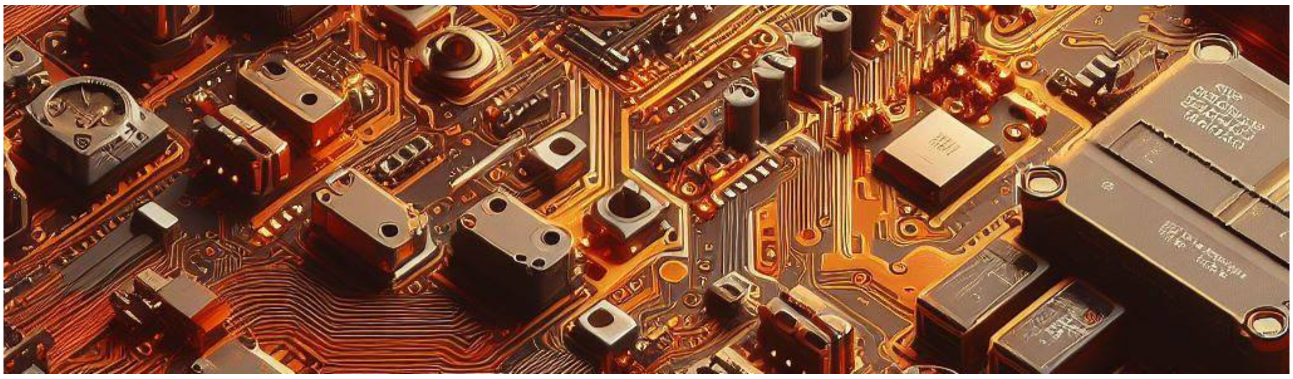
The great book for eFORTH Linux

version 1.0 - 21 novembre 2023



Author

- Marc PETREMANN

Collaborateur(s)

- xxx

Contents

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 :

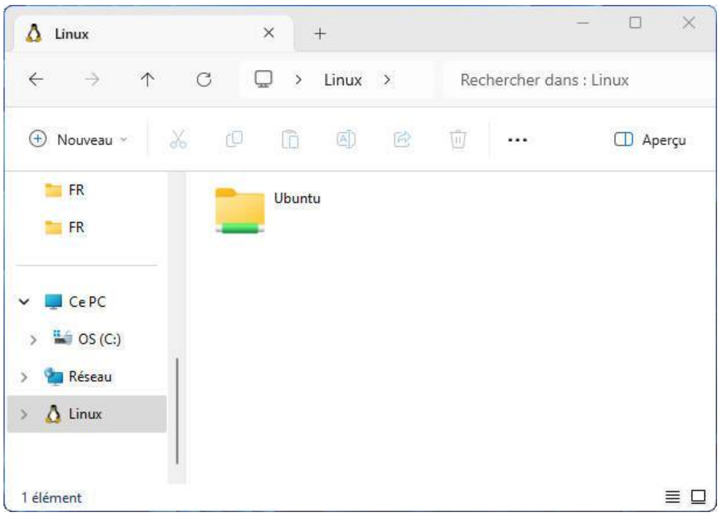


Figure 1: Ubuntu accessible in one click from WSL under Windows

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 creating a **ueforth folder** :

```
mkdir ueforth
```

Then select this folder:

```
cd ueforth
```

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` :

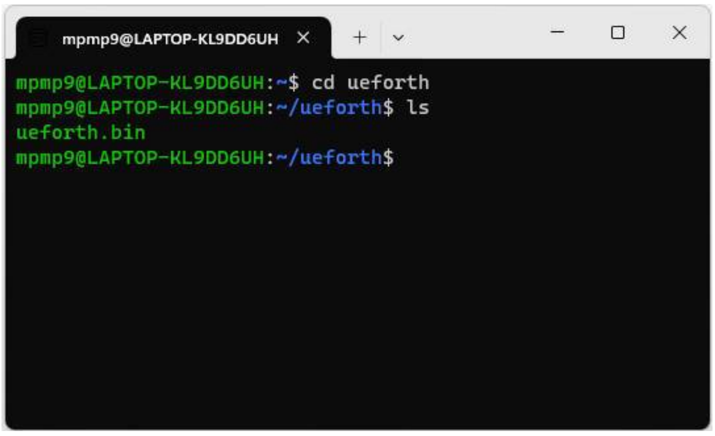
```
wget https://eforth.storage.googleapis.com/releases/ueforth-7.0.7.15.linux
```

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

`mv` command : ueforth-7.0.7.15.linux

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

We check that everything went well with a simple `ls command` :



```
mpmp9@LAPTOP-KL9DD6UH:~$ cd ueforth
mpmp9@LAPTOP-KL9DD6UH:~/ueforth$ ls
ueforth.bin
mpmp9@LAPTOP-KL9DD6UH:~/ueforth$
```

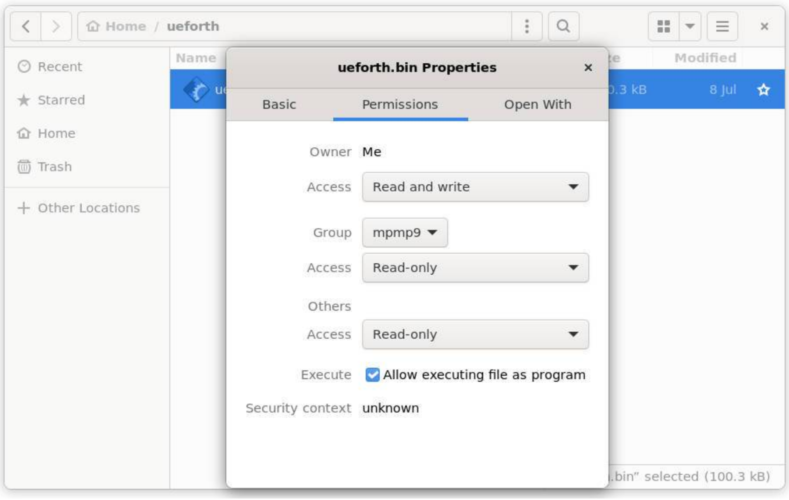
Figure 2: the ueforth.bin file is located in our ueforth folder

We still have one last manipulation to perform, making this file executable by the Linux system. If you have **Nautilus** File Explorer installed, which you usually do, type:

```
Nautilus
```

In Nautilus, select the ueforth.bin file, then:

- right click → select *Properties*
- in the pop-up, select *Permissions*
- *Permissions* tab , check *Execute*



Owner Me
Access Read and write
Group mpmp9
Access Read-only
Others
Access Read-only
Execute ☑ Allow executing file as program
Security context unknown

Figure 3: Execute box checked makes the file executable as a program under Linux

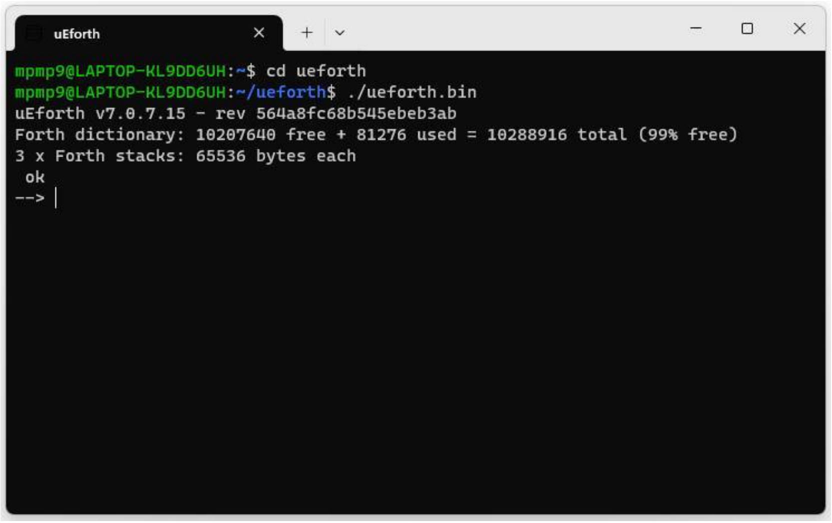
And it's done !

Launch eForth Linux

To launch **eForth when Linux** boots :

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

eForth Linux starts immediately:



```
mpmp9@LAPTOP-KL9DD6UH:~$ cd ueforth
mpmp9@LAPTOP-KL9DD6UH:~/ueforth$ ./ueforth.bin
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 4: 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....

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
```

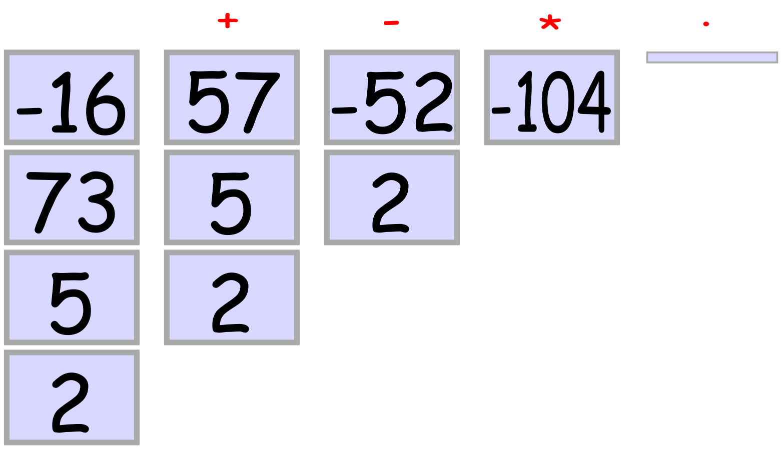
leaves the parameter stack as it is

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
```

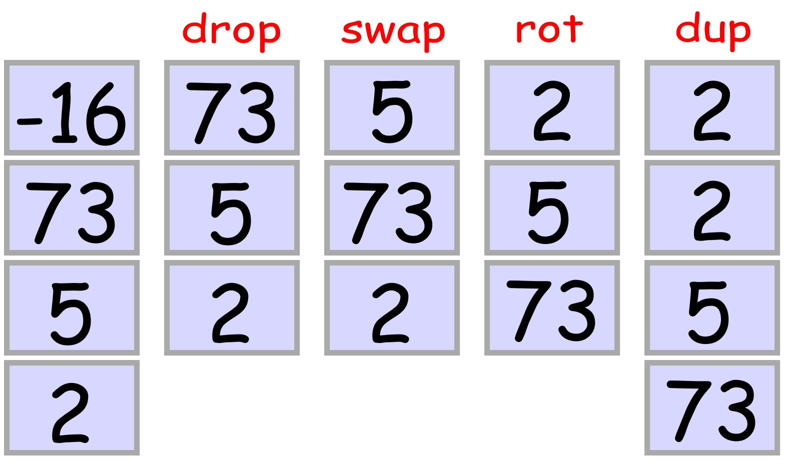
Note that eForth Linux conveniently displays the stack elements when interpreting each line and that the value of `-16` is displayed as a 64-bit unsigned integer. Furthermore, the word `.` consumes data value `-104`, leaving the stack empty. If we execute `.` on the now empty stack, the external interpreter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

Lexical index