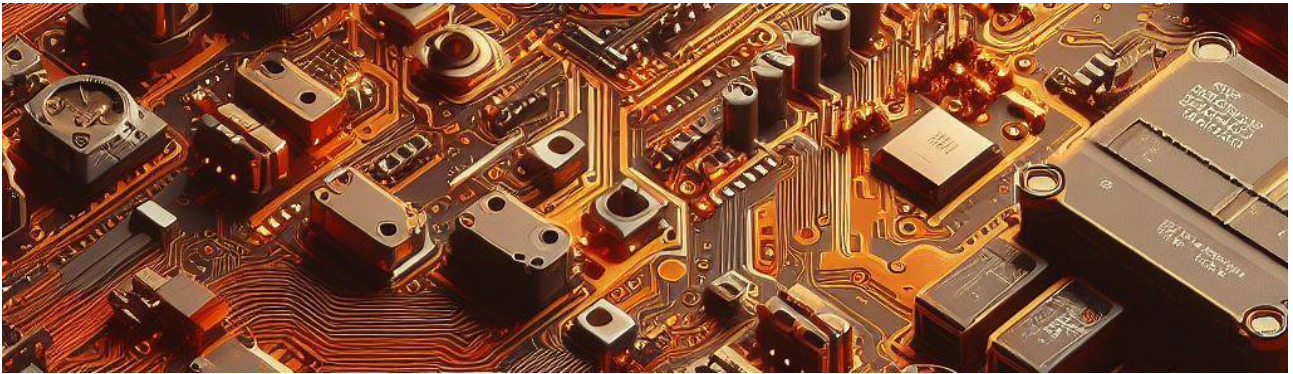


The great book for eFORTH Linux

version 1.1 - 28 novembre 2023



Author

- Marc PETREMANN

Contents

Install eForth on Linux.....	4
Prerequisites.....	4
Install eForth Linux on Linux.....	5
Launch eForth Linux.....	5
A real 64-bit FORTH with eForth Linux.....	7
Values on the data stack.....	7
Values in memory.....	7
Word processing depending on data size or type.....	8
Conclusion.....	9
Editing and managing source files for eForth Linux.....	11
Text file editors.....	11
Storage on GitHub.....	11
Edit files for eForth Linux from Windows.....	12
Creation and management of FORTH projects with Netbeans.....	13
Create an eForth project with Netbeans.....	13
Some good practices.....	15
Executing the contents of a file by eForth Linux.....	16
The Linux file system.....	18
Handling files.....	18
Organize and compile your files with eForth Linux.....	19
Organize your files.....	19
Chaining of files.....	20
Conclusion.....	21
Comments and debugging.....	22
Write readable FORTH code.....	22
Source code indentation.....	23
Comments.....	24
Stack comments.....	24
Meaning of stack parameters in comments.....	25
Word Definition Word Comments.....	25
Textual comments.....	26
Comment at the beginning of the source code.....	26
Diagnostic and tuning tools.....	27
The decompiler.....	27
Memory dump.....	27
Data stack monitor.....	28
Dictionary / Stack / Variables / Constants.....	29
Expand Dictionary.....	29
Dictionary management.....	29
Stacks and reverse Polish notation.....	30
Handling the parameter stack.....	31
The Return Stack and Its Uses.....	31

Memory usage.....	32
Variables.....	32
Constants.....	32
Pseudo-constant values.....	33
Basic tools for memory allocation.....	33
Displaying numbers and character strings.....	34
Change of numerical base.....	34
Definition of new display formats.....	35
Displaying characters and character strings.....	37
String variables.....	39
Text variable management word code.....	39
Adding character to an alphanumeric variable.....	41
Word Creation Words.....	43
Using does>.....	43
Color management example.....	44
Example, writing in pinyin.....	45

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 WSL¹ 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 :

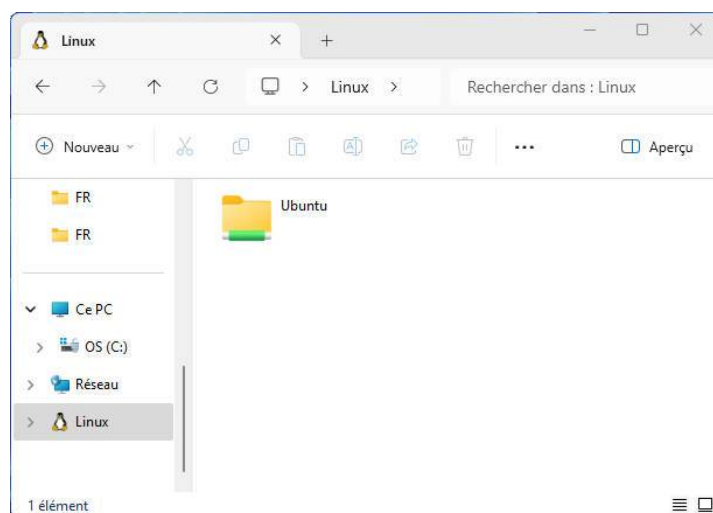


Figure 1: Ubuntu accessible in one click from WSL under Windows

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
```

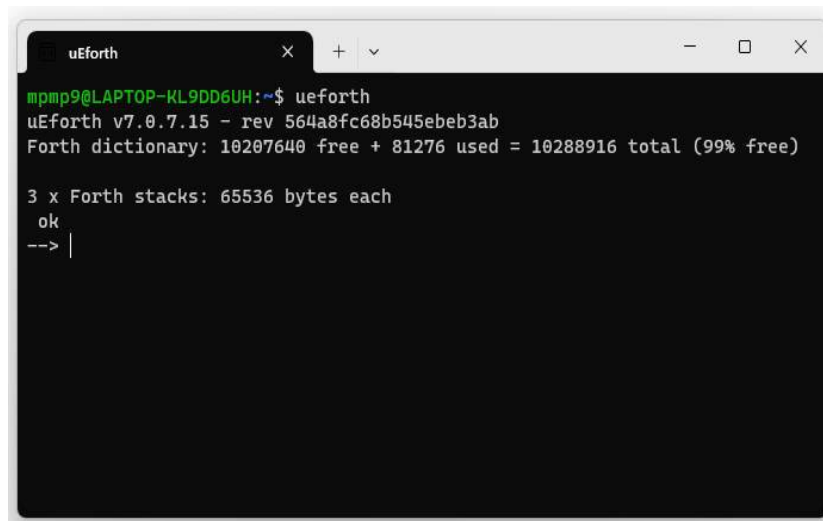
And it's done ! eForth Linux can now be used from any Linux directory.

Launch eForth Linux

To launch **eForth when Linux** boots :

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

eForth Linux starts immediately:

A terminal window titled 'uEforth' with standard window controls. The prompt is 'mpmp9@LAPTOP-KL9DD6UH:~\$'. The command 'ueforth' has been executed, resulting in the following output: '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', and a '-->' prompt with a cursor.

```
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

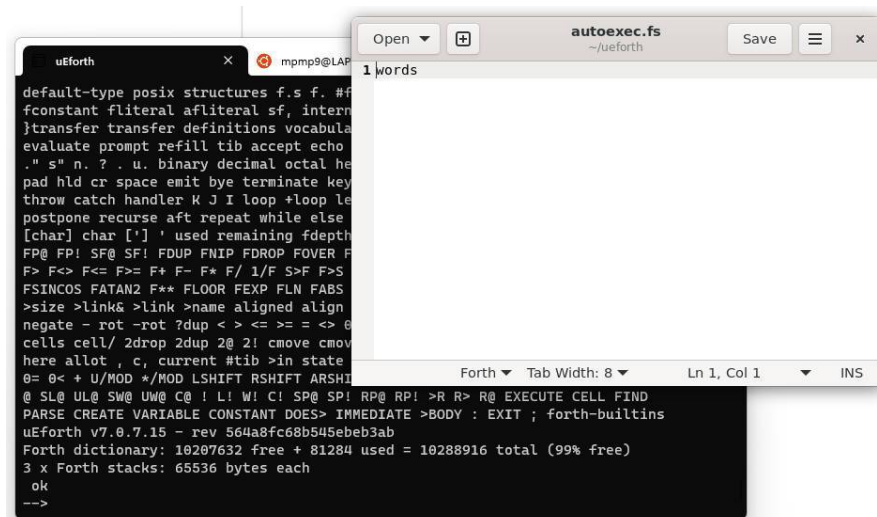


Figure 3: editing `autoexec.fs` file with `gedit` on Linux

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

On GitHub, you can share a

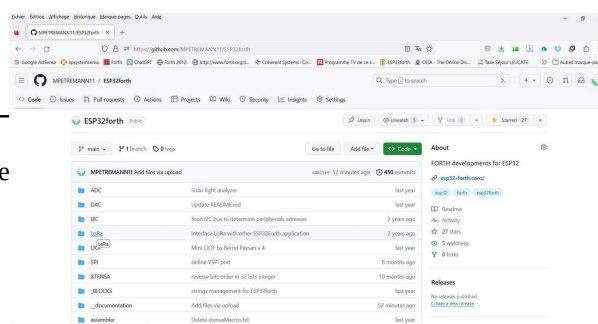


Figure 4: files storage on Github

2 <https://github.com/>

3 <https://sourceforge.net>

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.

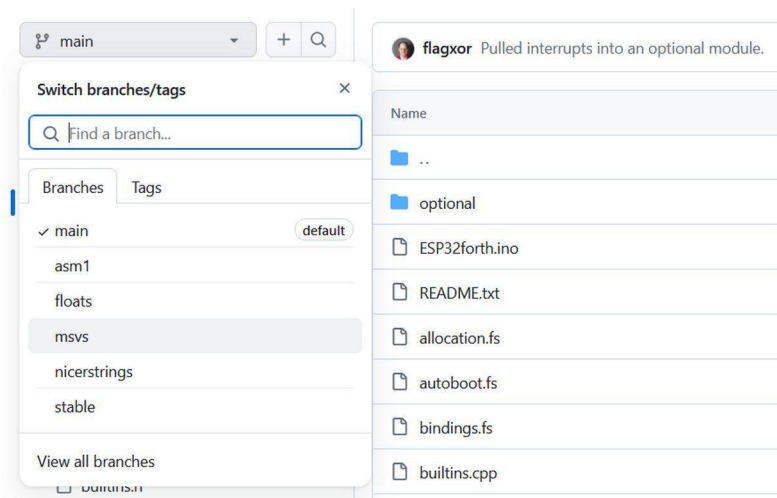


Figure 5: access to a branch in a project

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* ;

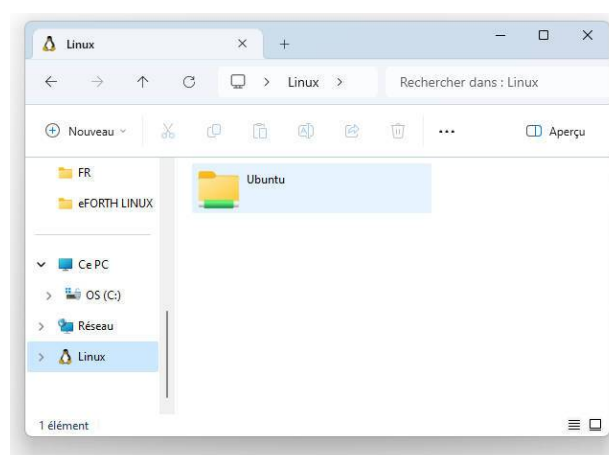


Figure 6: accessing Linux files from Windows

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

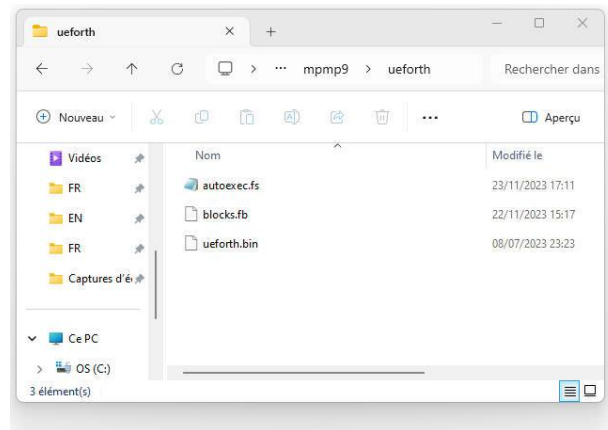


Figure 7: Linux files visible from Windows

- 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*

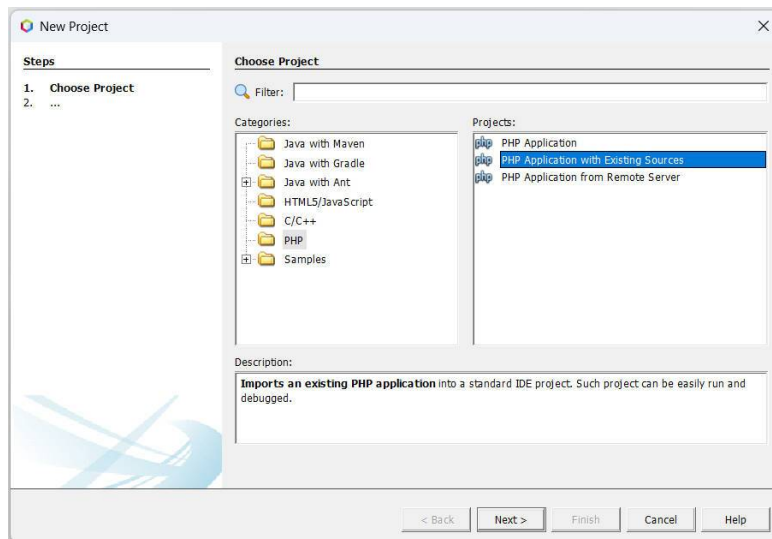


Figure 8: création projet PHP

- 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:

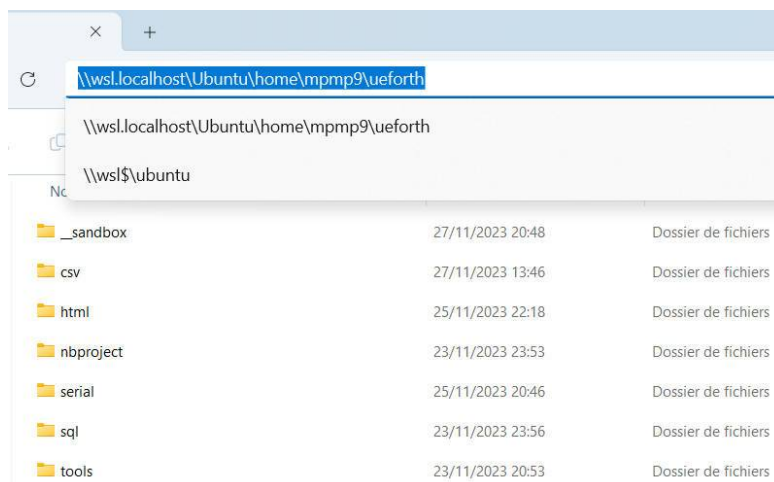


Figure 9: copy path to ueforth on Linux

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:

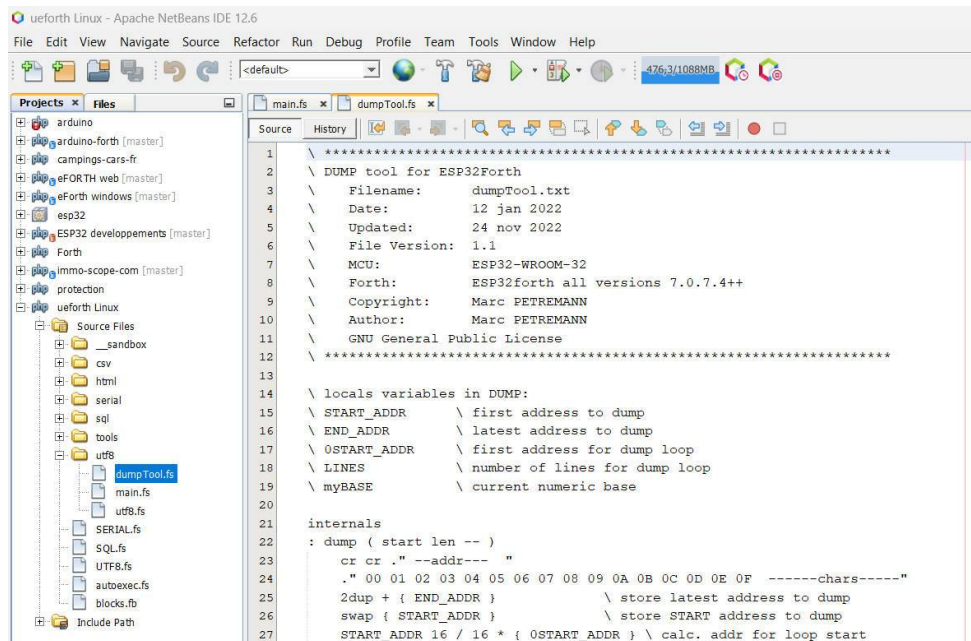


Figure 10: the new project is operational

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	LOTTO 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
```

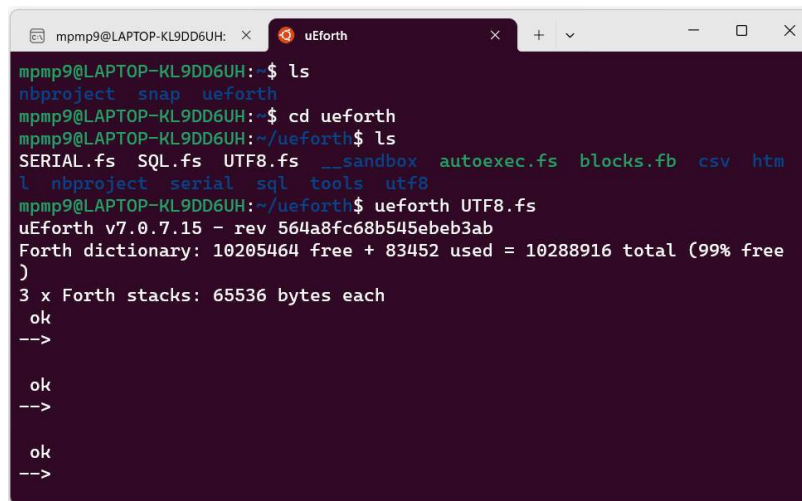
executes the contents of the **autoexec.fs** file .

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
```


A terminal window titled 'uEforth' is shown. The user 'mpmp9' is at a Linux prompt on a machine named 'LAPTOP-KL9DD6UH'. The terminal shows the following sequence of commands and output:

```
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 564a8fc68b545eb3ab
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.

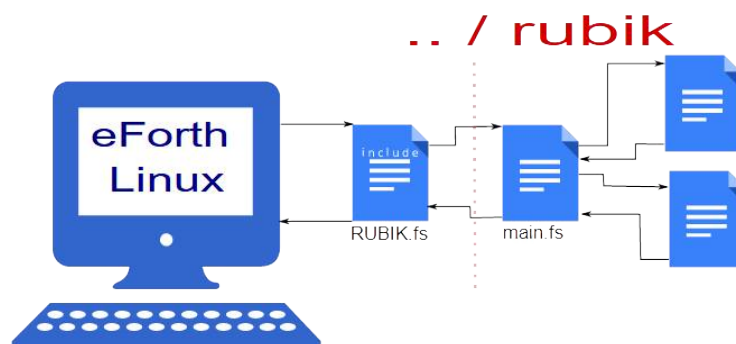


Figure 13: sequence of RUBIK project files

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:

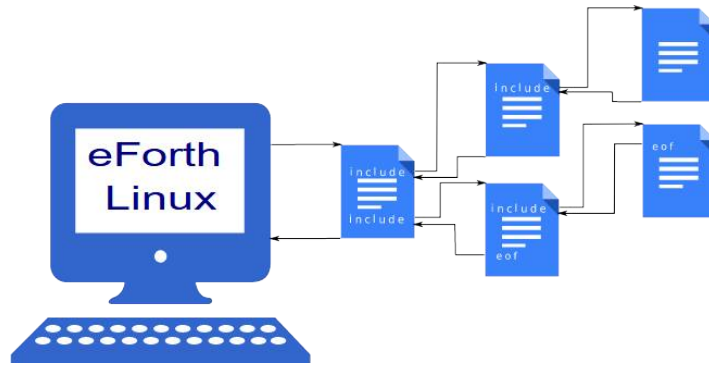


Figure 14: enchaînement de fichiers

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 IDE⁴ 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 -- f1 }
  0 { f1 }
  gridPos getGridAddr
  for
    aft
      getNumber n =
      if
        -1 to f1
      then
    then
  next
  drop
  f1
;
```

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

```
: inGrid? { n gridPos -- f1 }
  0 { f1 }   gridPos getGridAddr
  for aft
    getNumber n =
    if -1 to f1 then
  then
  next
  drop f1
;
```

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 ;
- **b1 (-- 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 dictionary 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
    sendDataToSSD1306
;

```

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 PETREMAN
\   Author:         Marc PETREMAN
\   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 ( 0C --- 0F) \ 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.

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 dictionary
```

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
```

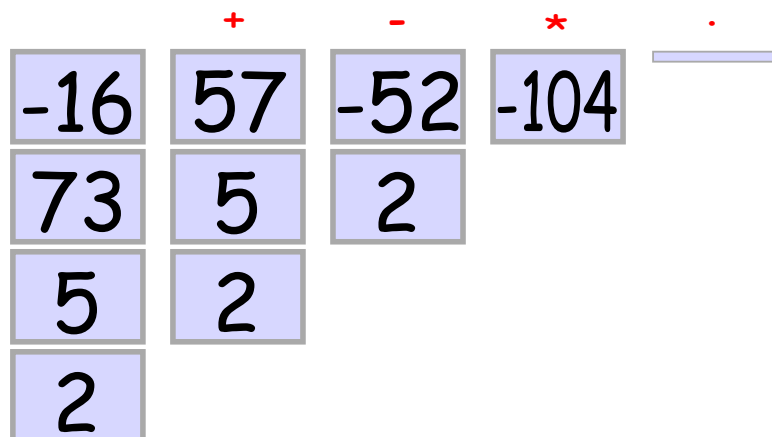
leaves the parameter stack as it is

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
```

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

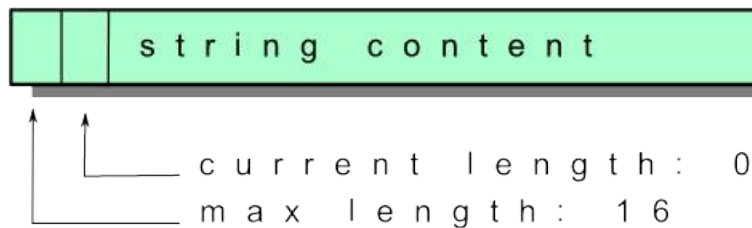
Alpha-numeric text variables do not exist natively in ESP32forth. Here is the first attempt to define the word **string** :

```
\ 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 --- f1)
  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 from 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 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
;
```

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?

Lexical index

allot.....	33	forget.....	29	value.....	33
BASE.....	34	HEX.....	34	variable.....	32
BINARY.....	34	HOLD.....	35	29
c!.....	32	include.....	18	29
c@.....	32	mv.....	18	.s.....	
cat.....	19	r@.....	31	28
constant.....	32	r>.....	31	#.....	35
create.....	33, 43	rdrop.....	31	#>.....	35
DECIMAL.....	34	rm.....	18	#S.....	35
delete file.....	18	S".....	38	<#.....	35
DOES>.....	43	see.....	27	>r.....	31
dump.....	27	SPACE.....	38		
files list.....	18	type.....	9		