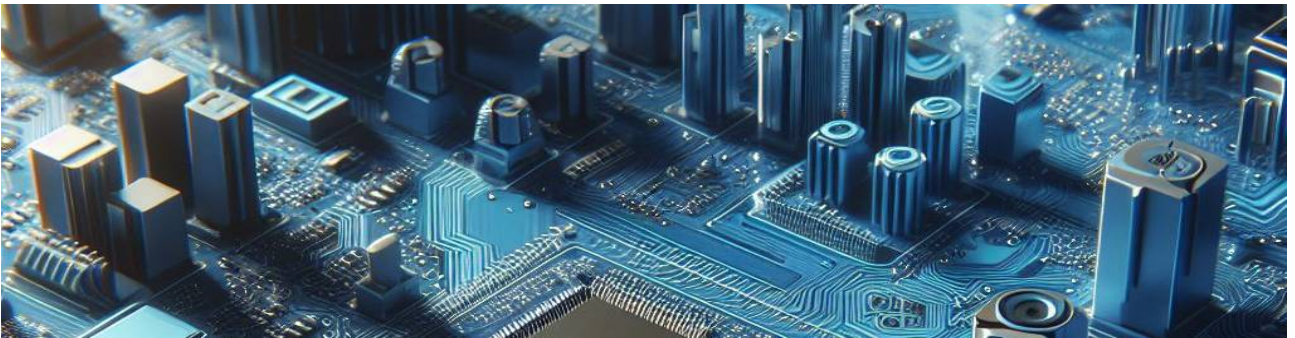


The great book for eFORTH Windows

version 1.2 - 14 novembre 2024



Author

- Marc PETREMANN

Contents

Author.....	1
Introduction.....	4
Translation help.....	4
Why program in FORTH language on eForth Windows?.....	5
Preamble.....	5
Limits between language and application.....	5
What is a FORTH word?.....	6
A word is a function?.....	6
FORTH language compared to C language.....	7
What FORTH can do compared to the C language.....	8
But why a stack rather than variables?.....	8
Are you convinced?.....	9
Are there any professional applications written in FORTH?.....	9
Installation on Windows.....	11
Setting up eForth Windows.....	11
A real 64-bit Forth with eForth Windows.....	14
Values on the data stack.....	14
Values in memory.....	14
Processing by words according to size or type of data.....	15
Conclusion.....	16
Editing and managing source files for eForth Windows.....	18
Text File Editors.....	18
Using an IDE.....	19
Storage on GitHub.....	21
Some good practices.....	21
The main.fs file.....	22
Example of project organization.....	23
Comments and program development.....	25
Write readable FORTH code.....	25
Source code indentation.....	26
Comments.....	27
Stack Comments.....	27
Meaning of stack parameters in comments.....	28
Word Definition Words Comments.....	28
Text comments.....	29
Comment at the beginning of the source code.....	29
Diagnostic and tuning tools.....	30
The decompiler.....	30
Memory dump.....	30
Stack monitor.....	30
Dictionary / Stack / Variables / Constants.....	32
Expand the dictionary.....	32
Stacks and Reverse Polish Notation.....	32

Handling the parameter stack.....	34
The Return Stack and Its Uses.....	34
Memory Usage.....	35
Variables.....	35
Constants.....	35
Pseudo-constant values.....	35
Basic tools for memory allocation.....	36
Local variables with eForth Windows.....	37
Introduction.....	37
The Fake Stack Comment.....	37
Action on local variables.....	38
Expanding the graphics vocabulary for Windows.....	41
Definition of functions in graphics internals.....	42
Definition of words in graphics.....	43
Version v 7.0.7.15.....	44
FORTH.....	44
windows.....	44
Ressources.....	47
English.....	47
French.....	47
GitHub.....	47
Facebook.....	47

Introduction

Since 2019, I have been managing several websites dedicated to FORTH language developments for ARDUINO and ESP32 cards, as well as the eForth web – Linux - Windows versions:

- ARDUINO: <https://arduino-forth.com/>
- ESP32: <https://esp32.arduino-forth.com/>
- eForth web: <https://eforth.arduino-forth.com/>
- eForth Windows: <https://eforth.win.arduino-forth.com/>

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

Sooner or later – and as late as possible – it will happen 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 will disappear.

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

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

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

Translation help

Google Translate is easy to translate texts, but with errors. So I am asking for help to correct the translations.

In practice, I provide the chapters already translated, in 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

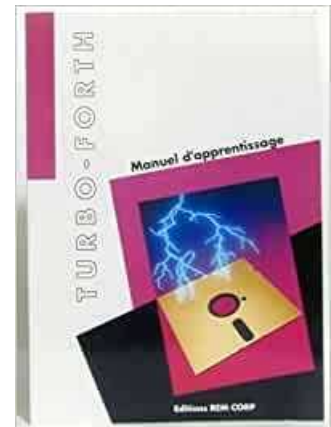
Why program in FORTH language on eForth Windows?

Preamble

I have been programming in Forth since 1983. I stopped programming in Forth in 1996. But I have always monitored the evolution of this language. I resumed Forth programming in 2019 on ARDUINO with FlashForth then ESP32forth.

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

- Introduction to the ZX-FORTH (ed Eyrolles - 1984 - ASIN:B0014IGOXO)
- FORTH Towers (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH for CP/M and MSDOS (ed Loistech - 1986)
- TURBO-Forth, learning manual (ed Rem CORP - 1990)
- TURBO-Forth, reference guide (ed Rem CORP - 1991)



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

The two software designers at this company were programming in C: Borland's TURBO-C to be precise. And their code couldn't be compact and fast enough to fit into the 64 kilobytes of RAM. It was 1992 and flash memory extensions didn't exist. In those 64 KB of RAM, they had to fit MS-DOS 3.0 and the application!

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

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

Limits between language and application

All programming languages are shared like this:

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

The FORTH language is an exception. It integrates:

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

What is a FORTH word?

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

Some FORTH words can only be used in compilation: **if else then** for example.

With the FORTH language, the essential principle is that you don't create an application. In FORTH, you extend the dictionary! Each new word that 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 !      \ desactive l'echo d'affichage du terminal
  ['] serial2-type is type
;
: typeToTerm ( -- )
  ['] default-type is type
  -1 echo !     \ active l'echo d'affichage du 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 language:

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

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

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

Yes, but then why FORTH rather than C?

I expected this question.

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

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

Compilation of programs written in FORTH language is done in eForth Windows. Example:

```
: >gray ( n -- n' )  
  dup 2/ xor      \ n' = n xor ( 1 décalage logique a droite )  
  ;
```

This definition is transmitted by copy/paste into the terminal. 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 be transformed.

FORTH language compared to C language

This is the part I like the least. I don't like comparing FORTH language to C language. But since almost all developers use C language, I'll try the exercise.

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

```
if(j > 13){  
    // Si tous les bits sont recus  
    rc5_ok = 1;      // Le processus de decodage est OK  
    detachInterrupt(0); // Desactiver l'interruption externe (INT0)  
    return;  
}
```

Test with `if` in FORTH language (code extract):

```
var-j @ 13 >      \ Si tous les bits sont recus  
  if  
    1 rc5_ok !    \ Le processus de decodage est OK  
    di            \ Desactiver l'interruption externe (INT0)  
    exit  
  then  
then
```

Here is the initialization of registers in C language:

```
void setup() {  
  // Configuration du module Timer1  
  TCCR1A = 0;  
  TCCR1B = 0;      // Desactive le module Timer1  
  TCNT1 = 0;       // Definit valeur préchargement Timer1 sur 0 (reset)  
  TIMSK1 = 1;      // activer interruption de debordement Timer1  
}
```

The same definition in FORTH language:

```
: setup ( -- )
  \ Configuration du module Timer1
  0 TCCR1A !
  0 TCCR1B !      \ Desactive le module Timer1
  0 TCNT1 !       \ Définit valeur préchargement Timer1 sur 0 (reset)
  1 TIMSK1 !      \ activer interruption de débordement Timer1
;
```

What FORTH can do compared to the C language

As we understand, FORTH gives immediate access to all the words in the dictionary, but not only that. Via the interpreter, we also access all the memory allocated to eForth Windows:

```
hex here 100 dump
```

You should see something like 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 the flash memory.

And that, the C language could not do?

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

Let's look at another case highlighting the extraordinary compactness of the 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 uses a stack, but you don't have access to it.

Only the FORTH language gives full access to the data stack. For example, to do an addition, we stack two values, perform the addition, and display the result: **2 5 + .** displays **7 .**

It's a bit unsettling, but once 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 faster than by variable processing as in C or any other variable-based language.

Are you convinced?

Personally, I doubt that this chapter alone will convert you irremediably to FORTH programming. In seeking to master WINDOWS, you have two possibilities:

- program in C language and exploit the many libraries available. But you will remain locked into the capabilities of these libraries. Adapting codes in C language requires real knowledge of programming in C language and mastering the WINDOWS architecture. Developing complex programs will always be a concern.
- try the FORTH adventure and explore a new and exciting world. Of course, it won't be easy. You'll need to understand the WINDOWS architecture, libraries... In return, you'll have access to programming that's perfectly adapted to your projects.

Are there any professional applications written in FORTH?

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

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

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

The choice of FORTH language for professional applications is 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 perfectly reliable, this word becomes a black box, that is to say a function whose proper functioning is trusted without limit. Word by word, a complex program is more easily made reliable in FORTH than in any other programming language.

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



Finally, it is possible, in FORTH language, to write the words you define in any human language. However, the usable characters are limited to the ASCII character set between 33 and 127. Here is how one could symbolically rewrite the words **high** and **low** :

```
\ Active broche de port, ne changez pas les autres.  
: __/ ( pinmask portadr -- )  
  mset  
;  
\ Desactivez une broche de port, ne change pas les autres.  
: \__ ( pinmask portadr -- )  
  mclr  
;
```

From this point on, to turn on the LED, you can type:

```
_0_ __/ \ turns on LED
```

Yes! The sequence **_0_ __/** is in FORTH language!

Good programming.

Installation on Windows

You can find the latest versions of eFORTH for WINDOWS here:

<https://eforth.appspot.com/windows.html>

The program version to be downloaded is in STABLE RELEASE or Beta Release.

Since μ Eforth version 7.0.7.21 only the 64 version remains available.

The downloaded program is directly executable. Once the program is downloaded, start by copying it to a working folder. Here, I chose to put the downloaded program in a folder named **eforth**.

To run μ Eforth Windows, click on the downloaded program and copy it to this eforth folder. If Windows issues a warning message:

- Click on Additional Information
- then click *Run Anyway*

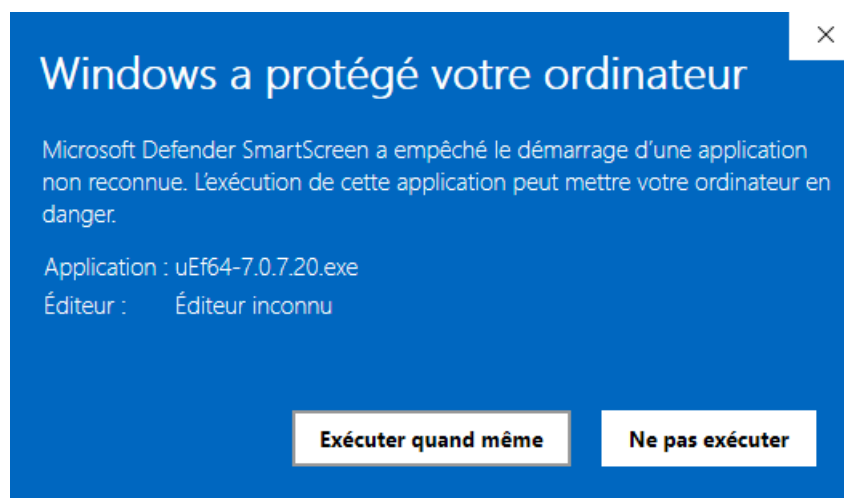


Figure 1: skip windows alert message

Once you have validated this choice, you will be able to run eForth like any other Windows program.

Setting up eForth Windows

eFORTH does not need to be installed to work. We simply run the downloaded file, here **uEf64-7.0.7.21.exe**. Here is the μ Eforth window:

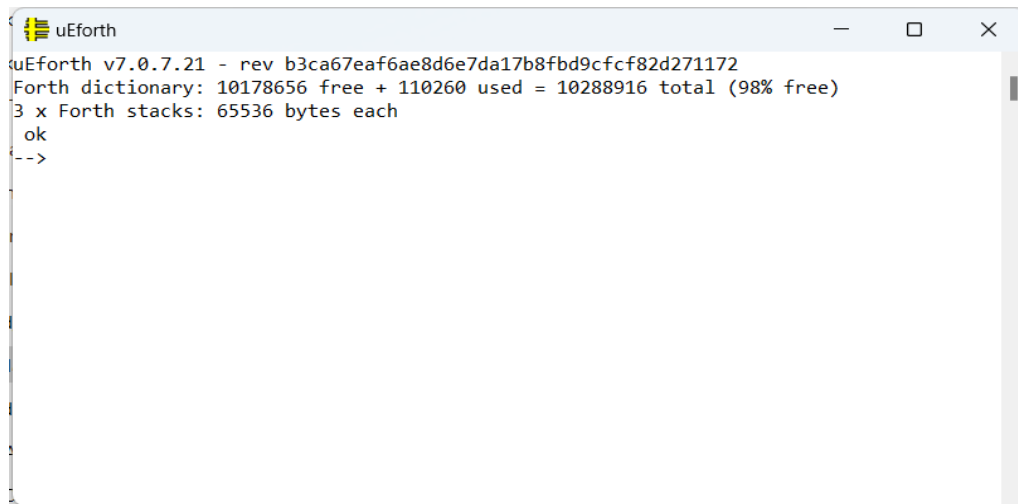


Figure 2: The μ Eforth window on Windows

To test that eForth is working properly, type **words** .

To exit eForth, type **bye** .

When eForth is open, you can create a shortcut to pin to the taskbar, which will make it easier to launch eForth again.

To change the background and text colors of eForth, hover over the μ eForth logo, right-click and select *Properties*:

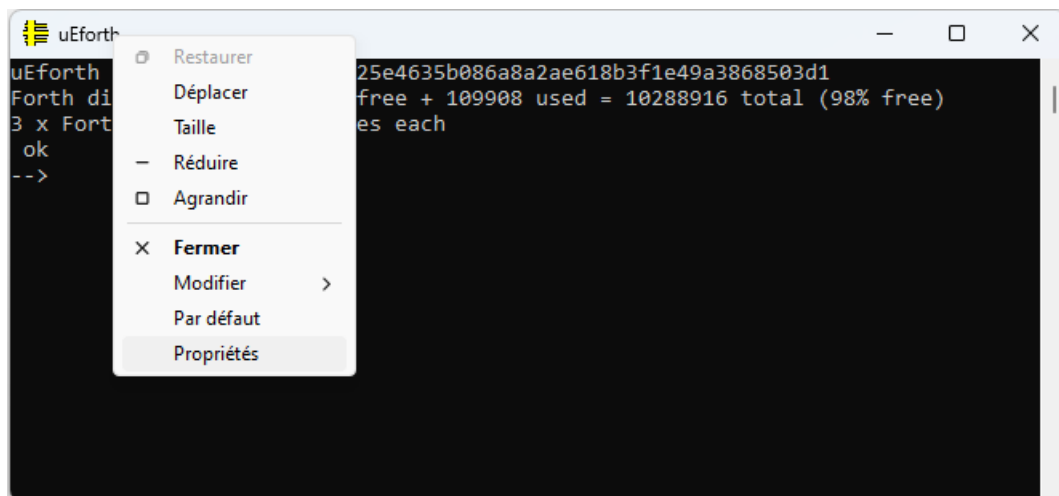


Figure 3: Selecting display properties

In Properties, click the *Colors* tab :

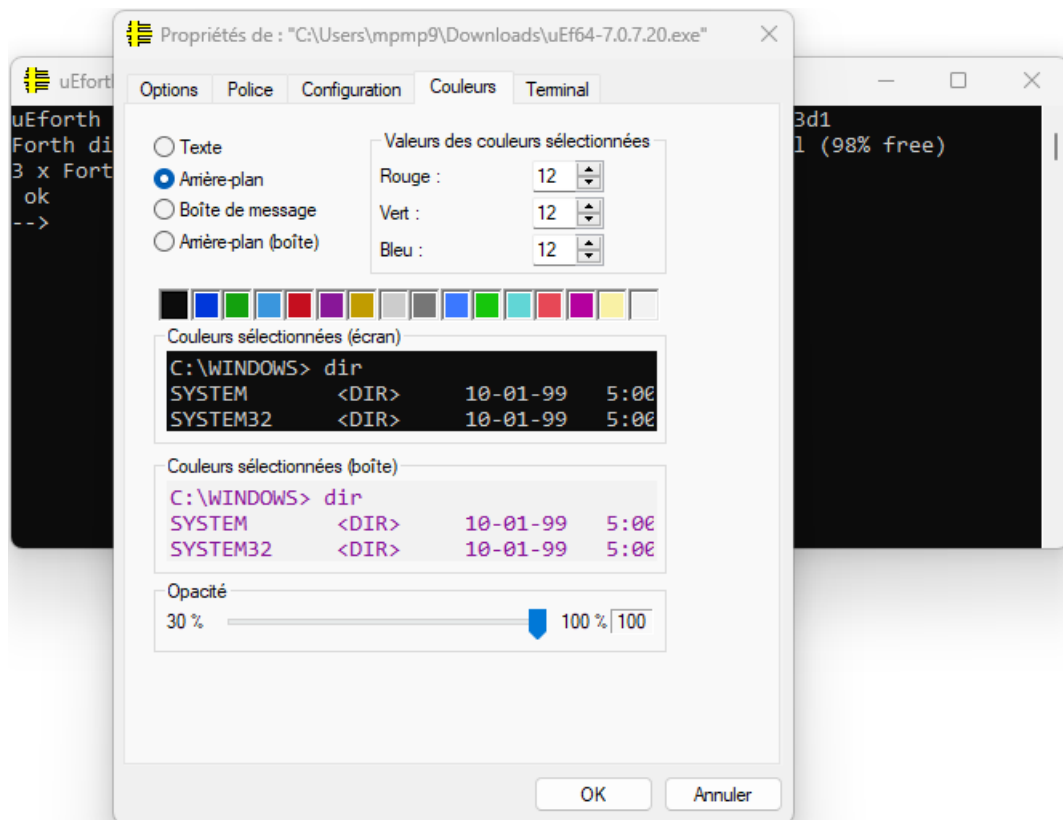


Figure 4: Choice of display colors

For my part, I chose to display the text in black on a white background. Click OK to validate this choice. The next time you launch eForth, you will find the colors selected as default settings for display in the eForth window.

A real 64-bit Forth with eForth Windows

eForth Windows is a real 64-bit FORTH. What does that 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 Windows starts, the FORTH interpreter is available. If you enter any number, it will be pushed 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 pushed onto the stack. To display this result, we will use the word **.** :

```
. \ displays 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 type** .

With eForth Windows, an ASCII character will be designated by a 64-bit integer, but whose value will be bounded [32..127]. Example:

```
67 emit \ display C
```

Values in memory

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

- one to define the number of morse code signs

- one or more bytes for each letter of Morse code

```
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, we store several bytes. The question is: how will we retrieve the information in these words?

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

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

The first byte extracted in this way will be used to manage a loop to display the Morse code of a letter:

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

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

Processing by words according to size or type of data

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

```
$bread = "Bake bread";
$price = 2.30;
echo $bread . " : " . $price;
// displays   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
\ displays    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 tailored processing methods that tell us about the nature of the data being 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 able to perform this HH:MM:SS conversion in such an elegant and concise way.

As you can see, the secret of FORTH is in its vocabulary.

Conclusion

FORTH has no data typing. All data flows 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 data stack 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 word of definition, just like **constant** or **variable** ...

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

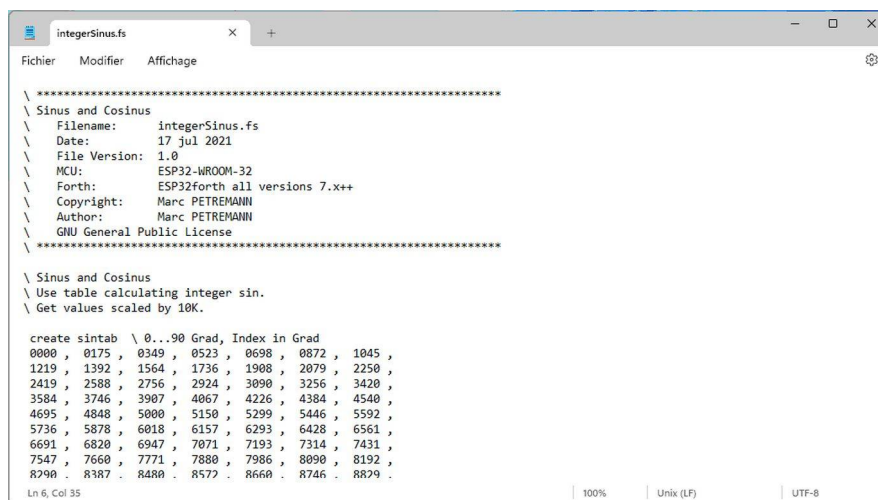
Editing and managing source files for eForth Windows

As with the vast majority of programming languages, source files written in FORTH are in plain text format. The extension of files in FORTH 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

edit file editor is the simplest:



```
\ integerSinus.fs
\ *****
\ Sinus and Cosinus
\ Filename: integerSinus.fs
\ Date: 17 jul 2021
\ File Version: 1.0
\ MCU: ESP32-WROOM-32
\ Forth: ESP32Forth all versions 7.x++
\ Copyright: Marc PETREMANN
\ Author: Marc PETREMANN
\ GNU General Public License
\ *****

\ Sinus and Cosinus
\ Use table calculating integer sin.
\ Get values scaled by 10K.

create sintab \ 0...90 Grad, Index in Grad
0000 , 0175 , 0349 , 0523 , 0698 , 0872 , 1045 ,
1219 , 1392 , 1564 , 1736 , 1908 , 2079 , 2250 ,
2419 , 2588 , 2756 , 2924 , 3090 , 3256 , 3420 ,
3584 , 3746 , 3907 , 4067 , 4226 , 4384 , 4540 ,
4695 , 4848 , 5000 , 5150 , 5299 , 5446 , 5592 ,
5736 , 5878 , 6018 , 6157 , 6293 , 6428 , 6561 ,
6691 , 6820 , 6947 , 7071 , 7193 , 7314 , 7431 ,
7547 , 7660 , 7771 , 7880 , 7986 , 8090 , 8192 ,
8290 , 8387 , 8480 , 8572 , 8660 , 8746 , 8829 ,
```

Figure 5: editing with *edit* under windows 11

Other editors, such as **WordPad** , are not recommended because you risk saving the FORTH source code in a file format that is not compatible with eForth Windows.

If you use a custom file extension, such as **fs** , for your FORTH source files, you must have this file extension recognized by your system to allow them to be opened by the text editor.

Using an IDE

Nothing prevents you from using an IDE ¹. For my part, I have a preference for **Netbeans** that I also use for PHP, MySQL, Javascript, C, assembler... It is a very powerful IDE and as efficient as **Eclipse** :

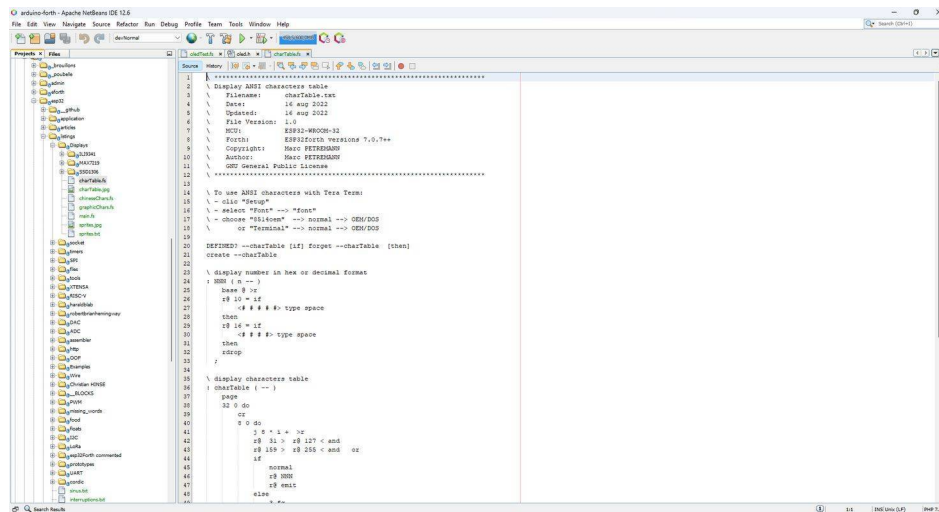


Figure 6: editing with Netbeans

Netbeans offers several interesting features:

- version control with **GIT** ;
- recovery of previous versions of modified files;
- file comparison with **Diff** ;
- one-click **FTP transmission** to the online hosting of your choice;

With the **GIT option** , you can share files on a repository and manage collaborations on complex projects. Locally or collaboratively, **GIT** allows you to manage different versions of the same project, then merge these versions. You can create your local GIT repository. Each time you *commit* a file or a complete directory, the developments are kept as is. This allows you to find old versions of the same file or folder of files.

¹ Integrated Development Environment

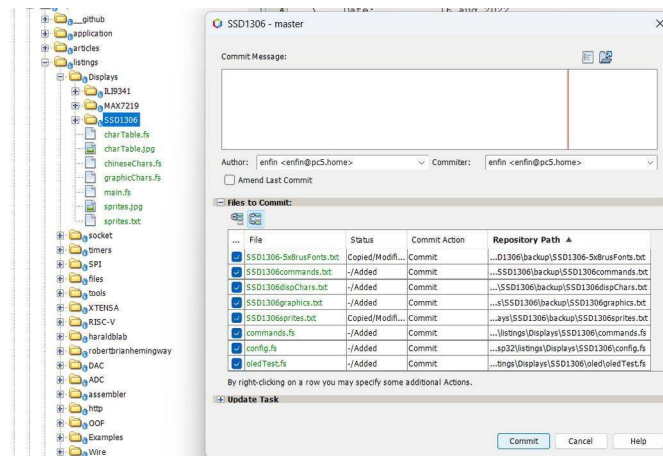


Figure 7: GIT operation in Netbeans

With NetBeans you can define a development branch for a complex project. Here we create a new branch:

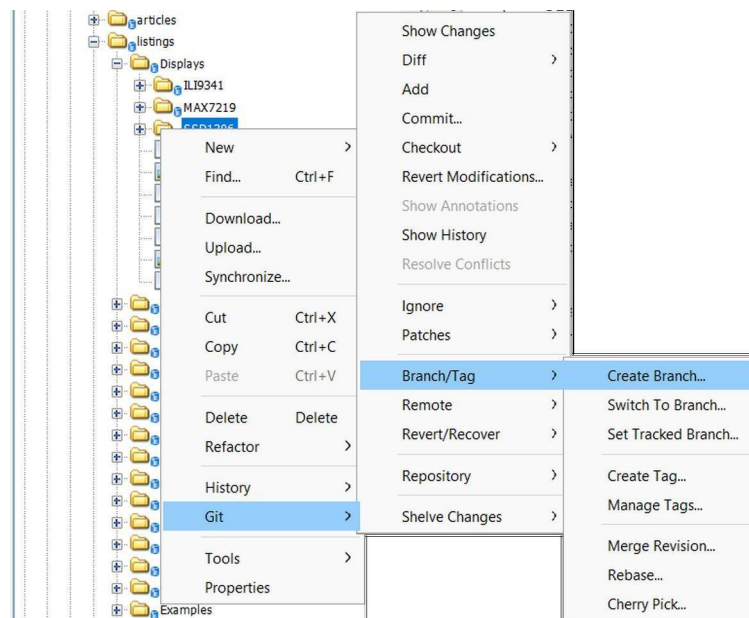


Figure 8: creating a branch on a project

Example of a situation that justifies the creation of a branch:

- you have a working project;
- you are considering optimizing it;
- create a branch and do the optimizations in that branch...

Changes to source files in a branch do not affect files in the *main trunk* .

Incidentally, it is more than advisable to have a physical backup medium. An SSD hard drive costs around €50 for 300Gb of storage space. The read or write access speed of an SSD medium is simply amazing!

Storage on GitHub

GitHub² website is, along with **SourceForge**³, one of the best places to store your source files. On GitHub, you can share a working folder with other developers and manage complex projects. The Netbeans editor can connect to the project and allows you to push or retrieve file changes.

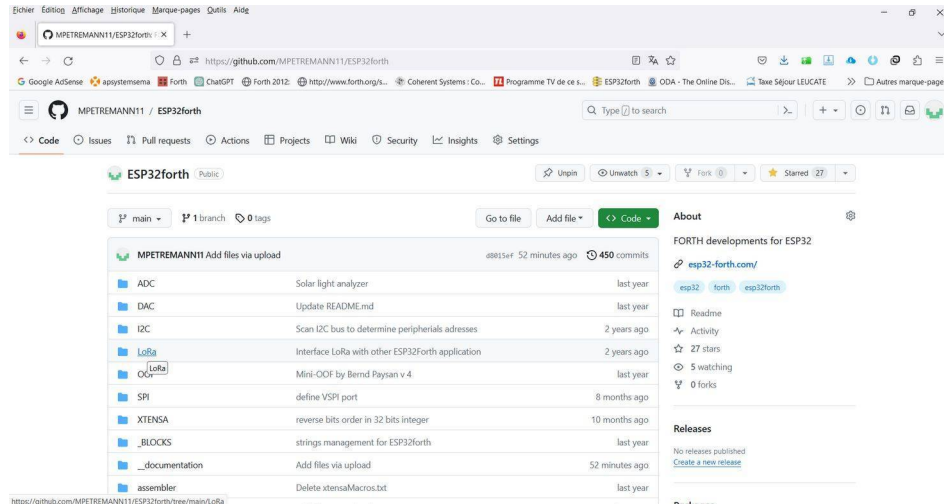


Figure 9: storing files on Github

On **GitHub**, you can manage project branches (*forks*). You can also make parts of your projects confidential. Here are the branches in flagxor/ueforth's projects:

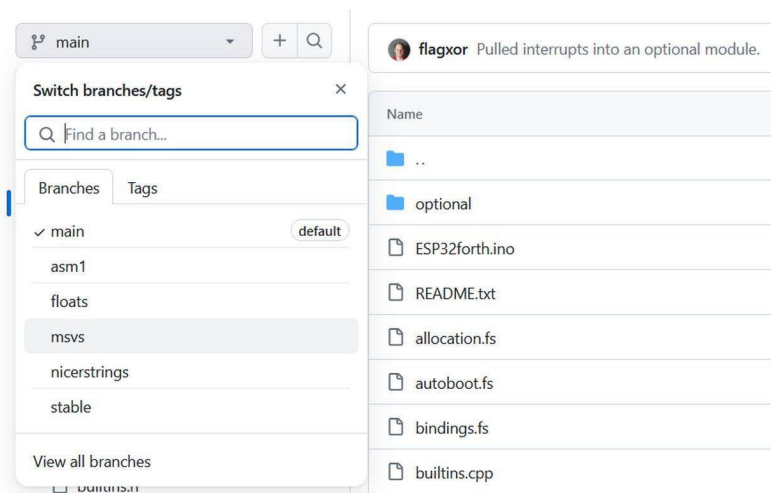


Figure 10: access to a project branch

Some good practices

The first best practice is to name your working files and folders well. You are developing for eForth Windows, so create a folder named **eForth**.

sandbox subfolder in this folder.

² <https://github.com/>

³ <https://sourceforge.net/>

For well-built projects, create one folder per project. For example, you want to make a game, create a subfolder **game** .

If you have general-purpose scripts, create a **tools folder** . If you use a file from that **tools folder** in a project, copy and paste that file into that project's folder. This will prevent a change to a file in **tools** from disrupting your project later.

The second best practice is to split 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;
- **myApp.fs** for your project definitions. Choose a file name that is fairly descriptive. For example, to manage a robot, take the name **robot-commands.fs**

..	
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: Forth source file naming example

It is the content of these files that will be processed by eForth Windows.

The main.fs file

Windows files are stored in the **eForth folder** and can be read by eForth Windows. If you have written a **config.fs** file, here is the line of code to write in **main.fs** to access the contents of **config.fs**:

```
include config.fs
```

From this point on, you have two possibilities to interpret the contents of **config.fs** . From the terminal:

```
include config.fs
```

Or

```
include main.fs
```

The point is that **main.fs** can call other files. Example:

```
\ load FLOG
include flog.fs

\ load FLOG tests
\ include tests/flogTest.fs
```

Processing many files takes less than a second.

Example of project organization

eforth folder which contains our Forth interpreter-compiler **uEf64-7.0.7.21.exe** :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
```

lotto subfolder :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
   └─ lottery
```

In this **lotto** subfolder, we create our **main.fs** file :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
   └─ lottery
      └─ main.fs
```

main.fs contents :

```
DEFINED? --loto [if] forget --loto [then]
create --loto
include generalWords.fs
include euroMillionFR.fs
include gridsManage.fs
include numbersFrequency.fs
include interface.fs
interface \ run main program
```

main.fs file calls other source files via **include**. Here is the organization of the files in this state of the project:

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
   └─ lottery
      └─ main.fs
      └─ generalWords.fs
      └─ euroMillionFR.fs
      └─ gridsManage.fs
      └─ numbersFrequency.fs
      └─ interface.fs
```

And finally, for practical reasons, we create the **LOTTO.fs** file in our **eforth folder** :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
   └─ lottery
      └─ main.fs
      └─ generalWords.fs
```

```
└─ euroMillionFR.fs
└─ gridsManage.fs
└─ numbersFrequency.fs
└─ interface.fs
└─ LOTTO.fs
```

Contents of this **LOTTO.fs** file :

```
include lotto/main.fs
```

To test our project, simply launch eForth, then in the eForth interface type **include LOTTO.fs**.

Automatically, eforth will load the contents of **lotto/main.fs** and all files called from **lotto/main.fs**.

Only the project call files and the subfolders of these projects should appear in the **eforth folder**.

Comments and program development

There is no IDE ⁴to 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
- **PsPad** on windows
- **Netbeans** on Windows...

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

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

This code will be perfectly compiled by eForth Windows. 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 naming of the word to be defined, here **inGrid?**. Eforth Windows 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:

- in the manner of object programming in javaScript: **grid.test.number**
- in the CamelCoding way **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 proofread your FORTH code. However, FORTH language computer programmers have certain habits:

- **LOTTO_NUMBERS_IN_GRID** uppercase constants
- definition word of other words **lottoNumber:** 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**

And what about naming FORTH words in a language other than English? Again, there is only one rule: **total freedom** ! Be careful though, eForth Windows does not accept

4 Integrated Development Environment

names written in alphabets other than the Latin alphabet. You can, however, use these alphabets for comments:

```
: .date      \ Плакат сегодняшней даты
...code...  ;
```

Or

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

Source code indentation

Whether the code is on two lines, ten lines or more, it 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;
- one line per execution of a predefined word, preceded where appropriate 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 `(` followed by at least one space character. This comment is completed by the character `)` ;
- the word `\` 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 - n n )
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. We can therefore put anything between `(` and `)` . As for stack comments, we will remain very concise. The `-- sign` symbolizes the action of a FORTH word. The indications appearing before `--` correspond to the data placed on the data stack before the execution of the word. The indications appearing after `--` correspond to the data left on the data stack after the 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 an input data 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 the word is executed. As a reminder, the indications between `(` and `)` are for information purposes only.

Meaning of stack parameters in comments

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

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

c parameter indicates that we are stacking 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.

So we could 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 one delivered by a variable;
- **c** indicates an 8-bit value in the range [0..255]
- **d** indicates a double precision value.
Not used with eForth Windows which is already 32 or 64 bit;
- **fl** indicates a boolean value, 0 or non-zero;
- **n** indicates an integer. 32- or 64-bit signed integer for eForth Windows;
- **str** indicates a string. Equivalent to **addr len --**
- **u** indicates an unsigned integer

There is nothing to prevent us from being a little more explicit:

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

Word Definition Words Comments

Definition words use **create** and **does>** . For these words, it is recommended 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
  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 by **exec**:

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

Text 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 between
\ <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 end up with hundreds, even thousands of source files. To avoid file selection errors, it is strongly recommended to mark the beginning of each source file with a comment:

```

\ *****
\ key word for UT8 characters
\  Filename:      uekey.fs
\   Date:        29 nov 2023
\  Updated:      29 nov 2023
\  File Version: 1.1
\   MCU:         Linux / Web / Windows
\  Forth:        eForth Windows
\  Copyright:    Marc PETREMANN
\  Author:       Marc PETREMANN
\  GNU General Public License

```

```
\ *****
```

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

Finally, 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, the object code must be linked. At no time can the programmer access the executable code contained in his libraries with the compiler's resources alone.

With eForth Windows, the developer can decompile his 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 the eForth Windows 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 **dump word** accepts two parameters: the starting address in memory and the number of bytes to view:

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

Stack monitor

The contents of the data stack can be displayed at any time using the **.s** keyword. Here is the definition of the **.DEBUG** keyword which uses **.s** :

```

variable debugStack

: debugOn ( -- )
  -1 debugStack !
;

: debugOff ( -- )
  0 debugStack !
;

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

```

To exploit **.DEBUG** , simply insert it in a strategic location 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 executing the word **i** in our **do loop** . We activate the debug and execute **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 data stack pauses our **do loop**. Run **debugOff** to make the word **myTEST** run normally.

Dictionary / Stack / Variables / Constants

Expand the dictionary

Forth belongs to the class of woven interpreter 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 ones are `:` (start a new definition) and `;` (completes the definition). Let's try this by typing:

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

What happened? The action of `:` is to create a new dictionary entry named `*+` and switch from interpreter mode to compile mode. In compile mode, the interpreter looks up 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 Windows builds the number in the dictionary in the space allocated for the new word, following special code that pushes the stored number onto the stack each time the word is executed. The action of executing `*+` is therefore to sequentially execute the previously defined words `*` and `+`.

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

Now try your new word:

```
decimal 5 6 7 *+ . \ displays 47
```

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

Stacks and Reverse Polish Notation

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

In FORTH, the stack is analogous to a stack 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 Windows integrates two stacks: the parameter stack and the return stack, each consisting of a number of cells that can hold 16-bit numbers.

The FORTH input line:

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

leaves the parameter stack as is

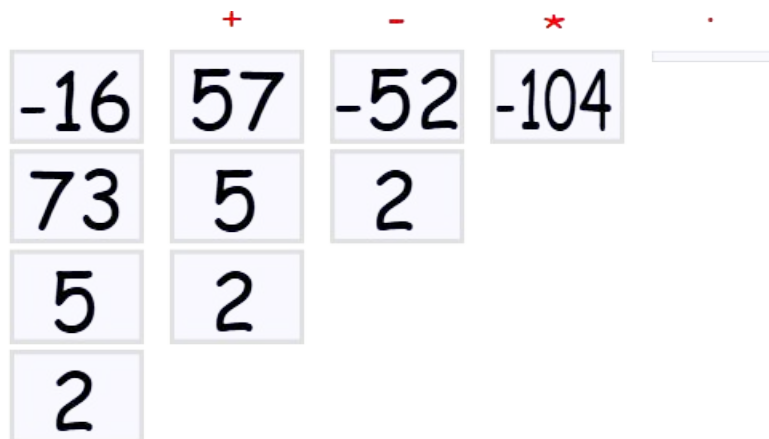
Cell	content	comment
0	-16	(TOS) Top stack
1	73	(NOS) Next in the stack
2	5	
3	2	

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.

Suppose we follow the original input line with the line

```
+ - * .
```

The operations would produce the successive stack operations:



After the two lines, the console displays:

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

Note that eForth Windows conveniently displays the stack elements as it interprets each line, and the value of -16 is displayed as a 32-bit unsigned integer. Also, the word `.` consumes the 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 Windows must provide ways to put numbers on the stack, to remove them, and to rearrange their order. We have already seen that we can put numbers on the stack simply by typing them. We can also integrate the numbers into the definition of a FORTH word.

The word **drop** removes a number from the top of the stack, putting the next number on top. The word **swap** swaps the first 2 numbers. **dup** copies the number on top, pushing all the other numbers down. **rot** rotates the first 3 numbers. These actions are shown below.



The Return Stack and Its Uses

When compiling a new word, eForth Windows 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 (rstack). The address of the next word to be invoked is placed on the return stack so that when the current word is completed during execution, the system knows where to jump 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 the eForth Windows system. Despite the danger, there are times when using the return stack as temporary storage can make your code less complex.

To store on the 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 of the return stack to the top of the parameter stack. To simply remove a value from the top of the stack, there is the word **rdrop**. The word **r@** copies the top of the return stack to the parameter stack.

Memory Usage

In eForth Windows, 32-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 memory cells using **c@** and **c!** .

```
create testVar
cell allot
$F7 testVar c!
testVar c@ .          \ displays 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 by leaving the address of its storage location on top of the stack:

```
x .          \ displays the address
```

We can then collect or store at this address:

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

```
\ defines extrem values for alpha channel
255 constant SDL_ALPHA_OPAQUE
0    constant SDL_ALPHA_TRANSPARENT
```

Pseudo-constant values

A value defined with value is a hybrid type of variable and constant. We define and initialize a value and it is invoked as we would a constant. We can also change a value as 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 on top of the stack. You have 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 graphic-array dictionary entry :

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

When executed, the **graphic-array** word will push 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 allocated to **graphic-array** we can write **graphic-array 2 +** , remembering that indices start at 0.

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

Local variables with eForth Windows

Introduction

The FORTH language processes data primarily through the data stack. This very simple mechanism offers unmatched performance. On the other hand, following the path of data can quickly become complex. Local variables offer an interesting alternative.

The Fake Stack Comment

If you follow the various FORTH examples, you will have noticed the stack comments surrounded 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 effect on the rest of the FORTH code. It is a pure comment.

When preparing a complicated definition, the solution is to use local variables surrounded 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 `}` look like the words `(` and `)` but do not have the same effect at all. The 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 could just 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 stack data in the order they are placed 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 purpose 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 this:

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

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

We assign a value to a local variable with the word **to** or **+to** to increment the contents 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 on the data stack the sum and the overflow value of this sum:

```

\ addition 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** by exploiting local variables:

```

\ local variables in DUMP:
\ START_ADDR      \ first address for dump
\ END_ADDR        \ last address for dump
\ ØSTART_ADDR     \ first address for loop in dump
\ LINES           \ number of lines for the dump loop
\ myBASE          \ current numeric base
: 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 * { ØSTART_ADDR } \ calc. addr for loop start
  16 / 1+ { LINES }
  base @ { myBASE }            \ save current base
  hex
  \ outer loop
  LINES 0 do
    ØSTART_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
      ØSTART_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
      ØSTART_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

Using 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, do not use the words **>r** and **r>** anymore, otherwise you risk disturbing 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 ;
```


Expanding the graphics vocabulary for Windows

eFORTH allows access to Windows function libraries using the word **dll** .

In the eForth source code, here is how the connection to the **Gdi32 library is done** :

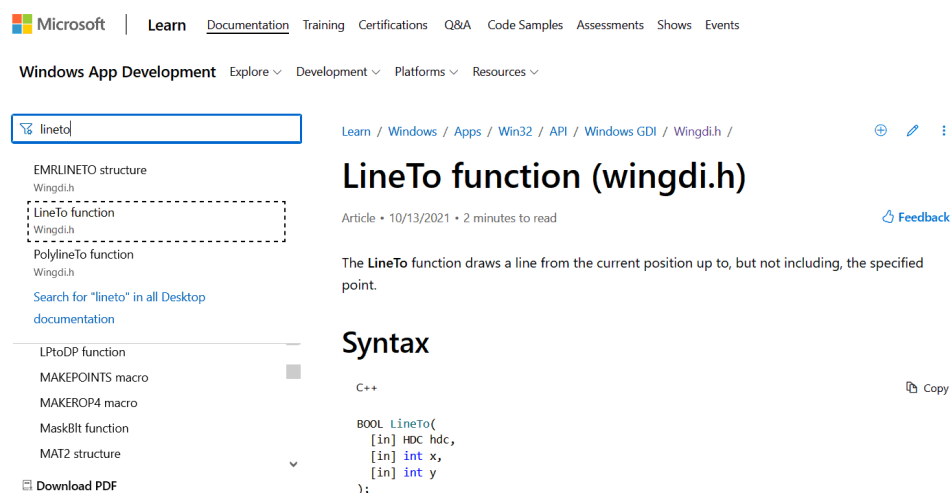
```
windows definitions
z" Gdi32.dll" dll Gdi32
```

Here, the word **Gdi32** becomes the entry point to define the words giving access to this library **Gdi32.dll** .

From this point on, every word defined for eFORTH using this **Gdi32 library** refers to the Microsoft documentation:

<https://learn.microsoft.com/en-us/windows/win32/api/wingdi/>

Here we will look for the documentation of the **LineTo function** :



The screenshot shows the Microsoft Learn documentation for the **LineTo function (wingdi.h)**. The page includes a search bar with "lineto" entered, a list of search results, and the main content area. The main content area shows the function name, a brief description, and the C++ syntax.

Search results:

- EMRLINETO structure
- Wingdi.h
- LineTo function
- Wingdi.h
- PolylineTo function
- Wingdi.h

Search for "lineto" in all Desktop documentation

LineTo function (wingdi.h)

Article • 10/13/2021 • 2 minutes to read

The **LineTo** function draws a line from the current position up to, but not including, the specified point.

Syntax

```
C++
BOOL LineTo(
    [in] HDC hdc,
    [in] int x,
    [in] int y
);
```

Figure 12: LineTo documentation

In this documentation, for the **LineTo function** , it is stated:

- accepts three input parameters: hdc, x and y
- returns a parameter as output: fl

The first reflex would therefore be to define the word eFORTH **LineTo** like this:

```
z" LineTo"      3 Gdi32 LineTo ( hdc x y -- fl )
```

The value 3 preceding the word **Gdi32** indicates that the called function must use three parameters.

If we agree to use the word **LineTo** in this way, we would be obliged, each time we use it, to proceed as follows:

```
graphics internals
: drawLines ( -- )
  hdc 20 20 LineTo drop
  hdc 50 20 LineTo drop
  hdc 50 50 LineTo drop
  hdc 20 50 LineTo drop
  hdc 45 45 LineTo drop
;
```

The systematic call to the **hdc** ticket is not necessary if you manage a single Windows window. Similarly, using **drop** after **LineTo** makes the eFORTH code heavier. The solution, to simplify these words, is to define them in two forms in two different vocabularies.

Definition of functions in graphics internals

Gdi32 library functions will be renamed by prefixing them with Gdi:

- **Gdi.LineTo** accesses **LineTo** ;
- **Gdi.Rectangle** accesses **Rectangle** , etc.

All words will be defined in the **graphics internals** vocabulary :

```
graphics internals definitions
windows also

\ The LineTo function draw a line.
z" LineTo"      3 Gdi32 Gdi.LineTo ( hdc x y -- fl )

z" Rectangle"   5 gdi32 Gdi.Rectangle ( hdc left top right bottom -- fl )

z" Ellipse"     5 gdi32 Gdi.Ellipse ( hdc left top right bottom -- fl )

\ The CloseFigure function close a figure in a path.
z" CloseFigure" 1 gdi32 Gdi.CloseFigure ( hdc -- fl )

\ The GetPixel function retrieves the red, green, blue (RGB) color value
\ of the pixel at the specified coordinates.
z" GetPixel"    3 gdi32 Gdi.GetPixel ( hdc x y -- color )

\ The SetPixel function sets the pixel at the specified coordinates
\ to the specified color.
z" SetPixel"    4 gdi32 Gdi.SetPixel ( hdc x y colorref -- colorref )
```

It is easy to check the correct compilation of these words in the **graphics internals** vocabulary :

```
Gdi.SetPixel Gdi.GetPixel Gdi.CloseFigure Gdi.Ellipse Gdi.Rectangle Gdi.LineTo
GrfWindowProc msg>pressed msg>button rescale binfo msgbuf ps hdc hwnd GrfClass
hinstance GrfWindowTitle GrfClassName 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
```

Here we have highlighted the new eFORTH words connected to the functions of the **Gdi32** library.

Definition of words in graphics

We will now define simplified graphic words in the **graphics vocabulary** :

```
only forth
graphics definitions internals

: lineTo ( x y -- )
  hdc -rot Gdi.LineTo drop ;
: rectangle ( left top right bottom -- )
  >r >r >r >r hdc r> r> r> r> Gdi.Rectangle drop ;
: closeFigure ( -- )
  hdc Gdi.CloseFigure drop ;
: getPixel ( x y -- colorref )
  hdc -rot Gdi.GetPixel ;
: setPixel ( x y color -- )
  hdc -rot Gdi.GetPixel ;
```

To use these words, let's go back to our example:

```
graphics
: drawLines ( -- )
  20 20 lineTo
  50 20 lineTo
  50 50 lineTo
  20 50 lineTo
  45 45 lineTo
;
```

Finally, don't try to extend eForth with all the functions of each bookstore. You would spend years on it!

The quickest and easiest strategy is to define only the words that exploit the functions you are interested in. Windows programming is very complex and requires the acquisition of solid foundations.

Version v 7.0.7.15

FORTH

=	-1	-rot	└	.	.	:noname	!
?	?do	?dup	└	."	.s	!	(local)
 	['']	[char]	[ELSE]	[IF]	[THEN]	 	f
}transfer	@	*	*/	*/MOD	/	/mod	#
#!	#>	#fs	#s	#tib	+	+!	+loop
+to	<	<#	<=	<>	=	>	>=
>BODY	>flags	>flags&	>in	>link	>link&	>name	>params
>R	>size	0	0<	0<>	0=	1	1-
1/F	1+	10	2!	2@	2*	2/	2drop
2dup	3dup	4*	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	CELL	cell/	cell+	cells
char	CLOSE-FILE	cmove	cmove>	CONSTANT	context	copy	cp
cr	CREATE	CREATE-FILE	current	decimal	default-key	default-key?	depth
default-type	default-use	defer	DEFINED?	definitions	DELETE-FILE	depth	editor
do	DOES>	DROP	dump	dump-file	DUP	echo	evaluate
else	emit	empty-buffers	ENDCASE	ENDOF	erase	F*	F**
EXECUTE	exit	extract	F-	f.	f.s	F>	F>=
F/	F+	F<	F<=	F<>	F=	FCOS	fdepth
F>S	F0<	F0=	FABS	FATAN2	fconstant	FILE-POSITION	flush
FDROP	FDUP	FEXP	fq	file-exists?	FLOOR	flush	FLUSH-FILE
FILE-SIZE	fill	FIND	fliteral	FLN	forget	FORTH	forth-builtins
FMAX	FMIN	FNEGATE	FNIP	for	FROT	FSIN	FSINCOS
FOVER	FP!	FP@	fp0	free	hex	hld	hold
FSORT	FSWAP	fvariable	graphics	here	included?	internals	invert
I	if	IMMEDIATE	include	included	L!	latesttxt	leave
is	J	K	key	key?	L!	min *	mod
list	literal	load	loop	LSHIFT	max	next	nip
ms	ms-ticks	mv	n.	needs	negate	only	open-blocks
nl	NON-BLOCK	normal	octal	OF	ok	PARSE	pause
OPEN-FILE	OR	order	OVER	pad	page	prompt	quit
pause?	PI	postpone	postpone,	precision	previous	r~	rdrop
r"	R@	R/O	R/W	R>	r!	remember	RENAME-FILE
READ-FILE	recognizers	recurse	refill	remaining	resize	RESIZE-FILE	restore
REPOSITION-FILE	required	reset	RP!	rp0	RSHIFT	s"	S>F
rm	rot	save-buffers	SF!	scr	sealed	see	set-precision
S>Z	save	SF@	SF@	SFLOAT	SFLOAT+	SFLOATS	sign
set-title	sf,	SP!	SP@	space	spaces	start-task	startswith?
SL@	SP!	str	str=	streams	structures	SW@	SWAP
startup:	state	str	str=	throw	thru	tib	to
task	tasks	terminate	then	u.	U/MOD	U<	UL@
touch	transfer	type	use	used	UW@	value	VARIABLE
UNLOOP	until	update	vocabulary	W!	while	windows	words
visual	vlist	z"	z>s				
WRITE-FILE	XOR						

windows

process-heap HeapReAlloc HeapFree HeapAlloc GetProcessHeap [WM_>name](#) [WM_PENWINLAST](#)
[WM_PENEVENT](#) [WM_CTLINIT](#) [WM_PENMISC](#) [WM_PENCTL](#) [WM_HEDITCTL](#) [WM_SKB](#) [WM_PENMISCINFO](#)
[WM_GLOBALRCCHANGE](#) [WM_HOOKRCRESULT](#) [WM_RCRESULT](#) [WM_PENWINFIRST](#) [WM_AFXLAST](#)
[WM_AFXFIRST](#) [WM_HANDHELDLAST](#) [WM_HANDHELDFIRST](#) [WM_APPCOMMAND](#) [WM_PRINTCLIENT](#)
[WM_PRINT](#) [WM_HOTKEY](#) [WM_PALETTECHANGED](#) [WM_PALETTEISCHANGING](#) [WM_QUERYNEWPALETTE](#)
[WM_HSCROLLCLIPBOARD](#) [WM_CHANGECHAIN](#) [WM_ASKCBFORMATNAME](#) [WM_SIZECLIPBOARD](#)
[WM_VSCROLLCLIPBOARD](#) [WM_PAINTCLIPBOARD](#) [WM_DRAWCLIPBOARD](#) [WM_DESTROYCLIPBOARD](#)
[WM_RENDERALLFORMATS](#) [WM_RENDERFORMAT](#) [WM_UNDO](#) [WM_CLEAR](#) [WM_PASTE](#) [WM_COPY](#) [WM_CUT](#)
[WM_MOUSELEAVE](#) [WM_NCMOUSELEAVE](#) [WM_MOUSEHOVER](#) [WM_NCMOUSEHOVER](#) [WM_IME_KEYUP](#)
[WM_IMEKEYUP](#) [WM_IME_KEYDOWN](#) [WM_IMEKEYDOWN](#) [WM_IME_REQUEST](#) [WM_IME_CHAR](#) [WM_IME_SELECT](#)

WM_IME_COMPOSITIONFULL WM_IME_CONTROL WM_IME_NOTIFY WM_IME_SETCONTEXT WM_IME_REPORT
 WM_MDIREFRESHMENU WM_DROPFILES WM_EXITSIZEMOVE WM_ENTERSIZEMOVE WM_MDISETMENU
 WM_MDIGETACTIVE WM_MDICONARRANGE WM_MDICASCADE WM_MDTILE WM_MDMAXIMIZE
 WM_MDINEXT WM_MDISTORE WM_MDIACTIVATE WM_MDIDESTROY WM_MDICREATE WM_DEVICECHANGE
 WM_POWERBROADCAST WM_MOVING WM_CAPTURECHANGED WM_SIZING WM_NEXTMENU WM_EXITMENULOOP
 WM_ENTERMENULOOP WM_PARENTNOTIFY WM_MOUSEWHEEL WM_XBUTTONDOWNBLCLK WM_XBUTTONUP
 WM_XBUTTONDOWN WM_MOUSEWHEEL WM_MOUSELAST WM_MBUTTONDOWNBLCLK WM_MBUTTONUP
WM_MBUTTONDOWN WM_RBUTTONDOWNBLCLK WM_RBUTTONUP WM_RBUTTONDOWN WM_LBUTTONDOWNBLCLK
WM_LBUTTONUP WM_LBUTTONDOWN WM_MOUSEMOVE WM_MOUSEFIRST CB_MSGMAX CB_GETCOMBOBOXINFO
CB_MULTIPLEADDSTRING CB_INITSTORAGE CB_SETDROPPEDWIDTH CB_GETDROPPEDWIDTH
CB_SETHORIZONTALEXTENT CB_GETHORIZONTALEXTENT CB_SETTOPINDEX CB_GETTOPINDEX
CB_GETLOCALE CB_SETLOCALE CB_FINDSTRINGEXACT CB_GETDROPPEDSTATE CB_GETEXTENDEDUI
CB_SETEXTENDEDUI CB_GETITEMHEIGHT CB_SETITEMHEIGHT CB_GETDROPPEDCONTROLRECT
CB_SETITEMDATA CB_GETITEMDATA CB_SHOWDROPDOWN CB_SETCURSEL CB_SELECTSTRING
CB_FINDSTRING CB_RESETCONTENT CB_INSERTSTRING CB_GETLBTEXTLEN CB_GETLBTEXT
CB_GETCURSEL CB_GETCOUNT CB_DIR CB_DELETESTRING CB_ADDSTRING CB_SETEDITSEL
CB_LIMITTEXT CB_GETEDITSEL WM_CTLCOLORSTATIC WM_CTLCOLORSCROLLBAR WM_CTLCOLORDLG
 WM_CTLCOLORBTN WM_CTLCOLORLISTBOX WM_CTLCOLOREDIT WM_CTLCOLORMSGBOX WM_LBTRACKPOINT
 WM_QUERYUISTATE WM_UPDATEUISTATE WM_CHANGEUISTATE WM_MENUCOMMAND WM_UNINITMENUPOPUP
 WM_MENUGETOBJECT WM_MENUDRAG WM_MENUBUTTONUP WM_ENTERIDLE WM_MENUCHAR
WM_MENUSELECT WM_SYSTIMER WM_INITMENUPOPUP WM_INITMENU WM_VSCROLL WM_HSCROLL
 WM_TIMER WM_SYSCOMMAND WM_COMMAND WM_INITDIALOG WM_IME_KEYLAST WM_IME_COMPOSITION
 WM_IME_ENDCOMPOSITION WM_IME_STARTCOMPOSITION WM_INTERIM WM_CONVERTRESULT
 WM_CONVERTREQUEST WM_WNT_CONVERTREQUESTEX WM_UNICHAR WM_SYSDEADCHAR WM_SYSCHAR
 WM_SYSKEYUP WM_SYSKEYDOWN WM_DEADCHAR WM_CHAR WM_KEYUP WM_KEYDOWN WM_INPUT
BM_SETDONTClick BM_SETIMAGE BM_GETIMAGE BM_CLICK BM_SETSTYLE BM_SETSTATE
BM_GETSTATE BM_SETCHECK BM_GETCHECK SBM_GETSCROLLBARINFO SBM_GETSCROLLINFO
SBM_SETSCROLLINFO SBM_SETRANGEREDRAW SBM_ENABLE_ARROWS SBM_GETRANGE SBM_SETRANGE
SBM_GETPOS SBM_SETPOS EM_GETIMESTATUS EM_SETIMESTATUS EM_CHARFROMPOS EM_POSFROMCHAR
EM_GETLIMITTEXT EM_GETMARGINS EM_SETMARGINS EM_GETPASSWORDCHAR EM_GETWORDBREAKPROC
EM_SETWORDBREAKPROC EM_SETREADONLY EM_GETFIRSTVISIBLELINE EM_EMPTYUNDOBUFFER
EM_SETPASSWORDCHAR EM_SETTABSTOPS EM_SETWORDBREAK EM_LINEFROMCHAR EM_FMTLINES
EM_UNDO EM_CANUNDO EM_SETLIMITTEXT EM_LIMITTEXT EM_GETLINE EM_SETFONT EM_REPLACESEL
 EM_LINELENGTH EM_GETTHUMB EM_GETHANDLE EM_SETHANDLE EM_LINEINDEX EM_GETLINECOUNT
 EM_SETMODIFY EM_GETMODIFY EM_SCROLLCARET EM_LINESCROLL EM_SCROLL EM_SETRECTNP
 EM_SETRECT EM_GETRECT EM_SETSEL EM_GETSEL WM_NCXBUTTONDOWNBLCLK WM_NCXBUTTONUP
 WM_NCXBUTTONDOWN WM_NCMBUTTONDBLCLK WM_NCMBUTTONUP WM_NCMBUTTONDOWN WM_NCRBUTTONDOWNBLCLK
 WM_NCRBUTTONUP WM_NCRBUTTONDOWN WM_NCLBUTTONDOWNBLCLK WM_NCLBUTTONUP WM_NCLBUTTONDOWN
 WM_NCMOUSEMOVE WM_SYNCPAINT WM_GETDLGCODE WM_NCACTIVATE WM_NCPAINT WM_NCHITEST
 WM_NCCALCSIZE WM_NCDESTROY WM_NCCREATE WM_SETICON WM_GETICON WM_DISPLAYCHANGE
 WM_STYLECHANGED WM_STYLECHANGING WM_CONTEXTMENU WM_NOTIFYFORMAT WM_USERCHANGED
 WM_HELP WM_TCARD WM_INPUTLANGCHANGE WM_INPUTLANGCHANGEREQUEST WM_NOTIFY
 WM_CANCELJOURNAL WM_COPYDATA WM_COPYGLOBALDATA WM_POWER WM_WINDOWPOSCHANGED
 WM_WINDOWPOSCHANGING WM_COMMNOTIFY WM_COMPACTING WM_GETOBJECT WM_COMPAREITEM
 WM_QUERYDRAGICON WM_GETHOTKEY WM_SETHOTKEY WM_GETFONT WM_SETFONT WM_CHARTOITEM
 WM_KEYTOITEM WM_DELETEITEM WM_MEASUREITEM WM_DRAWITEM WM_SPOOLERSTATUS
 WM_NEXTDLGCTL WM_ICONERASEBKGD WM_PAINTICON WM_GETMINMAXINFO WM_QUEUESYNC
 WM_CHILDACTIVATE WM_MOUSEACTIVATE WM_SETCURSOR WM_CANCELMODE WM_TIMECHANGE
 WM_FONTCHANGE WM_ACTIVATEAPP WM_DEVMODECHANGE WM_WININICHANGE WM_CTLCOLOR
 WM_SHOWWINDOW WM_ENDSESSION WM_SYSCOLORCHANGE WM_ERASEBKGD WM_QUERYOPEN
 WM_QUIT WM_QUERYENDSESSION WM_CLOSE WM_PAINT WM_GETTEXTLENGTH WM_GETTEXT
WM_SETTEXT WM_SETREDRAW WM_ENABLE WM_KILLFOCUS WM_SETFOCUS WM_ACTIVATE
WM_SIZE WM_MOVE WM_DESTROY WM_CREATE WM_NULL SRCCOPY DIB_RGB_COLORS BI_RGB
 ->bmiColors ->bmiHeader BITMAPINFO ->biClrImportant ->biClrUsed ->biYPelsPerMeter
 ->biXPelsPerMeter ->biSizeImage ->biCompression ->biBitCount ->biPlanes
 ->biHeight ->biWidth ->biSize BITMAPINFOHEADER ->rgbReserved ->rgbRed ->rgbGreen
 ->rgbBlue RGBQUAD StretchDIBits DC_PEN DC_BRUSH DEFAULT_GUI_FONT SYSTEM_FIXED_FONT
DEFAULT_PALETTE DEVICE_DEFAULT_PALETTE SYSTEM_FONT ANSI_VAR_FONT ANSI_FIXED_FONT
 OEM_FIXED_FONT BLACK_PEN WHITE_PEN NULL_BRUSH BLACK_BRUSH DKGRAY_BRUSH
 GRAY_BRUSH LTGRAY_BRUSH WHITE_BRUSH GetStockObject COLOR_WINDOW RGB CreateSolidBrush
 DeleteObject Gdi32 dpi-aware SetThreadDpiAwarenessContext VK_ALT GET_X_LPARAM
 GET_Y_LPARAM IDI_INFORMATION IDI_ERROR IDI_WARNING IDI_SHIELD IDI_WINLOGO
 IDI_asterisk IDI_exclamation IDI_question IDI_HAND IDI_APPLICATION LoadIconA
 IDC_HELP IDC_APPSTARTING IDC_HAND IDC_NO IDC_SIZEALL IDC_SIZES IDC_SIZEWE
 IDC_SIZENESW IDC_SIZENWSE IDC_ICON IDC_SIZE IDC_UPARROW IDC_CROSS IDC_WAIT
 IDC_IBEAM IDC_ARROW LoadCursorA PostQuitMessage FillRect ->rgbReserved
 ->fIncUpdate ->fRestore ->rcPaint ->fErase ->hdc PAINTSTRUCT EndPaint BeginPaint

```

GetDC PM_NOYIELD PM_REMOVE PM_NOREMOVE ->lPrivate ->pt ->time ->lParam
->wParam ->message ->hwnd MSG DispatchMessageA TranslateMessage PeekMessageA
GetMessageA ->bottom ->right ->top ->left RECT ->y ->x POINT CW_USEDEFAULT
IDI_MAIN_ICON DefaultInstance WS_TILEDWINDOW WS_POPUPWINDOW WS_OVERLAPPEDWINDOW
WS_CAPTION WS_TILED WS_ICONIC WS_CHILDWINDOW WS_GROUP WS_TABSTOP WS_POPUP
WS_CHILD WS_MINIMIZE WS_VISIBLE WS_DISABLED WS_CLIPSIBLINGS WS_CLIPCHILDREN
WS_MAXIMIZE WS_BORDER WS_DLGFRAME WS_VSCROLL WS_HSCROLL WS_SYSMENU WS_THICKFRAME
WS_MINIMIZEBOX WS_MAXIMIZEBOX WS_OVERLAPPED CreateWindowExA callback DefWindowProcA
SetForegroundWindow SW_SHOWMAXIMIZED SW_SHOWNORMAL SW_FORCEMINIMIZE SW_SHOWDEFAULT
SW_RESTORE SW_SHOWNA SW_SHWMINNOACTIVE SW_MINIMIZE SW_SHOW SW_SHOWNOACTIVATE
SW_MAXIMIZED SW_SHOWMINIMIZED SW_NORMAL SW_HIDE ShowWindow ->lpszClassName
->lpszMenuName ->hbrBackground ->hCursor ->hIcon ->hInstance ->cbWndExtra
->cbClsExtra ->lpfnWndProc ->style WINDCLASSA RegisterClassA MB_CANCELTRYCONTINUE
MB_RETRYCANCEL MB_YESNO MB_YESNOCANCEL MB_ABORTRETRYIGNORE MB_OKCANCEL
MB_OK MessageBoxA User32 win-key win-key? raw-key win-type init-console
console-mode stderr stdout stdin console-started FlushConsoleInputBuffer
SetConsoleMode GetConsoleMode GetStdHandle ExitProcess AllocConsole
ENABLE_LVB_GRID_WORLDWIDE
DISABLE\_NEWLINE\_AUTO\_RETURN ENABLE_VIRTUAL_TERMINAL_PROCESSING
ENABLE_WRAP_AT_EOL_OUTPUT
ENABLE_PROCESSED_OUTPUT ENABLE_VIRTUAL_TERMINAL_INPUT ENABLE_QUICK_EDIT_MODE
ENABLE\_INSERT\_MODE ENABLE_MOUSE_INPUT ENABLE_WINDOW_INPUT ENABLE_ECHO_INPUT
ENABLE_LINE_INPUT ENABLE\_PROCESSED\_INPUT STD_ERROR_HANDLE STD_OUTPUT_HANDLE
STD_INPUT_HANDLE invalid?ior dONULL wargs-convert wz>sz wargv
wargc CommandLineToArgvW Shell32 GetModuleHandleA GetCommandLineW GetLastError
WaitForSingleObject GetTickCount Sleep ExitProcess Kernel32 contains? dll
sofunc GetProcAddress LoadLibraryA WindowProcShim SetupCtrlBreakHandler
windows-builtins calls

```

Ressources

English

- **ESP32forth** page maintained by Brad NELSON, the creator of ESP32forth. You will find all versions there (ESP32, Windows, Web, Linux...)
<https://esp32forth.appspot.com/ESP32forth.html>

French

- **eForth** two-language site (French, English) with lots of examples
<https://eforthwin.arduino-forth.com/>

GitHub

- **Ueforth** resources maintained by Brad NELSON. Contains all Forth and C source files for ESP32forth and ueForth Windows, Linux and web.
<https://github.com/flagxor/ueforth>
- **eForth Windows** source codes and documentation for eForth Windows. Resources maintained by Marc PETREMANN
<https://github.com/MPETREMANN11/eForth-Windows>
- **eForth SDL2 project** for eForth Windows
<https://github.com/MPETREMANN11/SDL2-eForth-windows>

Facebook

- **Eforth** group for eForth Windows
<https://www.facebook.com/groups/785868495783000>

Index lexical

c!.....	35	graphics.....	42	32
c@.....	35	local variables.....	37	32
constant.....	35	memory.....	35	.s.....	31
dll.....	41	Netbeans.....	19	(.....	27
drop.....	34	return stack.....	34	{.....	37
dump.....	30	see.....	30	}.....	37
dup.....	34	to.....	38	@.....	35
FORTH.....	44	value.....	36	\.....	27
GIT.....	19	variable.....	35	+to.....	38