one space character.

A character string can also be compiled by the word **s"** and delimited by the character **"** (quotation mark):

```
: LINE1 ( --- adr len)
    S" E..Data logging" ;
```

Executing **LINE1** places the address and length of the string compiled in the definition on the data stack. The display is carried out by the word **TYPE:**

```
LINE1 TYPE     \ displays      E..Data logging
```

At the end of displaying a character string, the line break must be triggered if desired:

```
CR TITLE CR CR LINE1 CR TYPE
\ displays:
\ GENERAL MENU
\
\ E..Data logging
```

One or more spaces can be added at the start or end of the display of an alphanumeric string :

```
SPACE          \ displays a space character
10 SPACES      \ displays 10 space characters
```

String variables

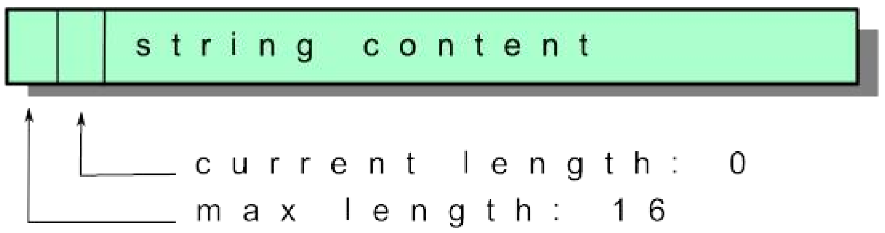
Alpha-numeric text variables do not exist natively in ESP32forth. Here is the first attempt to define the word `string` :

```
\ define a strvar
: string  ( comp: n --- names_strvar | exec: --- addr len )
    create
        dup
        c,      \ n is maxlength
        0 c,    \ 0 is real length
        allot
    does>
        2 +
        dup 1 - c@
  ;
```

A character string variable is defined like this:

```
16 string strState
```

Here is how the memory space reserved for this text variable is organized:



Text variable management word code

Here is the complete source code for managing text variables:

```
DEFINED? --str [if] forget --str  [then]
create --str


\ compare two strings
: $= ( addr1 len1 addr2 len2 --- fl)
    str=
  ;

\ define a strvar
: string ( n --- names_strvar )
    create
        dup
        ,                       \ n is maxlength
        0 ,                     \ 0 is real length
        allot
    does>
        cell+ cell+
        dup cell - @
    ;

\ get maxlength of a string
: maxlen$  ( strvar --- strvar maxlen )
    over cell - cell - @
    ;

\ store str into strvar
: $!  ( str strvar --- )
    maxlen$                  \ get maxlength of strvar
    nip rot min              \ keep min length
    2dup swap cell - !       \ store real length
    cmove                    \ copy string
    ;

\ Example:
\ : s1
\     s" this is constant string" ;
\ 200 string test
\ s1 test $!

\ set length of a string to zero
: 0$! ( addr len -- )
    drop 0 swap cell - !
  ;

\ extract n chars right from string
: right$  ( str1 n --- str2 )
    0 max over min >r + r@ - r>
    ;

\ extract n chars left frop string
: left$  ( str1 n --- str2 )
    0 max min
    ;

\ extract n chars from pos in string
: mid$  ( str1 pos len --- str2 )
    >r over swap - right$ r> left$
    ;

\ append char c to string
: c+$!  ( c str1 -- )
    over >r
    + c!
    r> cell - dup @ 1+ swap !
    ;
```

```
\ work only with strings. Don't use with other arrays
: input$ ( addr len -- )
    over swap maxlen$ nip accept
    swap cell - !
  ;
```

Creating an alphanumeric character string is very simple :

```
64 string myNewString
```

Here we create an alphanumeric variable `myNewString` which can contain up to 64 characters.

To display the contents of an alphanumeric variable, simply use `type` . Example :

```
s" This is my first example.." myNewString $!
myNewString type    \ display: This is my first example..
```

If we try to save a character string longer than the maximum size of our alphanumeric variable, the string will be truncated:

```
s" This is a very long string, with more than 64 characters. It can't store complete"
myNewString $!
myNewString type
\ displays: This is a very long string, with more than 64 characters. It can
```

Adding character to an alphanumeric variable

Some devices, the LoRa transmitter for example, require processing command lines containing the non-alphanumeric characters The word `c+$!` allows this code insertion:

```
32 string AT_BAND
s" AT+BAND=868500000" AT_BAND $!   \ set frequency at 865.5 Mhz
$0a AT_BAND c+$!
$0d AT_BAND c+$!     \ add CR LF code at end of command
```

The memory dump of the contents of our alphanumeric variable `AT_BAND` confirms the presence of the two control characters at the end of the string:

```
--> AT_BAND dump
--addr--- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ------chars-----
3FFF-8620 8C 84 FF 3F 20 00 00 00 13 00 00 00 41 54 2B 42 ...? .......AT+B
3FFF-8630 41 4E 44 3D 38 36 38 35 30 30 30 30 30 0A 0D BD AND=868500000...
OK
```

Here is a clever way to create an alphanumeric variable allowing you to transmit a carriage return, a **CR+LF** compatible with the end of commands for the LoRa transmitter:

```
2 string $crlf
$0d $crlf c+$!
$0a $crlf c+$!

: crlf ( -- )        \ same action as cr, but adapted for LoRa
    $crlf type
  ;
```

Delayed action words

Deferred action words are defined by the definition word `defer`. To understand the mechanisms and the interest in exploiting this type of word, let's look in more detail at the functioning of the internal interpreter of the FORTH language.

Any definition compiled by `:` (colon) contains a sequence of coded addresses corresponding to the code fields of the words previously compiled. At the heart of the FORTH system, the word `EXECUTE` accepts as parameters these code field addresses, addresses which we abbreviate by **cfa** for Code Field Address. Every FORTH word has a **cfa** and this address is used by the internal FORTH interpreter :

```
' <word>
\ drops the cfa of <word> onto the data stack
```

Example:

```
' WORDS
\ stacks the WORDS cfa.
```

From this **cfa** , known as the only literal value, the execution of the word can be carried out with `EXECUTE`:

```
' WORDS EXECUTE
\ executes WORDS
```

Of course, it would have been easier to type `WORDS` directly . From the moment a **cfa** is available as the only literal value, it can be manipulated and notably stored in a variable :

```
variable vector
' WORDS vector !
vector @ .
\ displays cfa of WORDS stored in vector variable
```

You can run `WORDS` indirectly from the contents of `vector`:

```
vector @ EXECUTE
```

This launches the execution of the word whose **cfa** was stored in the `vector` variable then put back on the stack before use by `EXECUTE`.

This is a similar mechanism that is exploited by the execution part of the `defer` definition word. To simplify, `defer` creates a header in the dictionary, like a `variable` or `constant`, but instead of simply dropping an address or value on the stack, it starts execution of the word whose cfa **was** stored in the parametric area of the word defined by `defer` .

Definition and usage of words with defer

The initialization of a word defined by `defer` is carried out by `is` :

```
defer vector
' words is vector
```

Executing `vector` causes the word whose **cfa** was previously assigned to be executed:

```
vector        \ exécute   words
```

A word created by `defer` is used to execute another word without explicitly calling on that word. The main interest of this type of word lies above all in the possibility of modifying the word to be executed:

```
' page is vector
```

`vector` now executes `page` and no longer `words`.

We essentially use the words defined by `defer` in two situations:

- definition of a forward reference;
- definition of a word depending on the operating context.

In the first case, the definition of a before reference makes it possible to overcome the constraints of the sacrosanct precedence of definitions.

In the second case, the definition of a word depending on the operating context makes it possible to resolve most of the interfacing problems with an evolving software environment, to maintain the portability of applications, to adapt the behavior of a program to situations controlled by various parameters without harming software performance.

Setting a Forward Reference

Unlike other compilers, FORTH does not allow a word to be compiled into a definition before it is defined. This is the principle of precedence of definitions :

```
: word1 ( ---)    word2    ;
: word2 ( ---)   ;
```

This generates an error when compiling `word1` , because `word2` is not yet defined. Here's how to get around this constraint with `defer` :

```
defer word2
: word1 ( ---)    word2    ;
: (word2) ( ---)   ;
' (word2) is  word2
```

This time `word2` compiled without errors. It is not necessary to assign a cfa to the vectorized execution word `word2` . It is only after the definition of `(word2)` that the parameter area of `word2` is updated. After assignment of the vectorized execution word `word2` , `word1` will be able to execute the content of its definition without error. The exploitation of words created by `defer` in this situation must remain exceptional.

A practical case

You have an application to create, with displays in two languages. Here is a clever way by exploiting a word defined by defer to generate text in French or English. To begin, we will simply create a table of days in English:

```
:noname s" Saterday" ;
:noname s" Friday" ;
:noname s" Thursday" ;
:noname s" Wednesday" ;
:noname s" Tuesday" ;
:noname s" Monday" ;
:noname s" Sunday" ;

create ENdayNames ( --- addr)
     , , , , , , ,
```

Then we create a similar table for the days in French:

```
:noname s" Samedi" ;
:noname s" Vendredi" ;
:noname s" Jeudi" ;
:noname s" Mercredi" ;
:noname s" Mardi" ;
:noname s" Lundi" ;
:noname s" Dimanche" ;

create FRdayNames ( -- addr)
     , , , , , , ,
```

Finally we create our deferred action word **dayNames** and how to initialize it:

```
defer dayNames

: in-ENGLISH
    ['] ENdayNames is dayNames  ;

: in-FRENCH
    ['] FRdayNames is dayNames  ;
```

Here are now the words to manage these two tables:

```
: _getString { array length -- addr len }
    array
    swap cell *
    + @ execute
    length ?dup if
        min
    then
  ;

10 value dayLength
: getDay ( n -- addr len )       \ n interval [0..6]
```

```
    dayNames dayLength _getString
  ;
```

Here's what running **`getDay does`** :

```
: .dayList { size -- }
    size to dayLength
    7 0 do
        i getDay type space
    loop
  ;

in-ENGLISH 3 .dayList cr   \ display :  Sun Mon Tue Wed Thu Fri Sat
in-FRENCH  1 .dayList cr   \ display :  D L M M J V S
```

In the second line, we only display the first letter of each day of the week.

In this example, we leverage **`defer`** to simplify programming. In web development, we would use *templates* to manage multilingual sites. In FORTH, we simply move a vector in a delayed action word. Here we only manage two languages. This mechanism can easily be extended to other languages, because we have separated the management of text messages from the purely application part.

Word Creation Words

FORTH is more than a programming language. It's a meta-language. A meta-language is a language used to describe, specify or manipulate other languages.
With eForth Linux, we can define the syntax and semantics of programming words beyond the formal framework of basic definitions.

We have already seen the words defined by `constant` , `variable` , `value` . These words are used to manage digital data.

In the Data Structures for eForth Linux chapter, we also used the word `create`. This word creates a header allowing access to a data area stored in memory. Example :

```
create temperatures
34 , 37 , 42 , 36 , 25 , 12 ,
```

Here, each value is stored in the parameters area of the word `temperatures` with the word `,`.

With eForth Linux, we will see how to customize the execution of words defined by `create`.

Using does>

However, there is a combination of "`CREATE`" and "`DOES>`" keywords, which are often used together to create custom words (vocabulary words) with specific behaviors.

Here's how it generally works in Forth:

- `CREATE` : this keyword is used to create a new data space in the eForth Linux dictionary. It takes one argument, which is the name you give your new word;
- `DOES>` : this keyword is used to define the behavior of the word you just created with `CREATE` . It is followed by a block of code that specifies what the word should do when encountered during program execution.

Together it looks something like this:

```
forth
CREATE my-new-word
\ code to execute when encountering my-new-word
    DOES>
;
```

When the word `my-new-word` is encountered in the FORTH program, the code specified in the `does>... ;` will be executed.

```
\ define a register, similar as constant
: defREG:
    create ( addr1 -- <name> )
```

```
        ,
    does>   ( -- regAddr )
        @
  ;
```

Here, we define the definition word `defREG:` which has exactly the same action as `constant` . But why create a word that recreates the action of a word that already exists?

```
$3FF44004 constant  DB2INSTANCE
```

or

```
$3FF44004 defREG:  DB2INSTANCE
```

are similar. However, by creating our registers with `defREG:` we have the following advantages:

- a more readable eForth Linux source code. We easily detect all the constants naming an ESP32 register;
- we leave ourselves the possibility of modifying the `does> part` of `defREG:` without then having to rewrite the lines of code which would not use `defREG:`

Here is a classic case, processing a data table:

```
\ definition word for one dimension arrays
:array (comp: -- <name> | exec: index <name> -- addr)
    create
    does>
        swap cell * +
  ;
array temperatures
    21 ,    32 ,    45 ,    44 ,    28 ,    12 ,
0 temperatures @ .   \ display 21
5 temperatures @ .   \ display 12
```

The execution of `temperatures` must be preceded by the position of the value to extract in this table. Here we only get the address containing the value to extract.

Color management example

In this first example, we define the word `color:` which will retrieve the color to select and store it in a variable:

```
0 value currentCOLOR

\ define word as COLOR constant
: color: ( n -- <name> )
    create
        ,
    does>
        @ to currentCOLOR
  ;
```

```
$00 color: setBLACK
$ff color: setWHITE
```

Running the word `setBLACK` or `setWHITE` greatly simplifies the eForth Linux code. Without this mechanism, one of these lines would have had to be repeated regularly :

```
$00 currentCOLOR !
```

Or

```
$00 constant BLACK
BLACK currentCOLOR !
```

Example, writing in pinyin

Pinyin is commonly used around the world to teach Mandarin Chinese pronunciation, and it is also used in various official contexts in China, such as street signs, dictionaries, and learning textbooks. It makes learning Chinese easier for people whose native language uses the Latin alphabet.

To write Chinese on a QWERTY keyboard, the Chinese generally use a system called "pinyin input". Pinyin is a system of romanization of Mandarin Chinese, which uses the Latin alphabet to represent the sounds of Mandarin.

On a QWERTY keyboard, users type Mandarin sounds using pinyin romanization. For example, if someone wants to write the character "你" ("nǐ" meaning "you" in English), they can type "ni".

In this very simplified code, you can program pinyin words to write in Mandarin. The following code only works in eForth Linux :

```
\ Work well in eForth Linux
internals
: chinese:
    create ( c1 c2 c3 -- )
        c, c, c,
    does>
        3 serial-type
  ;
forth
```

To find the UTF8 code of a Chinese character, copy the Chinese character, from Google Translate for example. Example :

```
Good Morning --> 早安 (Zao an)
```

Copy 早 and go to PuTTy terminal and type :

```
key key key \ followed by key <enter>
```

paste the character 早. eForth Linux should display the following codes:

```
230 151 169
```

For each Chinese character, we will use these three codes as follows:

```
169 151 230 chinese: Zao
137 174 229 chinese: Year
```

Use :

```
Zao An        \ display 早安
```

Admit that programming like this is something other than what you can do in C language. No?

Processing UTF8 characters

It was while carrying out some character entry tests on the keyboard that a small problem appeared. If we do this:

```
key      \ and press a key, push 97 on stack
```

So far everything is normal. But on the keyboard, we also have, in France on AZERTY keyboard, accented characters and certain characters like `€` . Let's try `key again` and try to recover the code for this character:

```
key
€
 ok
226 --> ��
ERROR: �� NOT FOUND!
```

The first code retrieved has the value 226. but there are two other codes which disrupt the FORTH interpreter. Let's see this solution:

```
key key key
€
 ok
226 130 172
```

Oh...?!?!? Three codes?

UTF8 encoding

Let's take the three codes `226 130 172` in hexadecimal: `E2 82 AC` . If we do this:

```
$e2 emit
```

It says `ok` . Mmmm.... Let's check in a loop that is in the range 32-255:

```
: dispChars   ( -- )
    256 32 do
        i emit
    loop
  ;
```

Running `dispChars` displays this:

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~��������
������� ����������������������������������
���������������������������������� �������������
ok
```

eForth Linux has some problems displaying characters with an ASCII code greater than 127. If we repeat this test with eForth Windows, the same word `dispChars` displays this:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~Çüéâäàåçêëè
```

```
îìÄÅÉæÆôöòûùÿÖÜø£Ø×ƒáíó úñÑªº¿®¬½¼¡«»░▒▓ │┤ÁÂÀ©╣║╗╝¢
¥┐ └┴┬├─┼ãÃ╚╔╩╦╠═╬¤ðÐÊËÈıÍÎÏ┘ ┌█▄¦Ì▀ÓßÔÒõ ÕµþÞÚÛÙýÝ¯´ ±‗¾¶§÷¸°¨·¹³²■ ok
```

For characters whose ASCII code is in the range [32..127], the characters are identical. For characters with an ASCII code greater than 127 (7F in hexadecimal), eForth Linux cannot display valid characters.

To display the `€ character` in eForth Linux, we have simple solutions:

```
: .eur ( -- )
    ." €" ;
```

Or

```
: .€ ( -- )
    ." €" ;
```

Or

```
: .eur ( -- )
    226 emit  130 emit  172 emit ;
```

But we do not resolve the problem of characters whose ASCII code is greater than 127. To resolve this problem, we must look at UTF8 encoding, the one used by eForth Linux.

In UTF8 encoding, ASCII characters are encoded on 7 bits:

- `0 bbb-bbbb` 1 byte encoding

For all other characters, the coding is 2 or 3 or 4 bytes:

- `11 0 b-bbbb 10 bb-bbbb` 2-byte encoding
- `111 0- bbbb 10 bb-bbbb 10 bb-bbbb` encoding on 3 bytes
- `1111 -0 bbb 10 bb-bbbb 10 bb-bbbb 10 bb-bbbb` 4-byte encoding

For all codes greater than $7F, the first most significant bits determine the number of bytes encoding a UTF8 character. Let's return to our character **€** . The first code that comes up when executing **`key`** is $E2. In binary: **`111 00010`** . Here we have three bits at 1. This means that the **`€ character`** is coded on 3 bytes.

Let's test with the UTF8 character 橙 . An execution of **`key`** brings up code 240, in binary: **`1111 0000`** . We have 4 bits at 1. The character 橙 is coded over four bytes.

Retrieve the UTF8 character code entered using the keyboard

`key` executions to be executed based on the character entered on the keyboard:

- we execute a first **`key`**
- if the code is greater than 127, we slide this code 1 bit to the left, then we test bit b7. If this bit is 1 we re-execute **`key`** .

Here is the code capable of entering any UTF8 character:

```
0 value keyUTF8

: toKeyUTF8  ( c -- )
    keyUTF8 8 lshift or to keyUTF8
  ;
```

The word **toKeyUTF8** receives an 8-bit keyboard code and concatenates it with the contents of the **keyUTF8 value** . The idea is to recover the UTF8 encoding into a single final numeric value.

```
\ execute key recursively
: getKeys ( n -- )
    1 lshift dup $80 and    \ test if bit b7 is not null
    if   recurse            \ re-execute xkey
    else drop  then         \ otherwise, drop n
    key toKeyUTF8           \ and execute key 1 or may times
  ;
```

The word **getkeys** processes the code returned by the first execution of **key** . It performs a one-byte shift to the left and tests bit b7 (sequence **1 lshift dup $80 and**). If this bit is 1, the word is re-executed (**if recurse sequence**).

Recursion allows you to control the number of iterations of **getKeys** without requiring complex loops and tests. The recursion stops as soon as a bit b7 is 0. The recursion exit takes place after **then** . The word **getKeys** will execute the sequence **key toKeyUTF8** as many times as there are recursive calls.

```
\ key version for UTF8 characters
: ukey
    key to keyUTF8
    keyUTF8 $7F > if                \ if 1st key code > $7F
        keyUTF8 1 lshift getKeys    \ execute xkey
    then
    keyUTF8
  ;
```

The word **ukey** can now replace the word **key** to retrieve the UTF8 code of any character in the UTF8 character set:

```
hex
ukey  .    \ paste €   and <enter>, display : E282AC
```

This is confirmed by the online UTF8 documentation.

Displaying UTF8 characters from their code

If we look at the definition of the word **emit** , we find this:

```
: emit
    >R RP@ 1 type rdrop
```

```
  ;
```

`RP@ 1 type` code sequence strictly limits the display of a single-byte code character. This **`hex E282AC emit`** sequence will not work. Likewise :

```
: uemit
    >R RP@ 4 type rdrop
  ;
\  hex e282ac uemit   display : ��� ok
```

The problem comes from the order of the bytes of a digital value. A memory dump of the stack gives this:

```
--addr--- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ------chars-----
0802-00FA 00 00 00 00 00 00 AC 82 E2 00 00 00 00 00 08 01 ......���......
```

We must therefore *flip* the bytes like a sock:

```
\ reverse integer bytes, example:
\  hex 1a2b3C --> 3c2b1a
: reverse-bytes  ( n0 -- )
    0 { result }
    3 for
        result 100 * to result
        100 u/mod swap +to result
    next
    drop
    result
  ;
```

we can now rewrite our word `uemit` :

```
\ emit UTF8 encoded character
: uemit ( n -- )
    reverse-bytes
    >r rp@ 4 type
    rdrop
  ;
```

Running `hex E282AC uemit` displays: **€** .

In conclusion, with `ukey` and `uemit` , we now have words allowing us to process non-ASCII characters. So, with a Greek keyboard:

```
hex
ukey      \ press key Σ display : CEA3
uemit     \ display Σ
```

Encoding from UTF8 character code point

Each abstract character is associated with a unique number. This number is called a code point. The code point is a number between 1 and 17×2^{16} , or potentially 1,114,112 characters. A code point is denoted U+ followed by the hexadecimal value of the code

point.
Example: **U+00E9** for the character `e` .

The problem, as you have already understood, is that you cannot do `hex e9 emit` with eForth Linux.

To have the correct UTF8 encoding sequence b1-b0 (for byte1 byte0), you must switch the two most significant bits of byte e9 to b1. We cut e9 like this:

```
hex
e9 40 /mod
```

Which leaves us on the stack of data `r` and `q` resulting from the execution of `/mod` , or in our example the values `29` and `3` . To transform this into a two-byte value, we then execute:

```
100 * +
```

Which now leaves us on the data stack with the hexadecimal value `329` .

Now let's return to the double-byte UTF8 encoding format:

- `110` ***b-bbbb*** `10` ***bb-bbbb***

Here, in yellow, we have a masking value, `1100000010000000` in binary, `c080` in hexadecimal. It is this mask value `c080` that we will apply to the result of our previous calculation. Here is the complete coding sequence from code point `e9` :

```
e9 40 /mod
100 * +
c080 or
```

Which leaves us with the final `c3a9 code` , which is now usable with `uemit` :

```
c3a9 uemit   \ display char : é
```

We will now automate this....

Re-encoding by recursion

In the sequence `n 40 /mod` , we recover at each iteration a remainder and a quotient. When an iteration gives a zero quotient, we stop. This lends itself wonderfully to processing by recursion:

```
$40 constant BYTE_DIVISOR

\ split n modulo BYTE_DIVISOR
: mod40Recombine ( n -- )
    BYTE_DIVISOR /mod
    dup 0 > if
        recurse
    then
    $100 * +
  ;
```

The word `mod40Recombine` splits the value n into `rq pairs` . If `q` is equal to 0, we exit the recursion after `then` and we execute `$100 * +` as many times as was cut.

It remains to apply a mask based on the size of the re-encoded code point. Here are the limit values for two, three, or four byte encodings:

```
$8000      constant LIMIT_2_BYTES
$10000     constant LIMIT_3_BYTES
$200000    constant LIMIT_4_bytes
```

For each of these limit values, here are the masks to apply:

```
$C080        constant MASK_2_BYTES
$E08080      constant MASK_3_BYTES
$F0808080    constant MASK_4_BYTES
```

And finally, here is the word `bytesToUTF8` which applies the madque adapted to the size of the code point number:

```
: bytesToUTF8 ( n -- n' )
    >r
    r@ LIMIT_2_BYTES < if
        r> mod40Recombine
        MASK_2_BYTES OR
        exit
    then
    r@ LIMIT_3_BYTES < if
        r> mod40Recombine
        MASK_3_BYTES OR
        exit
    then
    r@ LIMIT_4_BYTES < if
        r> mod40Recombine
        MASK_4_BYTES OR
        exit
    then
    abort" UTF8 conversion failed"
  ;
```

There are certainly ways to make it more elegant. This definition has the merit of working. As input, we stack the code point to be re-encoded. As output, we obtain the UTF8 code usable by `uemit` .

Generate a UTF8 character table

The idea is to take the first and last code point numbers from a character table. These values are processed in a loop to generate a character table.

Let's start with a few useful words:

```
8 constant LINE_LIMIT
```

```
\ CR only if i MOD = 0
: cr? ( i -- )
    1+ LINE_LIMIT mod
    0= if
        cr
    then
  ;

\ display hex value format NNNN
: .###### ( n -- )
    <# # # # # # # #> type
  ;
```

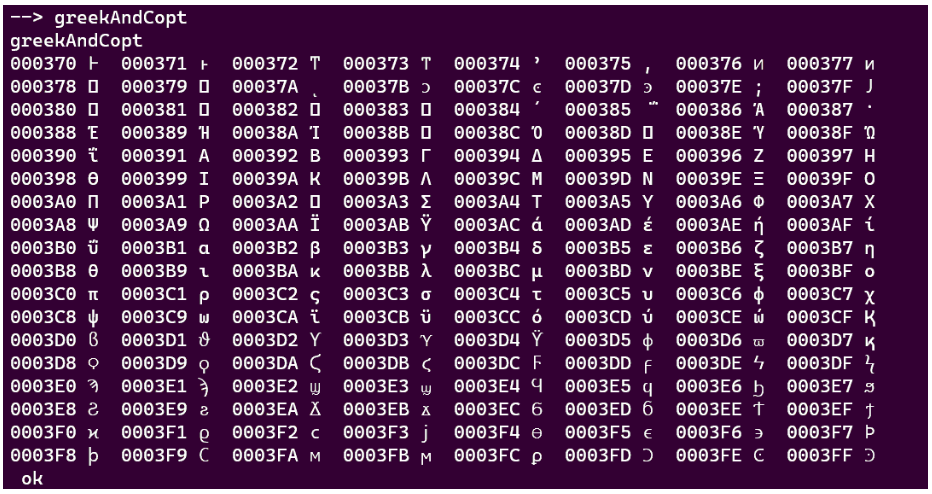
The word `cr?` executes a newline if the display reaches n columns. The word `.######` displays a 6-digit n value. Finally here is the display loop:

```
: utf8Set { start stop -- }
    base @ { currentBase }
    hex
    stop 1+ start do
        i .######
        space
        i bytesToUTF8 uemit
        2 spaces
        i cr?
    loop
    currentBase base !
  ;
```

Here is the definition to display the UTF8 character table of the Greek and Coptic character set:

```
: greekAndCopt ( -- )
    $370 $3ff utf8Set
  ;
```

Its execution displays this:



```
--> greekAndCopt
greekAndCopt
000370 Ͱ  000371 ͱ  000372 Ͳ  000373 ͳ  000374 ʹ  000375 ͵  000376 Ͷ  000377 ͷ
000378 □  000379 □  00037A ͺ  00037B ͻ  00037C ͼ  00037D ͽ  00037E ;  00037F Ϳ
000380 □  000381 □  000382 □  000383 □  000384 ΄  000385 ΅  000386 Ά  000387 ·
000388 Έ  000389 Ή  00038A Ί  00038B □  00038C Ό  00038D □  00038E Ύ  00038F Ώ
000390 ΐ  000391 Α  000392 Β  000393 Γ  000394 Δ  000395 Ε  000396 Ζ  000397 Η
000398 Θ  000399 Ι  00039A Κ  00039B Λ  00039C Μ  00039D Ν  00039E Ξ  00039F Ο
0003A0 Π  0003A1 Ρ  0003A2 □  0003A3 Σ  0003A4 Τ  0003A5 Υ  0003A6 Φ  0003A7 Χ
0003A8 Ψ  0003A9 Ω  0003AA Ϊ  0003AB Ϋ  0003AC ά  0003AD έ  0003AE ή  0003AF ί
0003B0 ΰ  0003B1 α  0003B2 β  0003B3 γ  0003B4 δ  0003B5 ε  0003B6 ζ  0003B7 η
0003B8 θ  0003B9 ι  0003BA κ  0003BB λ  0003BC μ  0003BD ν  0003BE ξ  0003BF ο
0003C0 π  0003C1 ρ  0003C2 ς  0003C3 σ  0003C4 τ  0003C5 υ  0003C6 φ  0003C7 χ
0003C8 ψ  0003C9 ω  0003CA ϊ  0003CB ϋ  0003CC ό  0003CD ύ  0003CE ώ  0003CF Ϗ
0003D0 ϐ  0003D1 ϑ  0003D2 ϒ  0003D3 ϓ  0003D4 ϔ  0003D5 ϕ  0003D6 ϖ  0003D7 ϗ
0003D8 Ϙ  0003D9 ϙ  0003DA Ϛ  0003DB ϛ  0003DC Ϝ  0003DD ϝ  0003DE Ϟ  0003DF ϟ
0003E0 Ϡ  0003E1 ϡ  0003E2 Ϣ  0003E3 ϣ  0003E4 Ϥ  0003E5 ϥ  0003E6 Ϧ  0003E7 ϧ
0003E8 Ϩ  0003E9 ϩ  0003EA Ϫ  0003EB ϫ  0003EC Ϭ  0003ED ϭ  0003EE Ϯ  0003EF ϯ
0003F0 ϰ  0003F1 ϱ  0003F2 ϲ  0003F3 ϳ  0003F4 ϴ  0003F5 ϵ  0003F6 ϶  0003F7 Ϸ
0003F8 ϸ  0003F9 Ϲ  0003FA Ϻ  0003FB ϻ  0003FC ϼ  0003FD Ͻ  0003FE Ͼ  0003FF Ͽ
 ok
```

Figure 15: table of Greek and Coptic characters

The numbers indicate the code point of each character. Here, we see for example that the character φ has the code point 3D5.

In conclusion, what is it for?

First, it helps to understand UTF8 encoding.

Then, we can take inspiration from part of this code to count the number of characters. Example :

```
s" nb: φ"
\ display :
134495848 6
```

At a glance, we have 5 characters, whereas eForth indicates a string length of 6 characters!

In an alphanumeric sorting procedure, it may be necessary to transform certain accented characters into their non-accented equivalent: à → a, é → e, etc.
I leave you complete freedom to find a practical application.

Detailed content of eForth Linux vocabularies

Eforth Linux provides numerous vocabularies:

- `FORTH` is the main vocabulary;
- certain vocabularies are used for internal mechanics for eForth Linux, such as `internals` , `asm`…
- many vocabularies allow the management of specific ports or accessories, such as `bluetooth` , `oled` , `spi` , `wifi` , `wire`…

Here you will find the list of all the words defined in these different vocabularies. Some words are presented with a colored link:

align is an ordinary FORTH word;

CONSTANT is definition word;

begin marks a control structure;

key is a deferred execution word;

LED is a word defined by `constant` , `variable` or `value` ;

registers marks a vocabulary.

`FORTH` vocabulary words are displayed in alphabetical order. For other vocabularies, the words are presented in their display order.

Version v 7.0.7.15

FORTH

-	-rot	,	;	:	:noname	!
?	?do	?dup	.	."	.s	'
(local)	[	[']	[char]	[ELSE]	[IF]	[THEN]
]	{	}transfer	@	*	*/	*/MOD
/	/mod	#	#!	#>	#fs	#s
#tib	+	+!	+loop	+to	<	<#
<=	<>	=	>	>=	>BODY	>flags
>flags&	>in	>link	>link&	>name	>params	>R
>size	0<	0<>	0=	1-	1/F	1+
2!	2@	2*	2/	2drop	2dup	4*
4/	abort	abort"	abs	accept	afliteral	aft
again	ahead	align	aligned	allocate	allot	also
AND	ansi	argc	argv	ARSHIFT	asm	assert
at-xy	base	begin	bg	BIN	binary	bl
blank	block	block-fid	block-id	buffer	bye	c,
C!	C@	CASE	cat	catch	cd	CELL

cell/	cell+	cells	char	CLOSE-DIR	CLOSE-FILE	cmove
cmove>	CONSTANT	context	copy	cp	cr	CREATE
CREATE-FILE	current	decimal	default-key	default-type		default-use
defer	DEFINED?	definitions	DELETE-FILE	depth	DLSYM	do
DOES>	DROP	dump	dump-file	DUP	echo	editor
else	emit	empty-buffers		ENDCASE	ENDOF	erase
evaluate	EXECUTE	EXIT	extract	F-	f.	f.s
F*	F**	F/	F+	F<	F<=	F<>
F=	F>	F>=	F>S	F0<	F0=	FABS
FATAN2	fconstant	FCOS	fdepth	FDROP	FDUP	FEXP
fg	file-exists?		FILE-POSITION		FILE-SIZE	fill
FIND	fliteral	FLN	FLOOR	flush	FLUSH-FILE	FMAX
FMIN	FNEGATE	FNIP	for	forget	form	FORTH
forth-builtins		FOVER	FP!	FP@	fp0	free
FROT	FSIN	FSINCOS	FSQRT	FSWAP	fvariable	graphics
handler	here	hex	hld	hold	httpd	I
if	IMMEDIATE	include	included	included?	internals	invert
is	J	K	key	key?	L!	latestxt
leave	list	literal	load	loop	ls	LSHIFT
max	min	mkdir	mod	ms	ms-ticks	mv
n.	needs	negate	nest-depth	next	nip	nl
NON-BLOCK	normal	octal	OF	ok	only	open-blocks
OPEN-DIR	OPEN-FILE	OR	order	OVER	pad	page
PARSE	pause	PI	posix	postpone	precision	previous
prompt	pwd	quit	r"	R@	R/O	R/W
R>	r|	r~	rdrop	READ-DIR	READ-FILE	recurse
refill	remaining	remember	RENAME-FILE	repeat	REPOSITION-FILE	
required	reset	resize	RESIZE-FILE	restore	revive	rm
rmdir	rot	RP!	RP@	rp0	RSHIFT	s"
S>F	s>z	save	save-buffers		scr	sealed
see	set-precision		set-title	sf,	SF!	SF@
SFLOAT	SFLOAT+	SFLOATS	sign	SL@	sockets	SP!
SP@	sp0	space	spaces	start-task	startswith?	startup:
state	str	str=	streams	structures	SW@	SWAP
task	tasks	telnetd	terminate	termios	then	throw
thru	tib	to	touch	transfer	transfer{	type
u.	U/MOD	UL@	UNLOOP	until	update	use
used	UW@	value	VARIABLE	visual	vlist	vocabulary
W!	W/O	web-interface		while	words	WRITE-FILE
x11	XOR	z"	z>s			

ansi

```
terminal-restore terminal-save show hide scroll-up scroll-down clear-to-eol
bel esc
```

asm

```
end-code code, code4, code3, code2, code1, callot chere reserve code-at
code-start
```

editor

```
a r d e wipe p n l
```

graphics

```
poll wait flip window window heart vertical-flip viewport scale translate
}g g{ screen>g box color pressed? pixel height width event last-char last-key
mouse-y mouse-x RIGHT-BUTTON MIDDLE-BUTTON LEFT-BUTTON FINISHED TYPED RELEASED
PRESSED MOTION EXPOSED RESIZED IDLE internals
```

graphics/internals

```
update-event pending-key? update-key update-mouse image-resize EVENT-MASK
keybuffer-used keybuffer keybuffer-size xevent xevent-type image gc window-handle
root-window white black screen-depth xvisual colormap screen display raw-heart
heart-ratio heart-initialize cmax! cmin! heart-end heart-start heart-size
heart-steps heart-f raw-box g> >g gp gstack hline ty tx sy sx key-state!
key-state key-count backbuffer
```

httpd

```
notfound-response bad-response ok-response response send path method hasHeader
handleClient read-headers completed? body content-length header crnl= eat
skipover skipto in@<> end< goal# goal strcase= upper server client-cr client-emit
client-read client-type client-len client httpd-port clientfd sockfd body-read
body-1st-read body-chunk body-chunk-size chunk-filled chunk chunk-size
max-connections
```

internals

```
errno CALLCODE CALL0 CALL1 CALL2 CALL3 CALL4 CALL5 CALL6 CALL7 CALL8 CALL9
CALL10 CALL11 CALL12 CALL13 CALL14 CALL15 DOFLIT S>FLOAT? fill32 'heap
'context 'latestxt 'notfound 'heap-start 'heap-size 'stack-cells 'boot
'boot-size 'tib 'argc 'argv 'runner 'throw-handler NOP BRANCH 0BRANCH DONEXT
DOLIT DOSET DOCOL DOCON DOVAR DOCREATE DODOES ALITERAL LONG-SIZE S>NUMBER?
'SYS YIELD EVALUATE1 'builtins internals-builtins autoexec boot-set-title
e' @line grow-blocks use?! common-default-use block-data block-dirty clobber
clobber-line include+ path-join included-files raw-included include-file
sourcedirname sourcefilename! sourcefilename sourcefilename# sourcefilename&
starts../ starts./ dirname ends/ default-remember-filename remember-filename
restore-name save-name forth-wordlist setup-saving-base 'cold park-forth
park-heap saving-base crtype cremit cases (+to) (to) --? }? ?room scope-create
do-local scope-clear scope-exit local-op scope-depth local+! local! local@
<>locals locals-here locals-area locals-gap locals-capacity ?ins. ins.
vins. onlines line-pos line-width size-all size-vocabulary vocs. voc. voclist
voclist-from see-all >vocnext see-vocabulary nonvoc? see-xt ?see-flags
see-loop see-one indent+! icr see. indent mem= ARGS_MARK -TAB +TAB NONAMED
BUILTIN_FORK SMUDGE IMMEDIATE_MARK dump-line ca@ cell-shift cell-base cell-mask
#f+s internalized BUILTIN_MARK zplace $place free. boot-prompt raw-ok [SKIP]'
[SKIP] ?stack sp-limit input-limit tib-setup raw.s $@ digit parse-quote
leaving, leaving )leaving leaving( value-bind evaluate&fill evaluate-buffer
```

```
arrow ?arrow. ?echo input-buffer immediate? eat-till-cr wascr *emit *key
notfound last-vocabulary voc-stack-end xt-transfer xt-hide xt-find& scope
```

posix

```
FNDELAY F_SETFL fcntl CLOCK_BOOTTIME_ALARM CLOCK_REALTIME_ALARM CLOCK_BOOTTIME
CLOCK_MONOTONIC_COARSE CLOCK_REALTIME_COARSE CLOCK_MONOTONIC_RAW
CLOCK_THREAD_CPUTIME_ID
CLOCK_PROCESS_CPUTIME_ID CLOCK_MONOTONIC CLOCK_REALTIME timespec clock_gettime
0777 SIGPIPE SIGBUS SIGKILL SIGINT SIGHUP SIG_IGN SIG_DFL EPIPE EAGAIN
d0=ior d0<ior 0=ior 0<ior stdin-key stdout-write O_NONBLOCK O_APPEND O_TRUNC
O_CREAT O_RDWR O_WRONLY O_RDONLY MAP_ANONYMOUS MAP_FIXED MAP_PRIVATE PROT_EXEC
PROT_WRITE PROT_READ PROT_NONE SEEK_END SEEK_CUR SEEK_SET stderr stdout
stdin errno .d_name .d_type readdir closedir opendir getwd rmdir mkdir
chdir signal usleep realloc sysfree malloc rename unlink mprotect munmap
mmap waitpid wait fork sysexit fsync ftruncate lseek write read close creat
open sign-extend shared-library sysfunc sofunc calls dlopen 'dlopen RTLD_NOW
RTLD_LAZY
```

sockets

```
sockaccept ip. ip# ->h_addr ->addr! ->addr@ ->port! ->port@ sockaddr l,
s, bs, SO_REUSEADDR SOL_SOCKET sizeof(sockaddr_in) AF_INET SOCK_RAW SOCK_DGRAM
SOCK_STREAM gethostbyname recvmsg recvfrom recv sendmsg sendto send setsockopt
poll sockaccept connect listen bind socket
```

tasks

```
main-task .tasks task-list
```

telnetd

```
server broker-connection wait-for-connection connection telnet-key
telnet-type telnet-emit broker client-len client telnet-port clientfd
sockfd
```

termios

```
termios-key termios-key? pending form winsize sizeof(winsize) TIOCGWINSZ
normal-mode raw-mode termios! termios@ VMIN VTIME TCSAFLUSH _ECHO ICANON
.c_cc[] .c_lflag new-termios old-termios sizeof(termios) delay-mode nodelay-mode
ioctl tcsetattr tcgetattr
```

web-interface

```
server webserver-task do-serve handle1 serve-key serve-type handle-input
handle-index out-string output-stream input-stream out-size webserver index-html
index-html#
```

x11

```
GenericEvent MappingNotify ClientMessage ColormapNotify SelectionNotify
SelectionRequest SelectionClear PropertyNotify CirculateRequest CirculateNotify
ResizeRequest GravityNotify ConfigureRequest ConfigureNotify ReparentNotify
MapRequest MapNotify UnmapNotify DestroyNotify CreateNotify VisibilityNotify
NoExpose GraphicsExpose Expose KeymapNotify FocusOut FocusIn LeaveNotify
EnterNotify MotionNotify ButtonRelease ButtonPress KeyRelease KeyPress
xevent# OwnerGrabButtonMask ColormapChangeMask PropertyChangeMask FocusChangeMask
SubstructureRedirectMask SubstructureNotifyMask ResizeRedirectMask
StructureNotifyMask
VisibilityChangeMask ExposureMask KeymapStateMask ButtonMotionMask
Button5MotionMask
Button4MotionMask Button3MotionMask Button2MotionMask Button1MotionMask
PointerMotionHintMask PointerMotionMask LeaveWindowMask EnterWindowMask
ButtonReleaseMask ButtonPressMask KeyReleaseMask KeyPressMask xmask NoEventMask
xexposure xconfigure xmotion xkey xbutton xany bool time win xevent-size
NULL ZPixmap XYPixmap XYBitmap XFillRectangle XSetBackground XSetForeground
XDrawString XSelectInput XPutImage XNextEvent XMapWindow XLookupString
XFlush XDestroyImage XDefaultVisual XDefaultDepth XCreateSimpleWindow XCreateImage
XCreateGC XCheckMaskEvent XRootWindow XDefaultScreen XDefaultColormap
XScreenOfDisplay
XDisplayOfScreen XWhitePixel XBlackPixel XOpenDisplay xlib
```

Lexical index