The great book
for eFORTH Windows

version 1.7 - 1 décembre 2024
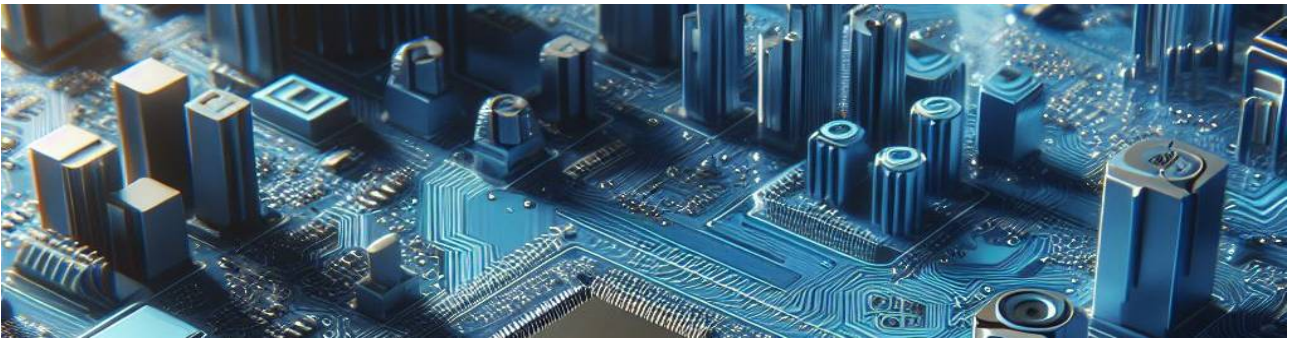
Author

- Marc PETREMANN

Contents

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 Loisitech - 1986)

- TURBO-Forth, learning manual (ed Rem CORP - 1990)

- TURBO-Forth, reference guide (ed Rem CORP - 1991)
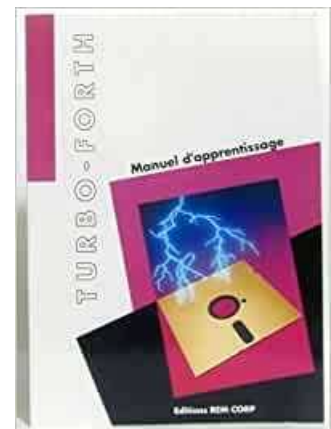
Programming in FORTH language has always been a hobby until 1992 when the manager of a company working as a subcontractor for the automobile industry contacted me. They had a problem with software development in C language. They needed to control an industrial automaton.

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 debordement 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:

```
_O_ __/ \ turns on LED
```

Yes! The sequence **_O_** **__/** 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 µEforth 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 µEforth 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*
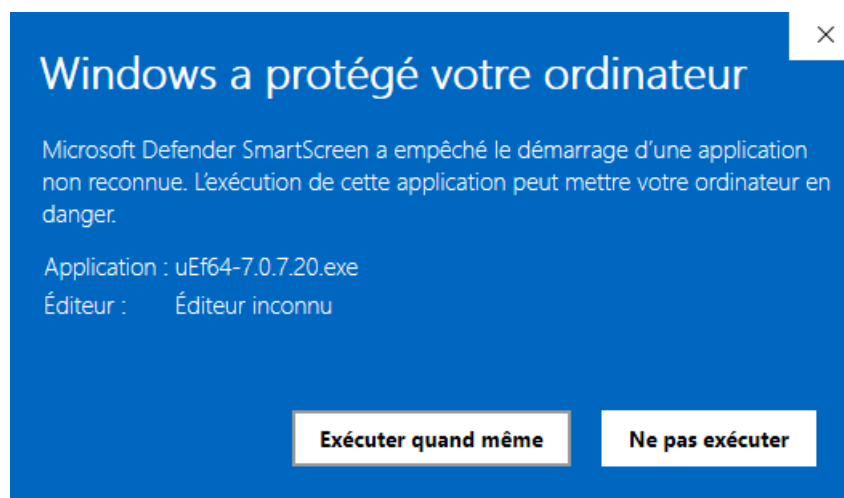
Figure 1: skip windows alert message

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 µEforth window:
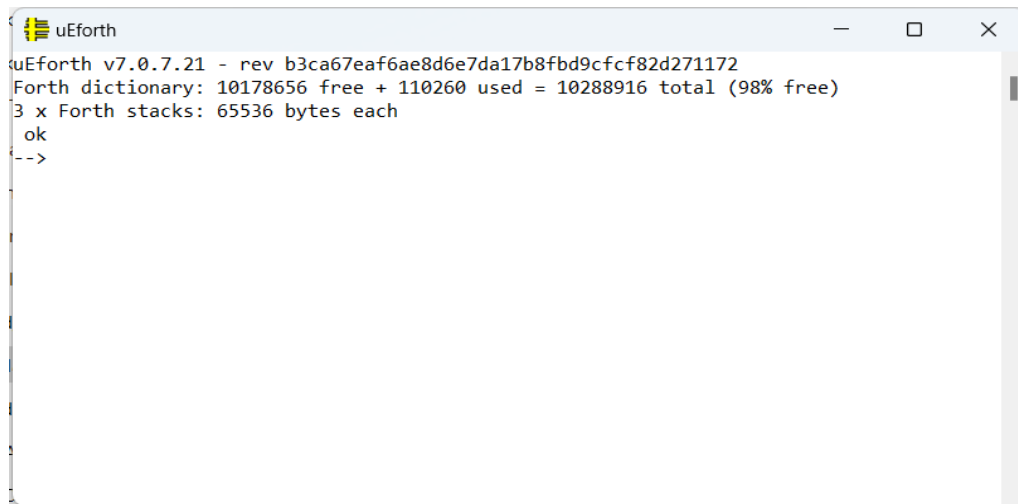
Figure 2: The µEforth window on Windows

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 µeForth logo, right-click and select *Properties:*
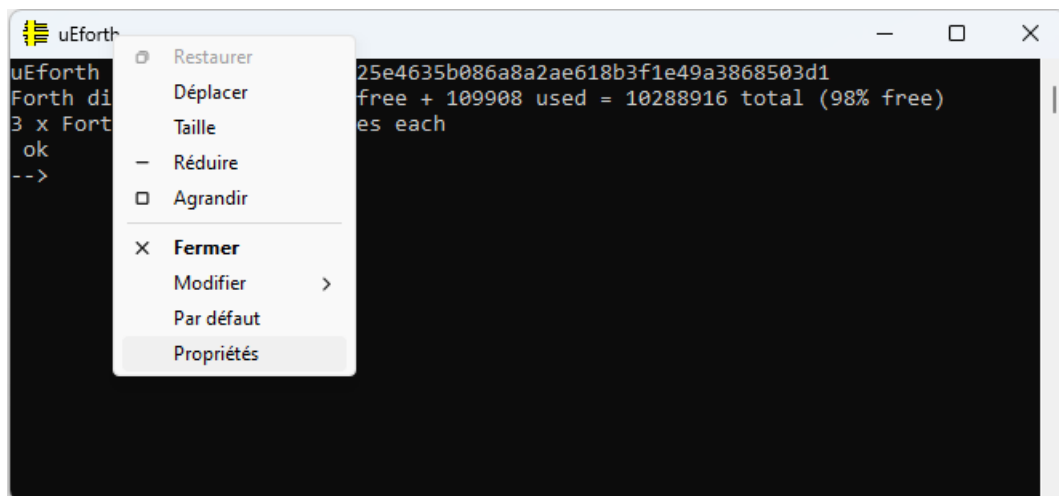
Figure 3: Selecting display properties

In Properties, click the *Colors tab* :
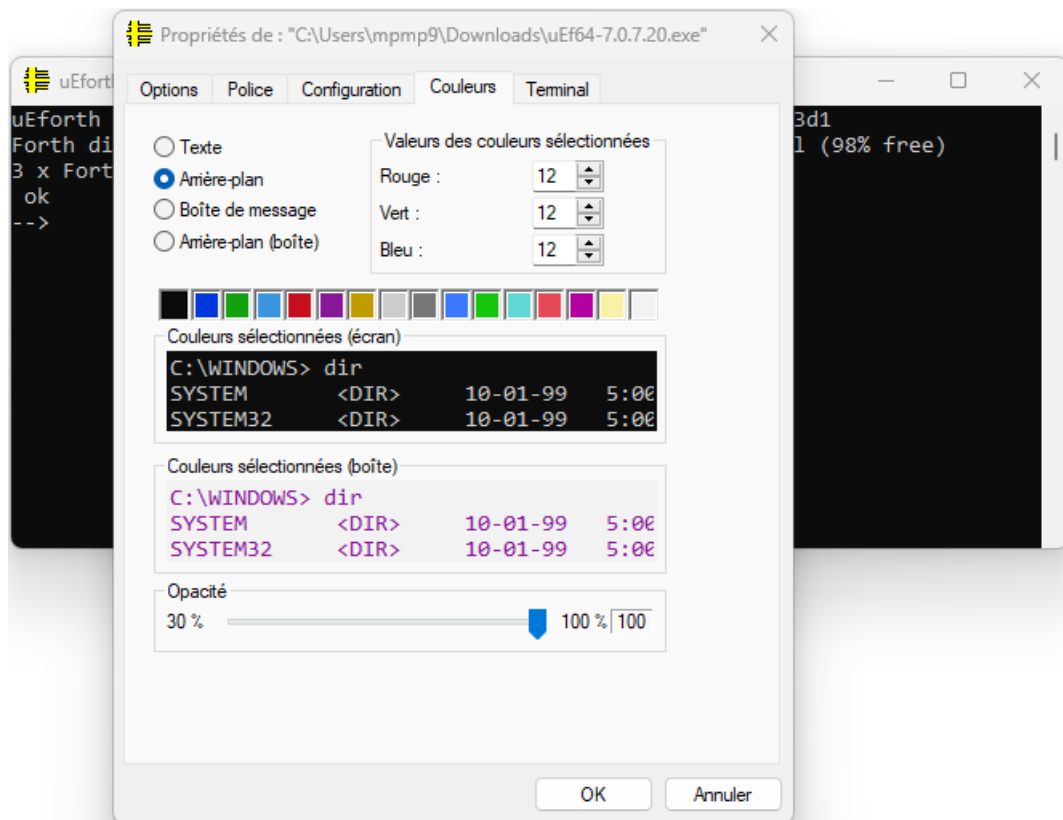
Figure 4: Choice of display colors

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:
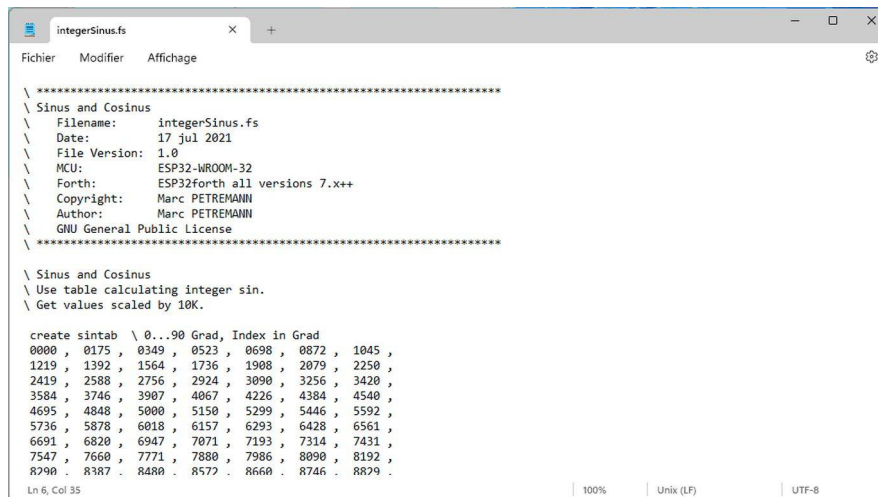
Figure 5: editing with edit under windows 11

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** :
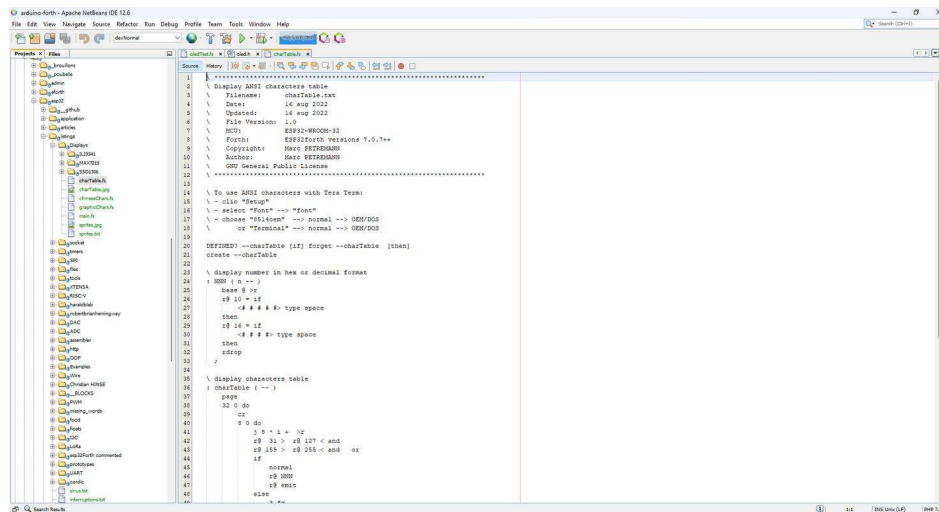
Figure 6: editing with Netbeans

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
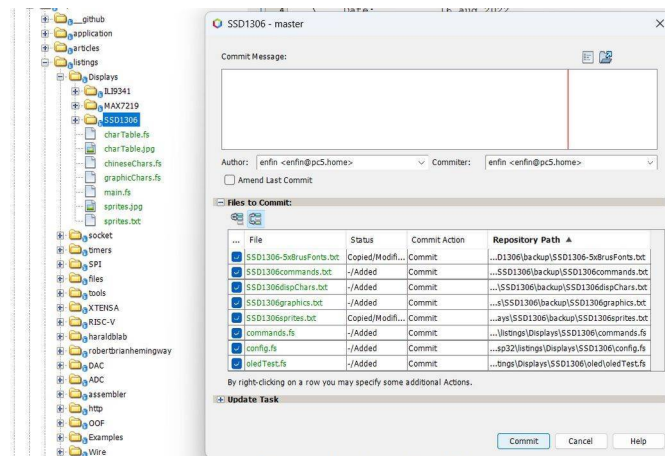
Figure 7: GIT operation in Netbeans

With NetBeans you can define a development branch for a complex project. Here we create a new branch:
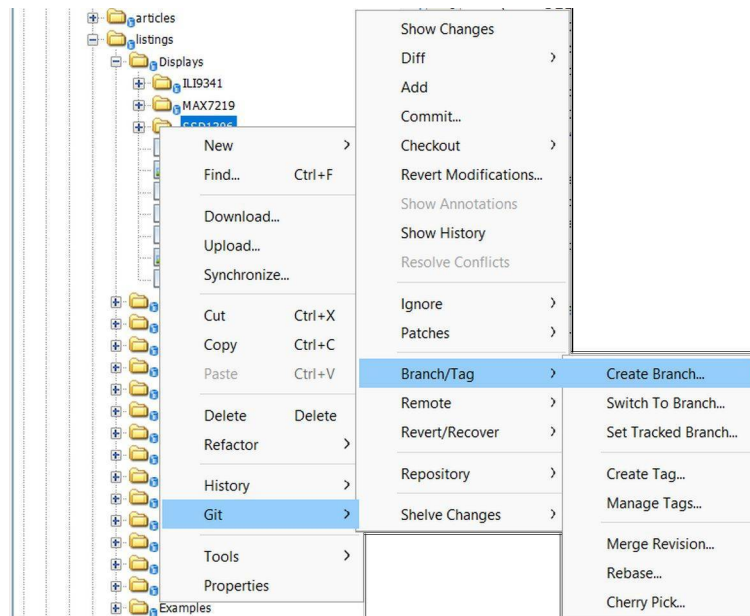
Figure 8: creating a branch on a project

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

GitHub [2]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.
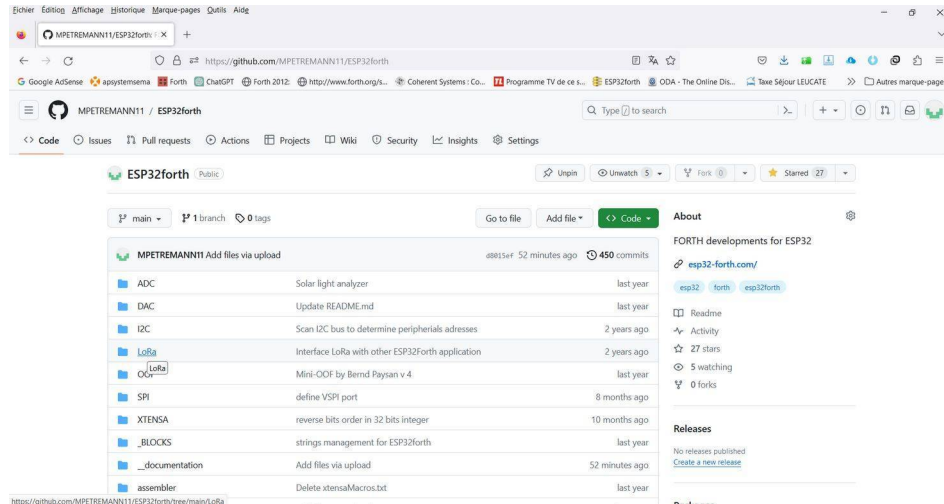
Figure 9: storing files on Github

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:
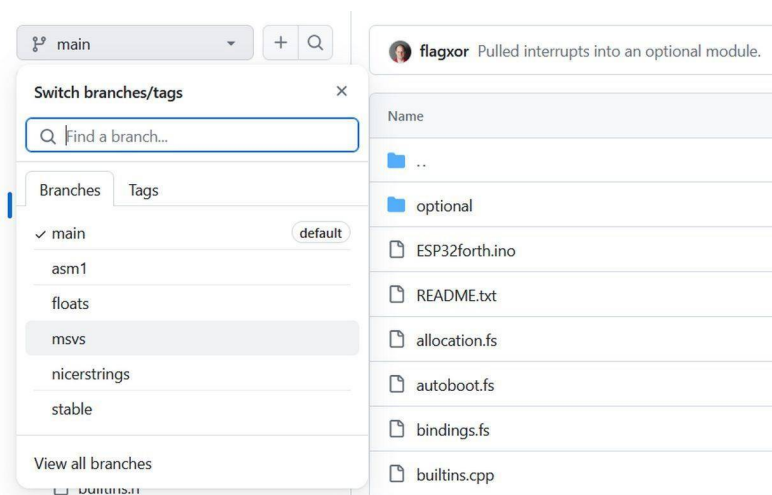
Figure 10: access to a project branch

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 [4] to manage and present code written in FORTH language in a structured way. At worst, you use an ASCII text editor, at best a real IDE and text files:

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

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

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

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

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

```
: socketError ( -- )
   errno dup
   case
      2 of    ." No such file "                endof
      5 of    ." I/O error "                   endof
```

```
      9 of     ." Bad file number "              endof
     22 of     ." Invalid argument "             endof
   endcase
   . quit
 ;
```

Comments

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

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

- the word **(** 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;

- **bl (--32)** means that this word does not process any input data and leaves the decimal value 32 on the data stack;

There is no limitation on the amount of data processed before or after 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 dictionnary pointer on stack
       0 c,    \ initial lenght data is 0
```

```
    does>
        dup 1+ swap c@
        \ send a data array to SSD1306 connected via I2C bus
        sendDatasToSSD1306
  ;
```

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

- on the left, the action part when the definition word is executed, prefixed by **comp:**

- on the right the action part of the word that will be defined, prefixed 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 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 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 ( øC --- øF) \ 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:
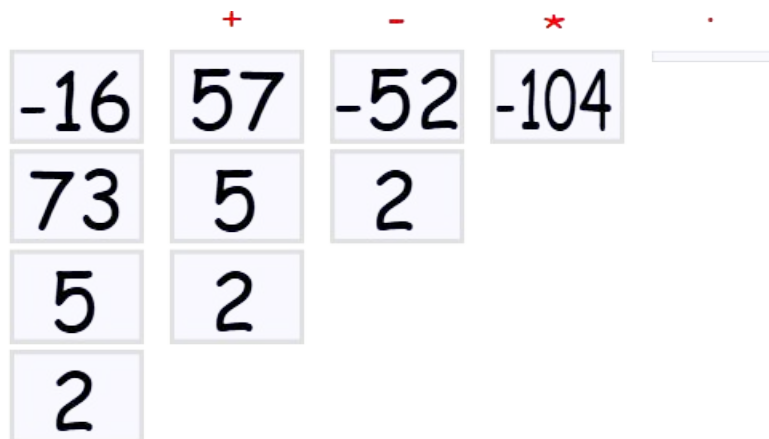
After the two lines, the console displays:

```
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.

	drop	swap	rot	dup
-16	73	5	2	2
73	5	73	5	2
5	2	2	73	5
2				73

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
\ 0START_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 * { 0START_ADDR } \ calc. addr for loop start
    16 / 1+ { LINES }
    base @ { myBASE }               \ save current base
    hex
    \ outer loop
    LINES 0 do
        0START_ADDR i 16 * +        \ calc start address for current line
        cr <# # # # #  [char] - hold # # # # #> type
        space space     \ and display address
        \ first inner loop, display bytes
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            ca@ <# # # #> type space \ display byte in format: NN
        loop
        space
        \ second inner loop, display chars
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            \ display char if code in interval 32-127
            ca@     dup 32 < over 127 > or
            if      drop [char] . emit
            else    emit
            then
        loop
    loop
    myBASE base !               \ restore current base
    cr cr
  ;
```

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 ;
```

Data Structures for eForth Windows

Preamble

eForth Windows is a 64-bit version of the FORTH language. Those who have been using FORTH since its inception have programmed with either 16-bit or 32-bit versions. This data size is determined by the size of the elements placed on the data stack. To know the size in bytes of the elements, execute the word `cell` . Executing this word for eForth:

```
cell .  \ displays 8
```

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

With a 16-bit FORTH version, `cell` will stack the value 2. Similarly, if you are using a 32-bit version, `cell` will stack the value 4.

Tables in FORTH

Let's start with fairly simple structures: arrays. We will only cover one- and two-dimensional arrays.

One-dimensional data table

This is the simplest type of array. To create an array of this type, we use the word `create` followed by the name of the array to be created:

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

In this table, we store 6 values: 34, 37....12. To retrieve a value, simply use the word @ by incrementing the address stacked by `temperatures` with the desired offset:

```
temperatures           \ stack addr
    0 cell *           \ calculate offset 0
    +                  \ add offset to addr
    @ .                \ displays 34

temperatures           \ stack addr
    1 cell *           \ calculate offset 1
    +                  \ add offset to addr
    @ .                \ displays 37
```

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

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

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

Table definition words

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

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

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

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

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

Reading and writing in a table

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

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

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

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

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

All you need to do is reserve a memory space that is the product of the X and Y dimensions of the array to be used. Now let's see how to handle this two-dimensional array:

```
: xySCRaddr { x y -- addr }
    SCR_WIDTH y *
    x + mySCREEN +
  ;
: SCR@ ( x y -- c )
    xySCRaddr c@
  ;
: SCR! ( c x y -- )
    xySCRaddr c!
  ;
char X 15 5 SCR!    \ store char X at col 15 line 5
15 5 SCR@ emit      \ display X
```

Practical example of screen management

Here's how to display the available character table:

```
: tableChars ( -- )
    base @ >r  hex
    128 32 do
      16 0 do
          j i + dup . space emit space space
      loop
      cr
    16 +loop
    256 160 do
      16 0 do
          j i + dup . space emit space space
      loop
      cr
    16 +loop
    cr
    r> base !
  ;
tableChars
```

Here is the result of running **tableChars** :

```
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 ⌂
A0 á  A1 í  A2 ó  A3 ú  A4 ñ  A5 Ñ  A6 ª  A7 º  A8 ¿  A9 ®  AA ¬  AB ½  AC ¼  AD ¡  AE «  AF »
B0 ░  B1 ▒  B2 ▓  B3 │  B4 ┤  B5 Á  B6 Â  B7 À  B8 ©  B9 ╣  BA ║  BB ╗  BC ╝  BD ¢  BE ¥  BF ┐
C0 └  C1 ┴  C2 ┬  C3 ├  C4 ─  C5 ┼  C6 ã  C7 Ã  C8 ╚  C9 ╔  CA ╩  CB ╦  CC ╠  CD ═  CE ╬  CF ¤
D0 ð  D1 Ð  D2 Ê  D3 Ë  D4 È  D5 ı  D6 Í  D7 Î  D8 Ï  D9 ┘  DA ┌  DB █  DC ▄  DD ¦  DE Ì  DF ▀
E0 Ó  E1 ß  E2 Ô  E3 Ò  E4 õ  E5 Õ  E6 µ  E7 þ  E8 þ  E9 Ú  EA Û  EB Ù  EC ý  ED Ý  EE ¯  EF ´
```

These characters are from the MS-DOS ASCII set. Some of these characters are semi-graphical. Here is a very simple insertion of one of these characters into our virtual screen:

```
$db dup 5 2 SCR!      6 2 SCR!
$b2 dup 7 3 SCR!      8 3 SCR!
$b1 dup 9 4 SCR!     10 4 SCR!
```

Now let's see how to display the contents of our virtual screen. If we consider each line of the virtual screen as an alphanumeric string, we just need to define this word to display one of the lines of our virtual screen:

```
: dispLine { numLine -- }
    SCR_WIDTH numLine *
    mySCREEN + SCR_WIDTH type
  ;
```

By the way, we will create a definition allowing the same character to be displayed n times:

```
: nEmit ( c n -- )
    for
        aft dup emit then
    next
    drop
  ;
```

And now, we define the word that allows us to display the contents of our virtual screen. To clearly see the contents of this virtual screen, we surround it with special characters:

```
: dispScreen
    0 0 at-xy
    \ display upper border
    $da emit    $c4 SCR_WIDTH nEmit    $bf emit    cr
    \ display content virtual screen
    SCR_HEIGHT 0 do
        $b3 emit    i dispLine          $b3 emit    cr
    loop
    \ display bottom border
    $c0 emit    $c4 SCR_WIDTH nEmit    $d9 emit    cr
  ; R_WIDTH nEmit $d9 emit cr
;
```
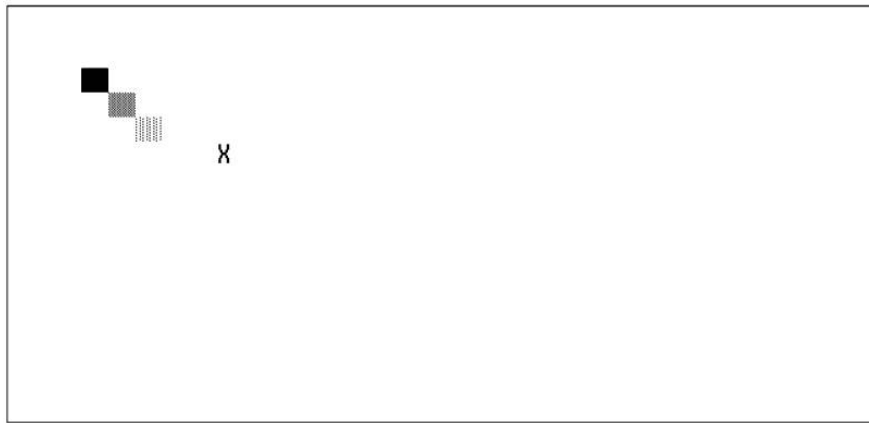
Running our **dispScreen word** displays this:

Figure 13: dispScreen execution

In our virtual screen example, we show that managing a two-dimensional array has a concrete application. Our virtual screen is writable and readable. Here, we display our virtual screen in the terminal window.

Management of complex structures

structures vocabulary . The content of this vocabulary allows you to define complex data structures.

Here is a trivial example of structure:

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

Here we define the YMDHMS structure. This structure manages the accessors **->YMDHMS-year ->YMDHMS-month ->YMDHMS-day ->YMDHMS-hour ->YMDHMS-min** and **->YMDHMS-sec** .

YMDHMS word is only used to initialize accessors. Here is how these accessors are used:

```
create DateTime
    YMDHMS allot

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

: .date ( date -- )
    >r
```

```
   ."  YEAR: " r@ ->YMDHMS-year    @ . cr
   ." MONTH: " r@ ->YMDHMS-month   @ . cr
   ."   DAY: " r@ ->YMDHMS-day     @ . cr
   ."    HH: " r@ ->YMDHMS-hour    @ . cr
   ."    MM: " r@ ->YMDHMS-min     @ . cr
   ."    SS: " r@ ->YMDHMS-sec     @ . cr
   r> drop
 ;

DateTime .date
```

Rules for naming structures and accessors

A structure is defined by the word `struct` . The name chosen depends on the context of use. It should not be too long, to remain readable. Here, a structure defining a color in the SDL library:

```
struct SDL_Color ( -- n )
```

This structure is named `SDL_Color` . This name was chosen because this structure has the same name in the C language library.

Here is the definition, in Forth, of the accessors corresponding to this `SDL_Color` structure :

```
struct SDL_Color ( -- n )
    i8 field ->Color-r
    i8 field ->Color-g
    i8 field ->Color-b
    i8 field ->Color-a
```

Each definition begins with a word indicating the size of the data field in the structure, here `i8` .

This word `i8` is followed by the word `field` which will name the accessor, `->Color-r` for example.

You can choose an accessor name as you wish, for example:

```
i8 field red-color
```

Or

```
i8 field colorRed
```

But these names quickly end up making a program difficult to decipher. For this reason, it is desirable to precede an accessor name with `->` . We follow it with the name of the structure, here `Color` , followed by a hyphen and a discriminating name, here `r` :

```
i8 field ->Color-r
```

Another example:

```
struct SDL_GenericEvent
    i32 field ->GenericEvent-type
    i32 field ->GenericEvent-timestamp
```

Here we define the `SDL_GenericEvent` structure and two 32-bit accessors: `->GenericEvent-type` and `->GenericEvent-timestamp` .

Subsequently, if we find the `->GenericEvent-type accessor` in a program, we will immediately know that it is an accessor associated with the `GenericEvent structure` .

Choosing the size of fields in a structure

The size of a field in a structure is defined by one of these words:

- `ptr` to point to data on the size of a cell, 8 bytes for eForth Windows

- `i64` to point to 8-byte data (64b its)

- `i32` to point to a 4-byte (32-bit) data

- `i16` to point to a 2-byte (16-bit) data

- `i8` to point to a data on 1 byte (8 bits)

For eForth Windows, the words `ptr` and `i64` have the same action.

You cannot manage fields of variable size, for a character string for example.

If we need to define a field of a special size, we will define this type like this:

```
structures definition
    10 10 typer phoneNum
     5  5 typer zipCode
```

We can also do without it by using the data size directly. Example:

```
structures
struct City
     5  field ->City-zip
    64  field ->City-name
```

If we run `City` , this word will stack the total size of the structure, here the value 69. We will use this value to reserve the number of characters required for data:

```
create BORDEAUX
    City allot
```

We are not going to go into the details of managing the fields of our `City` structure.

As for the fields defined by `i8` to `i64` , you cannot use the words `@` and `! alone` to read and write numeric values in these fields.

Here are the words to access the data based on its size:

	i8	i16	i32	i64
fetch	C@	UW@	UL@	@
store	C!	W!	L!	!

Figure 14: memory access word depending on field size

We defined the word `DateTime` which is a simple table of 6 consecutive 64-bit cells. Access to each of the cells is done through the corresponding accessor. We can redefine the allocated space of our YMDHMS structure using `i8` and `i16` :

```
structures
struct cYMDHMS
    i16 field ->cYMDHMS-year
    i8  field ->cYMDHMS-month
    i8  field ->cYMDHMS-day
    i8  field ->cYMDHMS-hour
    i8  field ->cYMDHMS-min
    i8  field ->cYMDHMS-sec

create cDateTime
    cYMDHMS allot

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

It is advisable to factor out the use of accessors in a global definition:

```
: date!  { year month day hour min sec addr -- }
    year     addr ->cYMDHMS-year  w!
    month    addr ->cYMDHMS-month c!
    day      addr ->cYMDHMS-day    c!
    hour     addr ->cYMDHMS-hour   c!
    min      addr ->cYMDHMS-min    c!
    sec      addr ->cYMDHMS-sec    c!
  ;
2024 11 09 18 25 40 cDateTime date!
```

With `date!` , we no longer care which fields are one or two bytes in the `cYMDHMS` structure.

If we need to change the size of a field, only the `date!` definition will need to be changed.

Here's how to read data into `cDateTime` :

```
: .date { date -- }
    ."  YEAR: " date ->cYMDHMS-year    uw@ . cr
    ." MONTH: " date ->cYMDHMS-month   c@ . cr
    ."   DAY: " date ->cYMDHMS-day     c@ . cr
```

```
    ."    HH: " date ->cYMDHMS-hour    c@ . cr
    ."    MM: " date ->cYMDHMS-min     c@ . cr
    ."    SS: " date ->cYMDHMS-sec     c@ . cr
  ;
cDateTime .date    \ display:
\   YEAR: 2024
\ MONTH: 11
\   DAY: 9
\    HH: 18
\    MM: 25
\    SS: 40
```

Definition of sprites

We previously defined a virtual screen as a two-dimensional array. The dimensions of this array are defined by two constants. Reminder of the definition of this virtual screen:

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

The disadvantage, with this programming method, is that the dimensions are defined in constants, therefore outside the table. It would be more interesting to embed the dimensions of the table in the table. To do this, we will define a structure adapted to this case:

```
structures
struct cARRAY
    i8  field ->cARRAY-width
    i8  field ->cARRAY-height
    i8  field ->cARRAY-content

: cArray-size@  { addr -- datas-size }
    addr ->cARRAY-width  c@
    addr ->cARRAY-height c@ *
  ;

create myVscreen     \ define a screen 8x32 bytes
    32 c,            \ compile width
    08 c,            \ compile height
    myVscreen cArray-size@ allot
```

To define a software sprite, we will very simply share this definition:

```
structures
struct cARRAY
    i8  field ->cARRAY-width
    i8  field ->cARRAY-height
    i8  field ->cARRAY-content

: cArray-width@  { addr -- width }
    addr ->cARRAY-width  c@
  ;
```

```
: cArray-height@  { addr -- height }
    addr ->cARRAY-height  c@
  ;

: cArray-size@  { addr -- datas-size }
    addr cArray-width@
    addr cArray-height@ *
  ;
```

Here's how to define a 5 x 7 byte sprite:

```
create char3
    5 c,  7 c,     \ compile width and height
    $20 c,  $db c,  $db c,  $db c,  $20 c,
    $db c,  $20 c,  $20 c,  $20 c,  $db c,
    $20 c,  $20 c,  $20 c,  $20 c,  $db c,
    $20 c,  $db c,  $db c,  $db c,  $20 c,
    $20 c,  $20 c,  $20 c,  $20 c,  $db c,
    $db c,  $20 c,  $20 c,  $20 c,  $db c,
    $20 c,  $db c,  $db c,  $db c,  $20 c,
```

To display the sprite, from an xy position in the terminal window, a simple loop is enough:

```
: .sprite { xpos ypos sprite-addr -- }
    sprite-addr cArray-height@ 0 do
        xpos ypos at-xy
        sprite-addr cArray-width@ i *       \ calculate offset in sprite datas
        sprite-addr ->cARRAY-content +      \ calculate real address for line n in
sprite datas
        sprite-addr cArray-width@ type      \ display line
        1 +to ypos                  \ increment y position
    loop
  ;

0 constant blackColor
1 constant redColor
4 constant blueColor
10 02 char3 .sprite
redColor fg
16 02 char3 .sprite
blueColor fg
22 02 char3 .sprite
blackColor fg
cr cr
```

Result of displaying our sprite:

Figure 15: sprite display

That's it. That's all.

The structures in detail

In the previous chapter, we discussed structures built with `struct` and `field` . We will analyze in more detail some subtleties on the manipulation of structures, in particular on data access, in reading and writing.

Field sizes

Windows makes extensive use of structures to exchange data between processes. Here is an example of a structure shared with the Windows API:

```
struct POINT
  i32 field ->x
  i32 field ->y

struct RECT
  i32 field ->left
  i32 field ->top
  i32 field ->right
  i32 field ->bottom

struct MSG
    ptr field ->hwnd
    i32 field ->message
    i16 field ->wParam
    i32 field ->lParam
    i32 field ->time
  POINT field ->pt
    i32 field ->lPrivate
```

This code is extracted from the eForth Windows source files. Here, three structures are defined: **POINT RECT** and **MSG** .

As a reminder, before each `field` , we have a word which indicates the size of the field in the structure:

- `i8` for an 8-bit field, i.e. 1 byte;

- `i16` for a 16-bit field, therefore 2 bytes;

- `i32` for a 32-bit field, so 4 bytes;

- `i64` and `ptr` for a 64-bit field, so 8 bytes;

The word `ptr` depends on the FORTH version. Here, in eForth Windows, it is indeed a 64-bit field. But on a 32-bit system, like ESP32forth, it will be a 32-bit field.

In the `MSG` structure, we find fields of different sizes:

- **`ptr field ->hwnd`** denotes a 64-bit field

- **`i32 field ->message`** denotes a 32-bit field

- **`i16 field ->wParam`** denotes a 16-bit field

For each field size, you must use the words **`@`** or **`!`** appropriate to the size of the target field.

Accessing data in the fields of a structure

As we understand, you need a word adapted to each data size:

- **8 bits** : access with the words **`C@`** and **`C!`**

- **16 bits** : access with the words **`SW@`** or **`UW@`** and **`W!`**

- **32 bits** : access with the words **`SL@`** or **`UL@`** and **`L!`**

- **64 bits** : access with the words **`@`** and **`!`**

Example :

```
create iconePos
    POINT allot
-5 iconePos ->x W!
 3 iconePos ->y W!
```

The value -5 is stored in the first 16-bit field, the value 3 in the second 16-bit field. To retrieve these values, proceed as follows:

```
iconPos ->x SW@
iconPos ->y SW@
```

To retrieve a 16-bit value, there are two options. Use **`SW@`** or **`UW@`** . The word **`SW@`** retrieves the 16-bit value, as a signed value.

`UW@` had been used , the value -5 would have been replaced by 65531 which is its 16-bit unsigned equivalent.

The mechanism is similar for 32-bit values with **`SL@`** and **`UL@`** .

Managing structure accessors

If you need to access data from a structure in multiple places in your program, the best solution is to define a single access word and then use that word exclusively. Example:

```
: setPoint  ( x y addr -- )
    >r
    r@ ->y W!
    r> ->x W!
  ;
```

```
: getPoint ( addr -- x y )
    >r
    r@ ->x SW@
    r> ->y SW@
  ;

-15 37 iconePos setPoint
iconePos getPoint  \ push -15 37 on stack
```

In the rest of the program, using **setPoint** and **getPoint** will make it much easier to manipulate the data in a **POINT** structure.

Nested structures

The Windows API exploits nested structures:

```
struct MSG
    ptr field ->hwnd
    i32 field ->message
    i16 field ->wParam
    i32 field ->lParam
    i32 field ->time
  POINT field ->pt
    i32 field ->lPrivate
```

MSG structure , a **->pt** field is defined . This field has the data size of a **POINT** structure. As a reminder, using a structure name simply returns the data size of this structure. Here, the word **POINT** returns the value 8, a value that corresponds to the size of the two 32-bit fields defined in this **POINT** structure.

Let's see a simple example of a nested structure. Let's use **POINT** to define a **TRIANGLE** structure:

```
sstructures
struct TRIANGLE
    POINT field ->triangle-pt1
    POINT field ->triangle-pt2
    POINT field ->triangle-pt3

: setTriangle ( x1 y1 x2 y2 x3 y3 addr -- )
    >r
    r@ ->triangle-pt3 setPoint
    r@ ->triangle-pt2 setPoint
    r> ->triangle-pt1 setPoint
  ;

create triAng01
    TRIANGLE ALLOT

10 10  70 10   40 70 triAng01 setTriangle
```

In some cases, it's a little less elegant. Here's how to access the data of a **TRIANGLE** structure without using **setPoint**:

```
: setTriangle ( x1 y1 x2 y2 x3 y3 addr -- )
    >r
    r@ ->triangle-pt3 ->y W!
    r@ ->triangle-pt3 ->x W!
    r@ ->triangle-pt2 ->y W!
    r@ ->triangle-pt2 ->x W!
    r@ ->triangle-pt1 ->y W!
    r> ->triangle-pt1 ->x W!
  ;
```

In summary, structures should be handled with care. Reserve them for interfaces with Windows API software bindings.

Real Numbers with eForth Windows

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

This is not surprising. By default, eForth Windows only uses 64-bit integers via the data stack. Integers offer some advantages:

- speed of processing;

- result of calculations without risk of drift in the event of iterations;

- are suitable for almost any situation.

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

But integers also have limits:

- impossible results for simple division calculations, like our 1/3 example;

- requires complex manipulations to apply physics formulas.

Since version 7.0.6.5, eForth Windows includes operators dealing with real numbers.

Real numbers are also called floating point numbers.

Reals with eForth Windows

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

```
3               \ stacks 3 on the data stack
3rd             \ stacks 3 on the real number stack
5.21e f.        \ displays 5.210000
```

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

Precision of real numbers with eForth Windows

`set-precision` word is used to specify the number of decimal places to display after the decimal point. Let's see this with the constant `pi` :

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

The limiting precision for processing real numbers with eForth Windows is six decimal places:

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

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

Real constants and variables

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

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

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

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

WARNING: All real numbers are pushed through the **real number stack** . In the case of a real variable, only the address pointing to the real value is pushed through the data stack.

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

SF@ word stacks the actual value pointed to by its memory address.

Arithmetic operators on real numbers

eForth Windows has four arithmetic operators **F+ F- F* F/** :

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

eForth Windows also has these words:

- **1/F** calculates the reciprocal of a real number;

- **fsqrt** calculates the square root of a real number.

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

Mathematical operators on real numbers

eForth Windows has several mathematical operators:

- **F**** raises a real r_val to the power r_exp

- **FATAN2** calculates the angle in radians from the tangent.

- **FCOS** (r1 -- r2) Calculates the cosine of an angle expressed in radians.

- **FEXP** (ln-r -- r) calculates the real corresponding to e EXP r

- **FLN** (r -- ln-r) calculates the natural logarithm of a real number.

- **FSIN** (r1 -- r2) calculates the sine of an angle expressed in radians.

- **FSINCOS** (r1 -- rcos rsin) calculates the cosine and sine of an angle expressed in radians.

Some examples:

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

4.605170e FEXP F.    \ display 100.000018

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

Logical operators on real numbers

eForth Windows also allows you to perform logic tests on real numbers:

- **F0<** (r -- fl) tests whether a real number is less than zero.

- **F0=** (r -- fl) indicates true if the real is zero.

- **f<** (r1 r2 -- fl) fl is true if r1 < r2.

- **f<=** (r1 r2 -- fl) fl is true if r1 <= r2.

- **f<>** (r1 r2 -- fl) fl is true if r1 <> r2.

- **f=** (r1 r2 -- fl) fl is true if r1 = r2.

- **f>** (r1 r2 -- fl) fl is true if r1 > r2.

- **f>=** (r1 r2 -- fl) fl is true if r1 >= r2.

Integer ↔ real transformations

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

- **F>S** (r -- n) converts a real to an integer. Leaves the integer part on the data stack if the real has decimal parts.

- **S>F** (n -- r: r) converts an integer to a real number and pushes that real number onto the real number stack.

Example :

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

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

Displaying numbers and strings

Change of digital base

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

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

An even larger numeric base can be chosen, but the available symbols will fall outside the alpha-numeric set [0..9,A..Z] and may become inconsistent.

The current numeric base is controlled by a variable named **BASE** , the contents of which can be modified. Thus, to switch to binary, simply store the value **2** in **BASE** . Example:

```
2 BASE !
```

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

eForth Windows has two predefined words to select different number bases:

- **DECIMAL** to select the decimal numeric base. This is the numeric base taken by default when starting eForth Windows;

- **HEX** to select the hexadecimal number base;

- **BINARY** to select the binary numeric base.

When one of these numeric bases is selected, literal numbers will be interpreted, displayed or processed in that base. Any number previously entered in a numeric base different from the current numeric base is automatically converted to the current numeric base. Example:

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

You can define your own numeric base by defining the appropriate word or by storing this base in **BASE** . Example:

```
: SEXTAL (---)            \ selects the binary numeric base
   6 BASE ! ;
   DECIMAL 255 SEXTAL .  \ displays 1103
```

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 content BASE 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 return stack
    HEX FF 10 + .    \ operation of the previous example
    R> BASE ! ;      \ restores initial value of BASE
```

WARNING : The words **>R** and **R>** are not usable in interpreted mode. You can only use these words in a definition that will be compiled.

Defining new display formats

Forth has primitives that allow you to adapt the display of a number to any format. With eForth Windows, these primitives handle integers:

- **<#** 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. For example, to display a number expressing an amount in euros with a comma as the decimal separator:

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

Examples of execution:

```
35 .EUROS           \ displays 0.35 EUR
3575 .EUROS         \ displays 35.75 EUR
1015 3575 + .EUROS  \ displays 45.90 EUR
```

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

In execution, 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, that is, starting with the least significant digits.

The processing of a number by a display format sequence is performed according to the current numeric base. The numeric base can be changed between two digits.

Here is a more complex example demonstrating the compactness of FORTH. It involves writing a program that converts any number of seconds to the HH:MM:SS format:

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

Examples of execution:

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

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

Another example is to define a program that converts a single-precision decimal integer to binary and displays it in the format bbbb bbbb bbbb bbbb:

```
: FOUR-DIGITS ( ---)
    # # # # 32 HOLD ;
: AFB ( d ---)                  \ 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
```

Example of execution:

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

Another example, let's create a phone book where we associate one or more phone numbers with a surname. We define a word by surname:

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

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

Displaying characters and strings

The display of a character is done by the word **EMIT** :

```
65 EMIT          \ display A
```

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

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

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

```
hex jeu-ascii
 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
```

Strings are displayed in several ways. The first, usable in compilation only, displays a string delimited by the " (quote) character:

```
: TITRE ." GENERAL MENU" ;
    TITRE      \ display    GENERAL MENU
```

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

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

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

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

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

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

```
CR TITLE CR CR LINE1 TYPE CR
\ display
\ GENERAL MENU
\
\ E..Data recording
```

One or more spaces can be added at the beginning 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 eForth Windows. Here is the first attempt at defining 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 string variable is defined like this:

```
16 string strState
```

Here is how the memory space reserved for this text variable is organized:
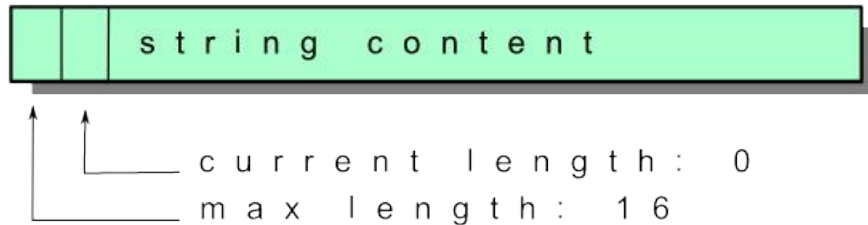
Text variable management words code

Here is the complete source code for managing text variables:

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


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

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

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

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

\ Example:
```

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

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

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

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

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

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

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

Creating an alphanumeric string is very simple:

```
64 string myNewString
```

Here we create an alphanumeric variable `myNewString` that can hold 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 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 a character to an alphanumeric variable

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

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

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

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

Here is a clever way to create an alphanumeric variable 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
  ;
```

Comparisons and connections

eForth Windows allows you to compare two numbers on the stack, using the relational operators `>` , `<` , and `=` :

```
2 3 = .      \ display: 0
2 3 >        \ display: 0
2 3 <        \ display: -1
```

These operators consume both parameters and leave a flag, to represent the Boolean result. Above, we see that "2 is equal to 3" is false (value 0), "2 is greater than 3" is also false, while "2 is less than 3" is true. The true flag has all bits set to 1, hence the representation `-1`. eForth Windows also provides the relational operators `0=` and `0<` which test whether the contents of the top of the stack are zero or negative.

Relational words are used for control. Example:

```
: test
    0= invert
    if
        cr ." Not zero!"
    then
  ;
0 test      \ no display
-14 test    \ display: Not zero!
```

The top of the stack is compared to zero. If the top of the stack is nonzero, the `if word` consumes the flag and executes all words between itself and the `then terminator` . If the top of the stack is zero, execution jumps to the word following `then` . The `cr word` emits a newline. The `else word` can be used to provide an alternate execution path, as shown here:

```
: truth
    0=
    if
        ." false"
    else
        ." true"
    then
  ;
1 truth     \ display: true
0 truth     \ display: false
```

A non-zero stack top causes words between `if` and `else` to be executed, and words between `else` and `then` to be ignored. A zero value produces the opposite behavior.

The Forth language does not handle any data types other than integers processed by the parameter stack.

Integers in the Windows eForth version are in 64-bit format. Therefore, there is no boolean type in Forth, especially with Windows eForth.

To break the linear flow of a sequence of instructions, we will use control structures whose role is to compile branches. These branches are of two types:

- conditional connections
- unconditional connections

A sequence of instructions will be repeated using a loop. These are also of two kinds:

- iterative or repetitive loops controlled by loop start and end indices
- indefinite loops

The Forth language exploits all these branching and looping possibilities through different basic structures.

Conditional Forward Branches

During the execution of a program, the result of a test or the contents of a variable may be brought to modify the execution order of the instructions. This value, called a Boolean flag, is in the true state if it is not zero, in the false state if it is zero. The Boolean flag is used by a branch instruction marking the beginning of a control structure:

- **IF** ... **THEN**

- **IF** ... **ELSE** ... **THEN**

if word is an immediate execution word and compiles a conditional branch. It can only be used in compilation. Example:


```
variable heat
: weather report ( -- )
: meteo ( -- )
   heat @ 25 >  if
       cr ."  Hot weather"
   then ;
```

and in execution:

```
15 HEAT ! WEATHER    \ does not display anything
35 HEAT ! WEATHER    \ displays "hot weather"
```

WEATHER word tests the contents of the **HEAT variable** and executes the definition part between **IF** and **THEN** if this value is greater than 25. But if it is 25 degrees or less, the **WEATHER** word does not announce anything.
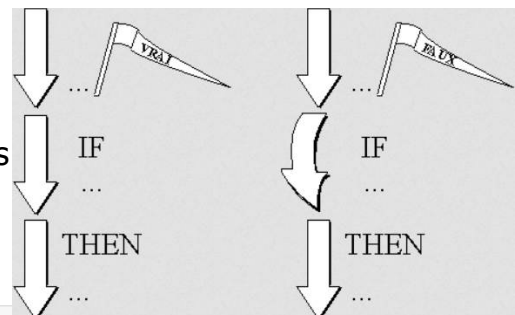
To remedy this, we can redefine it:

```
: weather ( -- )
   heat @ 25 >  if
       cr ."  Hot weather"
   else
       cr ."  Cold weather"
   then ;
```

Now:

```
15 HEAT ! Weather    \ displays cold weather
```

If the result of the test executed before **IF** delivers a false boolean flag, it is the part of the definition located between **ELSE** and **THEN** which will be executed.

A Boolean flag can be the result of various operations:

- reading a memory address,
- stacking the contents of a register at the output of execution of a definition written in machine code,
- result of a 64-bit arithmetic calculation, signed or unsigned,
- result of a logical operation (**OR** , **AND** ...) or a combination of logical operations,
- parameter stacked by the execution of a definition executed before **IF** ,

The **IF** statement tests and consumes the value at the top of the data stack. Here is another example showing whether a number is even or not:

```
: is-even?  ( n -- )
   dup 2 mod
   if         ."  is even"
   else       ."  is odd"
   then ;
```

Conditional backward branching

With the **IF** .. **THEN** and **IF** .. **ELSE** .. **THEN** type structure , only a non-repeating part of a definition can be executed. To repeat a do .. while type sequence, a new type of control structure must be used: the **BEGIN** .. **UNTIL** indefinite repeating loop .

BEGIN .. **UNTIL** loop , the part of the definition between these two words is repeated as long as the result of the test preceding **UNTIL** delivers a false boolean flag. Example:


```
: typist ( -- )
   begin
```
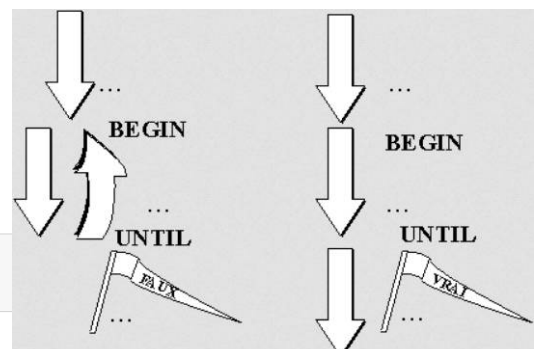
```
        key dup emit
        [char] $ =
    until ;
```

Running `typist` displays all characters typed on the keyboard. Only pressing the key marked with the '$' sign can interrupt the repetition. Pressing the <return> key returns to the beginning of the line.

If the result of the previous **UNTIL** test is still false, we can no longer exit the **BEGIN** .. **UNTIL** loop . This situation was provided for in Forth and is exploited by the **BEGIN**.. **AGAIN** loop. Using **AGAIN** is equivalent to **BEGIN** ... **0 UNTIL** . If we define `typist` by:

```
: typist ( -- )
    begin key emit again ;
```

The loop will no longer be able to be interrupted while it is running. The only way to exit will be to close the eForth Windows window.

Branching forward from an indefinite loop

With a **BEGIN** .. **UNTIL** or **BEGIN** .. **AGAIN** loop , the action is repeated at the end of the loop depending on the result of a test or unconditionally. But we can exploit a conditional forward branch from a loop whose structure is **BEGIN** .. **WHILE** .. **REPEAT** .

In this structure, the test is executed before **WHILE** . If the result is false, execution continues after **REPEAT** . If the result is true, the part of the definition between **WHILE** and **REPEAT** is executed, and then **REPEAT** performs an unconditional backward branch, i.e. returns execution to **BEGIN** .
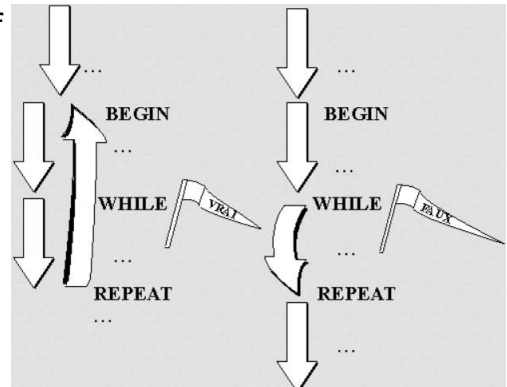
For example, let's rewrite `typist` with this new structure:

```
: typist ( -- )
    begin    key dup [char] $ <>
    while    emit
    repeat
    drop ;
```

Controlled repetition of an action

The last case of the various control structures available in Forth is the **DO** .. **LOOP** loop . This structure takes as input parameters two values which are the initial and terminal indices controlling the iteration.

Example :

```
: myLoop ( -- )
    10000 0 do loop ;
```

The word **DO** , usable only in compilation, is always preceded by two values n1 n2, where n2 is the initial index of the loop, n1 the terminal value. The terminal value is generally greater than the initial value. Example:

```
: characters ( -- )
    128 32 do i emit loop ;
```

displays all ASCII characters between 32 and 127. In this definition, the word **I** places the value of the current index of the loop on the stack. The index successively takes all values between 32 and 128, excluding 128. The incrementation and the loop exit test are performed by **LOOP** .

To increment the loop index by an amount other than one, replace the word **LOOP** with **+LOOP** and precede it with the loop increment value. Example:

```
variable table
: .table
    cr 11 table @ * table @
    do  i .  table @ +loop ;

3 table !  .table
\  display 3 6 9 12 15 ... 30

7 table ! .table
\  display 7 14 21 28 .... 70
```

The loop index increment before **+LOOP** can be negative. In this case, the first value before **DO** is less than the second. Nothing prevents us from reusing **.table** in a more general definition that will show us a real multiplication table:

```
: xTable ( -- )
    11 1 do
        i table !  .table
    loop ;
```

Don't be surprised by the reuse of the word **I** , already used in the definition of **.table** . The execution of **I** only refers to the loop in which it is defined and currently executing. On the other hand, in the case of two nested loops, to access the index of the outer loop from the nested loop, you must use the word **J** . Example:

```
: twoLoops ( -- )
    6 0 do
        cr i .
        10 0 do
            i j * .
        loop
    loop
    cr  ;
```

To interrupt the execution of a loop of type **DO** .. **LOOP** or **DO** .. **+LOOP** , you must execute the word **LEAVE** .

Multiple-choice uni-conditional structure

In some situations, it may be necessary to execute a specific action in the program depending on a specific condition, among other actions also dependent on this condition. The most frequently used control structure in this case is the **CASE** .. **OF** .. **ENDOF** .. **ENDCASE** type structure . Syntax:

```
CASE
  valeur1 OF action1 ENDOF
  valeur2 OF action2 ENDOF
  ...
  valueN OF actionN ENDOF
ENDCASE
```

The value to be tested is placed before the word **OF** . If the value on top of the stack before **CASE is executed** is the same as this value, the part of the definition between **OF** and **ENDOF** is executed, and then execution continues after **ENDCASE** . Otherwise, the next test or tests are executed until a test is valid. If no test could be verified, the part of the definition between the last **ENDOF** and **ENDCASE** is executed. Example:

```
: DAY ( n -- )
   7 mod
   case
       0 of s "SUNDAY"     endof
       1 of s "MONDAY"     endof
       2 of s "TUESDAY"    endof
       3 of s "WEDNESDAY" endof
       4 of s "THURSDAY"   endof
       5 of s "FRIDAY"     endof
       6 of s" SATURDAY"   endof
   endcase
 ;
```

Examples of execution:

```
2 DAY TYPE   \ displays TUESDAY
5 DAY TYPE   \ displays FRIDAY
7 DAY TYPE   \ displays SUNDAY
```

Recursion

Purists claim that a computer language is incomplete if it cannot handle recursion. Others claim that it can do very well without it. Forth will excite the former, because it is equipped to handle this type of situation.

Since Forth cannot explicitly refer to the word being defined, we will insert the word **RECURSE** wherever we want to call the definition being compiled. Example:

```
: factor  ( n – FACTn )
   dup 1 – dup 1 >
   if
       recurse
   then
   * ;
```

Decompiling **factor** , performed by typing **see factor** , highlights the substitution of **recurse** by the execution address of **factor** . Examples of **factor** execution :

```
2 factor .    \ displays 2
3 factor .    \ displays 6
4 factor .    \ displays 24
```

Recursion has its limits: the capacity of the data and return stacks. Poorly controlled recursion saturates the data or return stack and crashes the system.

Logical tests

Tests can be performed on comparisons of 64-bit numeric quantities only. The result of a test is always a Boolean flag, represented by a 64-bit integer:

- 0 for **false**

- -1 or any other non-zero value for **true**

A comparison test acts identically to an arithmetic operator. They are dyadic operators:

```
= < > <= >= <>
U< U> U>= U<=
```

Examples:

```
2 5 = .   \ displays 0
3 8 < .   \ displays -1
```

Comparison tests against zero are already defined. They are monadic operators:

```
0= 0< 0> 0<> 0<= 0>=
```

Example:

```
12 0> .   \ displays -1
```

The results of these tests can be combined into a single expression using the following logical operators:

- **OR AND XOR** combine two logical values

- **INVERT** inverts the state of a logical value

Logical operators operate bitwise on two unsigned 64-bit values.

Vocabularies with eForth Windows

In Forth, the notion of procedure and function does not exist. Forth instructions are called WORDS. Like a traditional language, Forth organizes the words that compose it into VOCABULARIES, a set of words having a common feature.

Programming in Forth consists of enriching an existing vocabulary, or defining a new one, relative to the application being developed.

List of vocabularies

A vocabulary is an ordered list of words, searched from most recently created to least recently created. The search order is a stack of vocabularies. Executing the name of a vocabulary replaces the top of the search order stack with that vocabulary.

To see the list of different vocabularies available in eForth Windows, we will use the word `voclist` :

```
--> internals voclist        \ display
internals
graphics
ansi
editor
streams
tasks
windows
structures
recognizers
internalized
internals
FORTH
```

This list is not limited. Additional vocabularies may appear if some extensions are compiled.

The main vocabulary is called `FORTH`. All other vocabularies are attached to the `FORTH vocabulary`.

Essential vocabularies

Here is the list of the main vocabularies available in eForth Windows:

- **ansi** display management in an ANSI terminal;

- **editor** provides access to editing commands for block-type files;

- **s structures** management of complex structures;

- **windows** windows environment management

List of the contents of a vocabulary

To see the contents of a vocabulary, use the word `vlist` after having previously selected the appropriate vocabulary:

```
graphics vlist
```

Selects the `graphics` vocabulary and displays its contents:

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

Selecting a vocabulary gives access to the words defined in that vocabulary.

For example, the word `voclist` is not accessible without first invoking the `internals` vocabulary.

The same word can be defined in two different vocabularies and have two different actions.

Using words from a vocabulary

To compile a word defined in a vocabulary other than Forth, there are two solutions. The first solution is to simply call this vocabulary before defining the word that will use words from this vocabulary.

Here we define a word `SDL2.dll` which uses the word `dll` defined in the `windows` vocabulary :

```
\ Entry point to SDL2.dll library
windows
z" SDL2.dll" dll SDL2.dll
```

Chaining vocabularies

The search order of a word in a vocabulary can be very important. In the case of words with the same name, we remove any ambiguity by mastering the search order in the different vocabularies that interest us.

Before creating a vocabulary chain, we restrict the search order with the word `only` :

```
windows
order           \ displays: windows >> FORTH
```

```
only
order          \ display: FORTH
```

We then duplicate the chaining of vocabularies with the word **also** :

```
only
order          \ display: FORTH
windows also
order          \ displays: windows >> FORTH
structures
order          \ displays:
               \ structures >> FORTH
               \ windows >> FORTH
```

Here is a compact chaining sequence:

```
vocabulary SDL2
only FORTH also  windows also  structures also
SDL2 definitions
```

The last vocabulary thus chained will be the first explored when executing or compiling a new word.

```
order          \ display:    SDL2 >> FORTH
               \             structures >> FORTH
               \             windows >> FORTH
               \             FORTH
```

The search order here will start with the **SDL2** vocabulary, then **structures**, then **windows** and finally, the **FORTH** vocabulary :

- if the searched word is not found, there is a compilation error;

- if the word is found in a vocabulary, that word will be compiled, even if it is defined in the next vocabulary;

Delayed Action Words

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

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

```
' <word>
\ puts the cfa of <word> on the data stack
```

Example:

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

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

```
' WORDS EXECUTE
\ execute WORDS
```

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

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

`WORDS` can be executed indirectly from the contents of `vector`:

```
vector @ EXECUTE
```

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

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

Definition and usage of words with defer

The initialization of a word defined by `defer` is done by `is` :

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

Executing **vector** causes the execution of the word whose **cfa** was previously assigned:

```
vector          \ execute words
```

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

```
' page is vector
```

vector now executes **page** instead of **words** .

defer are mainly used in two situations:

- defining a forward reference;

- definition of a word depending on the operating context.

In the first case, defining a forward reference allows us to overcome the constraints of the sacrosanct precedence of definitions.

In the second case, defining a word dependent on the operating context allows to solve most of the problems of interfacing with an evolving software environment, to preserve the portability of applications, to adapt the behavior of a program to situations controlled by various parameters without harming software performance.

Setting a Forward Reference

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

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

This causes a compile-time error for **word1** , because **word2** is not yet defined. Here's how to work around this constraint with **defer** :

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

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

A practical case

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

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

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

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

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

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

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

```
defer dayNames

: in-ENGLISH
    ['] ENdayNames is dayNames  ;

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

Now here are the words to manage these two tables:

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

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

Here's what running **getDay** gives :

```
in-ENGLISH 3 getDay type cr   \ display :  Wednesday
in-FRENCH  3 getDay type cr   \ display :  Mercredi
```

Here we define the word `.dayList` which displays the beginning of the names of the days of the week:

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

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

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

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

Creation Words

Forth is more than a programming language. It is a metalanguage. A metalanguage is a language used to describe, specify, or manipulate other languages.

With eForth Windows, one 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 numerical data.

In the Data Structures for eForth Windows chapter, the word `create` was also used. This word creates a header that allows access to a data area stored in memory. Example:

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

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

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

Using does>

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

Here's how it works in Forth:

- **`CREATE`** : This keyword is used to create a new data space in the Windows eForth dictionary. It takes one argument, which is the name you give to 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> … ;` part will be executed.

```
: defREG:
    create ( addr1 -- <name> )
         ,
    does> ( -- regAddr )
```

```
        @
  ;
```

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

```
$00 value DB2INSTANCE
```

Or

```
$00 defREG: DB2INSTANCE
```

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

- more readable eForth Windows source code. Easily detect all constants naming a register;

- we leave ourselves the possibility of modifying the `does> part` of `defREG:` without having to then rewrite the lines of code which would not use `defREG:`

Here is a classic case, processing a data table:

```
\ definition word for one-dimensional array
: 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 be extracted in this array. Here we only retrieve the address containing the value to be extracted.

Example of color management

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

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

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

```
$00 currentCOLOR !
```

Or

```
$00 variable BLACK
BLACK currentCOLOR !
```

Displaying modal boxes

Modal boxes are an integral part of the Windows environment. They are very useful for displaying information or issuing a warning. When closing the modal box, you can even recover certain choices caused by closing the modal box.

A modal box is a bit like a stop sign that pops up on your computer screen.

It is a small window that appears, covering all or part of the page. This window is a modal box:

- to attract attention. It signals something important, such as a confirmation to be given, a choice to be made, or information to be read;

- to request an action. It may require you to fill out a form, click a button, or make a decision before you can continue;

- to inform. It can display an error message, a confirmation of success or simply give additional information.

It's the word "modal" that's important here. It means that you have to act on this box before you can continue using the rest of the app. It's like answering a question before you can move on.

Concrete example: when you want to delete a file on the computer, a modal box often appears asking to confirm the decision. You have to click "Yes" or "No" before the file is actually deleted.

MessageBoxA

The word `MessageBoxA` is defined in the `windows vocabulary` .

Settings:

- **hWnd** Handle to the owner window of the message box to create. If this parameter is `NULL` , the message box has no owner window;

- **lpText** Message to display. If the string consists of multiple lines, you can separate the lines using a carriage return and/or a line feed character between each line.

- **lbCaption** Title of the dialog box. If this parameter is NULL, the default title is *Error* .

- **uType** Content and behavior of the dialog box. This parameter can be a combination of indicators from the indicator groups that we will see in more detail later.

Dialog box content and behavior

Example of a modal box:

```
only
windows also
graphics internals

: MSGbox
    NULL z" Beware, message test!" NULL MB_OK MessageBoxA
    ?dup if
        cr ." You have pressed: " .
    then  ;

: run
    MSGbox
;
```

Running **run** displays this:
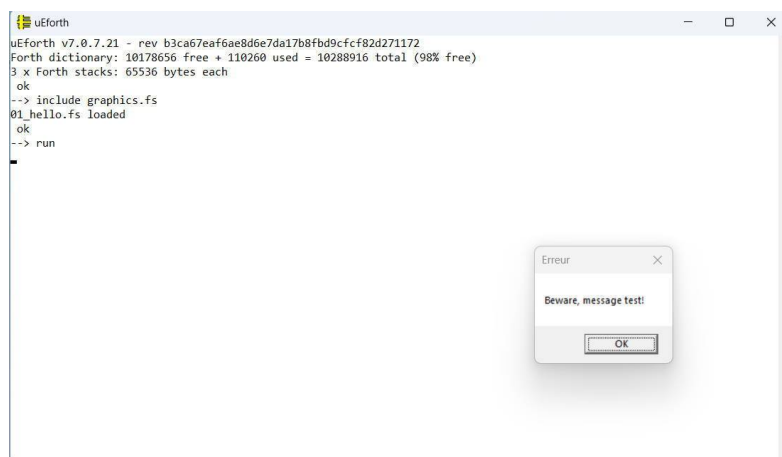
Figure 16: displaying our first modal window

In the **Windows** vocabulary , the constants prefixed MB are associated with **MessageBoxA**:

```
0 constant MB_OK
1 constant MB_OKCANCEL
2 constant MB_ABORTRETRYIGNORE
3 constant MB_YESNOCANCEL
4 constant MB_YESNO
5 constant MB_RETRYCANCEL
6 constant MB_CANCELTRYCONTINUE
```

They correspond to the **uType** parameter and select the buttons displayed in the modal box:

- **MB_OK** The message box contains a send button: OK. This is the default.

- **MB_OKCANCEL** The message box contains two push buttons: OK and Cancel.

- **MB_ABORTRETRYIGNORE** The message box contains three submit buttons: Abort, Retry, and Ignore.

- **MB_YESNOCANCEL** The message box contains three push buttons: Yes, No, and Cancel.

- **MB_YESNO** The message box contains two push buttons: Yes and No.

- **MB_RETRYCANCEL** The message box contains two submit buttons: Retry and Cancel.

- **MB_CANCELTRYCONTINUE** The message box contains three submit buttons: Cancel, Retry, Continue. Use this type of message box instead of **MB_ABORTRETRYIGNORE** .

Return value

If a message box has a *Cancel button* , the word returns the value 2 if you press the Escape key or if the *Cancel button* is selected.

If the message box does not have a *Cancel button* , pressing Esc will have no effect unless an **MB_OK** button is present. If an **MB_OK** button is displayed and the user presses Esc, the return value is 1.

If the function fails, the return value is zero. To get detailed information about the error, use **GetLastError**.

If the function succeeds, the return value is one of the following menu item values:

- **3** The Abandon button has been selected.

- **2** The Cancel button was selected.

- **11** The Continue button has been selected.

- **5** The Ignore button has been selected.

- **7** The No button was selected.

- **1** The OK button has been selected.

- **4** The Retry button was selected.

- **10** The Retry button was selected.

- **6** The Yes button has been selected.

Let's see this with a practical example.

Further explanations

Here is some information about the behavior of `MessageBoxA` .

```
: lpText r" You must make a choice:
 Yes to continue
 No  to stop" s>z ;
z" make a choice"      constant lpCaption

: MSGbox  ( -- )
    NULL lpText lpCaption MB_YESNO MessageBoxA
    ?dup if
        cr ." You have pressed: "
        case
            1 of ." OK"        endof
            6 of ." Yes"       endof
            7 of ." No"        endof
        endcase
    else
        cr ." You canceled the operation."
    then
  ;
```

Here, the `MessageBoxA` word suspends the execution of `MSGbox`. The modal box acts as `a key` in a non-graphical Forth application.
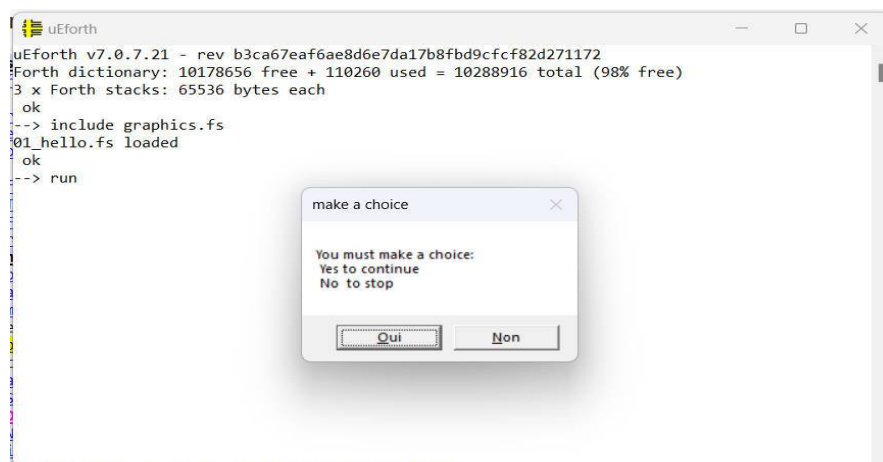
Figure 17: YES/NO modal box test

`lpCaption` word defines the message that serves as the title of the modal box.

`lpText` word contains the message to display in the modal box.

Setting the default button

You can designate a default button. Here are the constants needed to select a button:

```
$00000000 constant MB_DEFBUTTON1
$00000100 constant MB_DEFBUTTON2
$00000200 constant MB_DEFBUTTON3
```

```
$00000300 constant MB_DEFBUTTON4
```

If you have a modal box with a choice between two buttons, you can only use
MB_DEFBUTTON1 or **MB_DEFBUTTON2** .

Example:

```
: MSGbox  ( -- )
    NULL lpText lpCaption
    MB_YESNO MB_DEFBUTTON2 or
    MessageBoxA
    ?dup if
        cr ." You have pressed: "
        case
            1 of ." OK"       endof
            6 of ." Yes"      endof
            7 of ." No"       endof
        endcase
    else
        cr ." You canceled the operation."
    then
  ;
```

Here, the sequence **MB_YESNO MB_DEFBUTTON2** or selects the second button by default.

Added an icon in the modal box

You can add an icon that increases the importance of the content of the modal box,
among these options:

```
$00000010 constant MB_ICONHAND
$00000020 constant MB_ICONQUESTION
$00000030 constant MB_ICONEXCLAMATION
$00000040 constant MB_ICONASTERISK
```


Figure 18: icons for modal box

Integration into Forth code:

```
...
    MB_YESNO MB_DEFBUTTON2 or MB_ICONEXCLAMATION or
    MessageBoxA
...
```

Result :
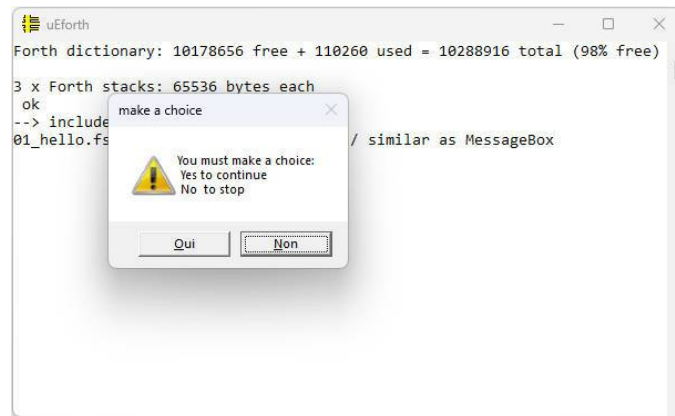
Figure 19: modal box with icon

In summary, we have, with modal windows, a simple and efficient way to interact with a program written in Forth. A modal window does not require reorganizing the display in the parent window.

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** :

Figure 20: 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

Words defined in the `graphics` vocabulary :

```
graphics definitions
windows also

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

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

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

\ The CloseFigure function close a figure in a path.
z" CloseFigure" 1 gdi32 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 GetPixel ( hdc x y -- color )

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

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

```
gdiError CreateFontA GetCurrentObject SetTextColor TextOutA SetPixel GetPixel
CloseFigure Ellipse Rectangle LineTo MoveToEx flip poll wait window heart
vertical-flip viewport scale translate }gg{ screen>g box color pressed?
pixel height width event last-char last-key mouse-y mouse-x RIGHT-BUTTON
MIDDLE-BUTTON LEFT-BUTTON FINISHED TYPED RELEASED PRESSED MOTION EXPOSED
RESIZED IDLE internals
```

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

> You will find online all the words added to the `graphics vocabulary` :
>   https://github.com/MPETREMANN11/eForth-Windows/blob/main/graphics/Gdi32-definitions.fs

Find available functions in a dll file

eForth Windows uses the functions of four DLL files:

Gdi32.dll graphics management functions

Kernel32.dll serves as an interface between applications and the system kernel

User32.dll provides an application programming interface

Shell32.dll plays a crucial role in the Windows user interface

The role of eForth Windows is to draw on all these functions and suggest words to use without worrying about which library this word is attached to.

The problems start when you want to add a word linked to a DLL library. Let's take as an example the **CreateDialog() function** , normally defined in **User32.dll** if we rely on the Microsoft documentation. Attempt to define it in Forth:

```
z"CreateDialog" 4 User32 CreateDialog
```
And there, it's failure!

In the Windows documentation there is a variant **CreateDialogA()** . Let's try again:

```
z"CreateDialogA" 4 User32 CreateDialogA
```
It still doesn't work...

The reason is that the Windows system evolves from version to version. Some functions disappear, others are rewritten. There is a prototype system proposing function aliases. But these mechanisms are only available through the Windows development APIs.

For us, Forth developers, if we don't want to fumble around, it's to list all the functions integrated into a DLL file installed on our machine.

Dependency Walker

This program is available here:
  https://www.dependencywalker.com/

It is a small program, easy to install and use.

On startup, click on *File* and select *Open* .

We will analyze the **User32.dll file** . In Windows 11, this file is located in the **Windows --> System32 folder** .

Don't worry about any error messages.

Result of opening **User32.dll** :
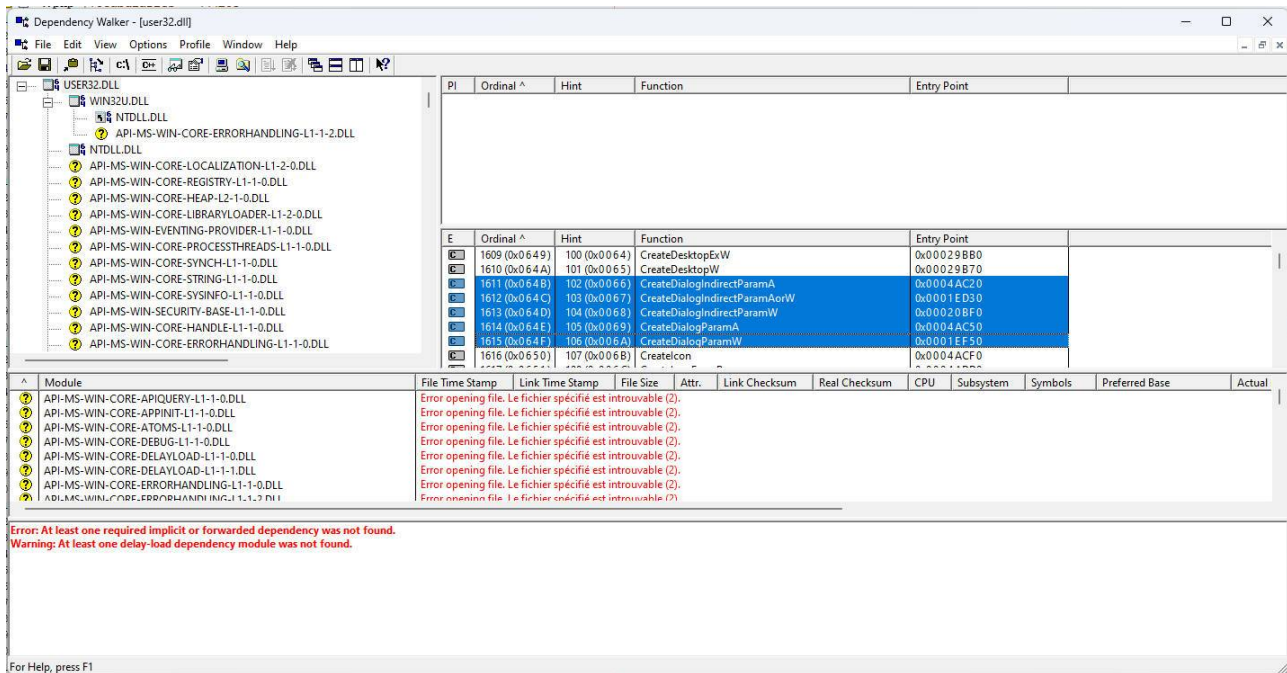
Figure 21: search for User32.dll functions

Here, we found functions that start with **CreateDialog** , but we only find variants. So there is no chance of making a connection with **CreateDialog()** or **CreateDialogA()** . We will therefore have to use the most suitable variant.

This list can be copied to avoid having to open **User32.dll** constantly.

You can rename an unmanaged function to your liking in your eForth code:

```
z"SendMessageA" 4 User32 SendMessage (hWnd msg wParam iParam -- LRESULT)
```

SendMessageA() function is used under the name `SendMessage` in eForth. But this is risky. The other solution is to create an alias:

```
z"SendMessageA" 4 User32 SendMessageA (hWnd msg wParam iParam -- LRESULT)

: SendMessage
    SendMessageA ;
```

Some functions may have a different number of parameters. The point of then going through an alias is to handle any error messages:

```
\ write text
z"TextOutA" 5 Gdi32 TextOutA (hdc xy lpString c -- fl)

: TextOut ( hdc xy lpString c -- fl )
    TextOutA dup 0= if
        abort" TextOut ERROR"
    then ;
```

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.

First graphic drawings

The Application Programming Interfaces (APIs) of the Windows environment are very complex. Mastering these APIs is essential to create programs that work elsewhere than in a simple text terminal. To be very blunt, it is a "gas factory", to put it politely.

Linux zealots might be tempted to claim that Linux is simpler. I have some doubts about this, based on some of the Linux source code. In reality, both systems, Windows and Linux, have similar approaches to managing graphical environments.

So I'm going to try to approach this vast subject in stages, hoping not to lose you along the way.

Open a window

The first word to know, to open a graphical window, is `window` . This word accepts two parameters:

- window width, expressed in pixels;
- window height, expressed in pixels.

Example:

```
graphics
600 400 window
```

This opens a window that is 640 pixels wide and 400 pixels high. This window is independent of the eForth Windows window.

Here, the word `graphics` selects the graphics vocabulary . Contents of this vocabulary:

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

Some of the primitives are defined in the `internals` vocabulary:

```
graphics internals vlist  \ display:
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
```

In the chapter *Extending the graphics vocabulary for Windows* , we already discussed how to extend the content of these two vocabularies.

Line drawing

Opening a window with window stores the handle of the current environment in the `hdc` value.

Before drawing, we position the graphic pointer with `MoveToEx` , which is defined as follows:

```
\ MoveToEx updates the current position
z" MoveToEx"    4 Gdi32 MoveToEx ( hdc x y LPPOINT -- fl )
```

`MoveToEx` parameters :

- **hdc** Handle to a device context
- **x** Specifies the x coordinate, in logical units, of the new position
- **y** Specifies the y coordinate, in logical units, of the new position.
- **LPPOINT** Pointer to a `POINT` structure that receives the current previous position. If this parameter is a `NULL` pointer , the previous position is not returned.

Example :

```
hdc 20 20 NULL moveToEx drop
```

Here we have positioned the graphic pointer at coordinates 20 20. Basically, this position corresponds to 20 pixels from the top edge and the left edge of the current window. We will now make some drawings from this position, with the word `LineTo,` which is first defined as follows:

```
\ LineTo draws a line from the current position to,
\ but not including, the specified point.
z" LineTo" 3 Gdi32 LineTo ( hdc xy -- fl )
```

`LineTo` settings :

- **hdc** Handle to a device context
- **x** Specifies the x coordinate, in logical units, of the new position
- **y** Specifies the y coordinate, in logical units, of the new position.

Example :

```
windows also
graphics internals

: draw
    hdc 20 20 NULL moveToEx drop
    hdc 90 20 LineTo drop
    hdc 90 50 LineTo drop
    hdc 20 90 LineTo drop
```

```
    hdc 20 20 LineTo drop
;

: run09
    600 400 window
    draw
    key drop
    bye
;
```
Result :
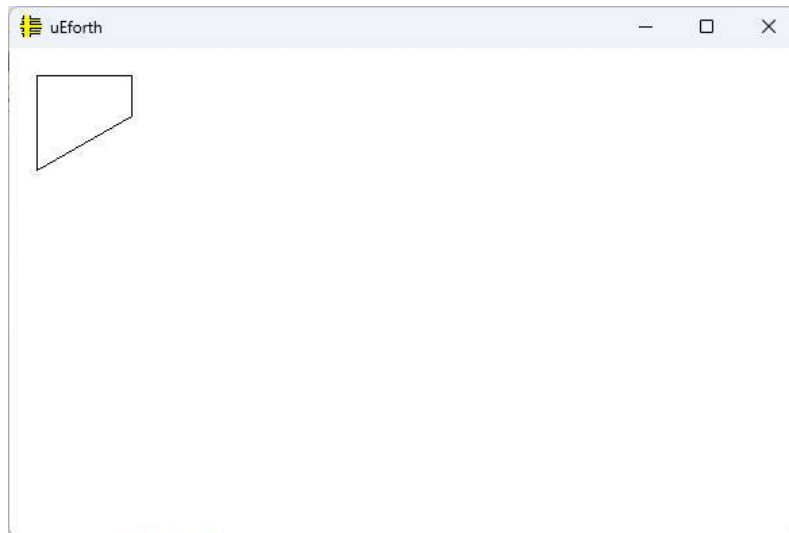
Figure 22: First drawing with LineTo

The default plot color is black.

Coloring of graphical paths

With WinCGI, nothing is simple, but everything is logical. And the color change is part of this immense gas factory that we will try to master.

To change the color of a line drawn with **LineTo** , you must set the pen color before running **LineTo** .

We first define a pen, with the word **CreatePen**, which is defined as follows:

```
z" CreatePen"        3 Gdi32 CreatePen ( iStyle cWidth color -- hPen )
```

Running **CreatePen** requires these parameters:

- **iStyle** style of the outline

- **cWidth** thickness of the plot

- **color** color of the outline

In return, we get a handle needed to select the pen. Example:

```
0 constant PS_SOLID
0 value HPEN_RED
PS_SOLID 1 $FF $00 $00 RGB CreatePen to HPEN_RED
```

Here we defined a **HPEN_RED** value which stores the handle from the execution of
CreatePen .

You can easily define multiple pens:

```
0 value HPEN_RED
0 value HPEN_BLUE

: setPens  ( -- )
    PS_SOLID    3 $FF $00 $00 RGB CreatePen to HPEN_RED
    PS_DOT      1 $00 $00 $FF RGB CreatePen to HPEN_BLUE
  ;
```

Here we define two pens whose handles are stored in **HPEN_RED** and **HPEN_BLUE** .

To select a pen, we go through its handle using the word **SelectObject**, the definition of
which is as follows:

```
z" SelectObject"    2 Gdi32 SelectObject ( hdc h -- fl )
```

Parameters:

- **hdc** handle for domain controller

- **h** handle of the object to select.

Use :

```
: draw
    setPens
    hdc HPEN_RED SelectObject drop
    hdc 20 20 NULL moveToEx drop
    hdc 90 20 LineTo drop
    hdc 90 50 LineTo drop
    hdc 20 90 LineTo drop
    hdc 20 20 LineTo drop

    hdc HPEN_BLUE SelectObject drop
    hdc 25 25 NULL moveToEx drop
    hdc 95 25 LineTo drop
    hdc 95 55 LineTo drop
    hdc 25 95 LineTo drop
    hdc 25 25 LineTo drop
;
```

Result :
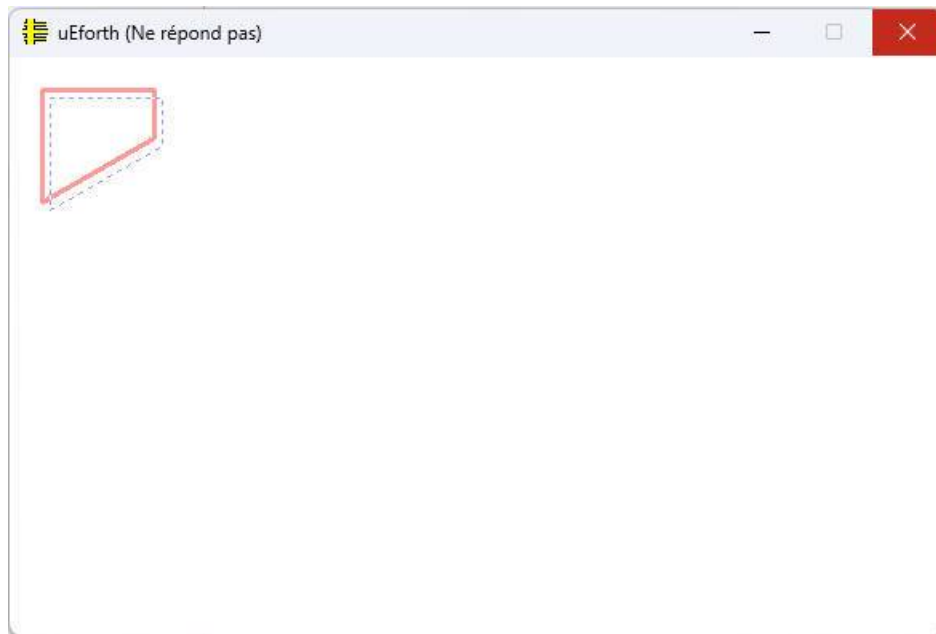
Figure 23: Using two pens

In summary, here are the steps to color a simple outline:

- define a pen with `CreatePen`

- select a pen with `SelectObject`

- make the graphical drawing

In the example above, the blue path is dotted. In fact, behind the apparent complexity on how to manage paths and path colors, we no longer have to re-create a graphics library in Forth.

Drawing geometric shapes

In the previous chapter, we learned how to define and use a pen for the color of paths. Before we get to drawing shapes, we'll see how to define and use a brush.

Color inside the shapes

To color the interior of a shape, we must first define the **CreateSolidBrush word** :

```
z" CreateSolidBrush"  1 Gdi32 CreateSolidBrush ( color -- HBRUSH )
```

Setting :

- color fill **color**

Example :

```
0 value HBRUSH_BLUE

: setBrushes  ( -- )
   $00 $00 $FF RGB CreateSolidBrush to HBRUSH_BLUE
 ;
```

When **setBrushes** is executed, the handle delivered by **CreateSolidBrush** is stored in **HBRUSH_BLUE** .

This **HBRUSH_BLUE** handle will then be selected by **SelectObject**.

Drawing rectangles

The drawing of rectangles is done by the word **Rectangle** , which is defined as follows:

```
\ draw a rectangle
z" Rectangle"   5 gdi32 Rectangle   ( hdc left top right bottom -- fl )
```

Settings:

- **hdc** graphics context handle
- **left** left edge position
- **top** edge position top
- **right** edge position right
- **bottom** edge position

Example :

```
0 value HPEN_RED
0 value HPEN_BLUE

: setPens  ( -- )
```

```
     PS_SOLID     3 $FF $00 $00 RGB CreatePen to HPEN_RED
     PS_DOT       1 $00 $00 $FF RGB CreatePen to HPEN_BLUE
   ;

create RECT01
     10 L,  10 L, 220 L,  80 L,

create RECT02
     25 L,  25 L, 235 L,  95 L,

: draw
    setPens
    setBrushes
    hdc RECT01 GetRect Rectangle drop
    hdc HPEN_RED     SelectObject drop
    hdc HBRUSH_BLUE SelectObject drop
    hdc RECT02 GetRect Rectangle drop
   ;
```

Result :
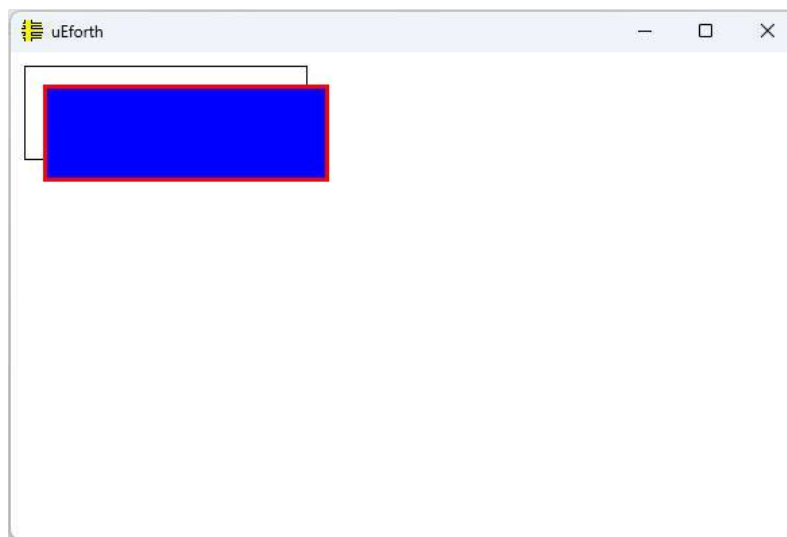
Figure 24: drawing rectangles

Here, we have defined two rectangles via the structures `RECT01` and `RECT02`.

The second rectangle has its edge and background colored by selecting the `HPEN_RED` and `HBRUSH_BLUE` objects.

Polygon drawing

Polygon plotting uses a `POINT type structure` . To define a plot, you first need to create an array containing pairs of xy coordinates, each pair corresponding to the position of an endpoint of a segment describing the edges of the polygon:

```
create POYGON01
     30 L, 37 L,
     44 L, 22 L,
     58 L, 41 L,
     43 L, 54 L,
```

```
     56 L,  68 L,
     43 L,  79 L,
     31 L,  65 L,
     20 L,  75 L,
      5 L,  58 L,
     19 L,  47 L,
      6 L,  31 L,
     19 L,  22 L,
```

Here, we have defined an array named **POLYGON01** containing 12 edges. To draw the polygon, we use the word **Polygon**, which is defined as follows:

```
\ draw a polygone
z" Polygon"   3 Gdi32 Polygon ( hdc *apt cpt -- fl )
```

Settings:

- **hdc** graphics context handle

- **apt** pointer to the array containing the polygon data

- **cpt** table size

Use :

```
: drawPolygon
   setPens
   setBrushes
   hdc HPEN_RED    SelectObject drop
   hdc HBRUSH_BLUE SelectObject drop
   hdc POYGON01 12 Polygon drop
 ;
```

Result :
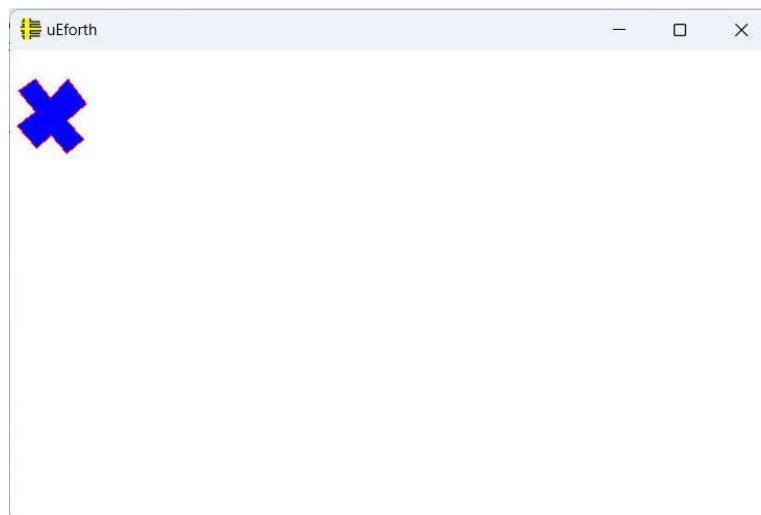
Figure 25: drawing a polygon

There is no need to return to the starting point. The word **Polygon** automatically closes the figure by connecting the last edge to the first edge drawn.

Drawing ellipses

To draw an ellipse, we will use the word **Ellipse**, the definition of which is as follows:

```
\ Draw ellipse
z" Ellipse"      5 gdi32 Ellipse      ( hdc left top right bottom -- fl )
```

Parameters:

- **hdc** graphics context handle

- **left** left edge position

- **top** edge position top

- **right** edge position right

- **bottom** edge position

Example :

```
: drawEllipse ( -- )
    setPens
    setBrushes
    hdc HPEN_RED    SelectObject drop
    hdc 10 10 400 200 Ellipse drop
    hdc HPEN_BLUE   SelectObject drop
    hdc 30 30 420 220 Ellipse drop
  ;
```

Result :
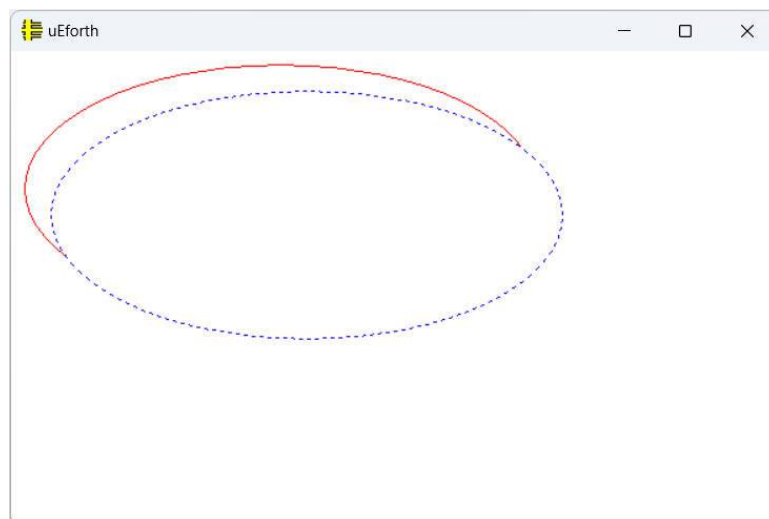
Figure 26: ellipse drawing

Drawing arcs

Arcs are curves drawn with the word **Arc,** the definition of which is as follows:

```
\ Draw arc
z" Arc"       9 gdi32 Arc       ( hdc x1 y1 x2 y2 x3 y3 x4 y4 -- fl )
```

Settings:

- **hdc** graphics context handle
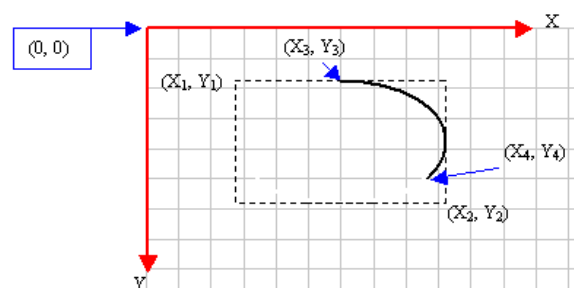
- **x1** and **y1** upper right corner enclosing the arc

- **x2** and **y2** lower right corner enclosing the arc

- **x3** and **y3** start position of arc drawing

- **x4** and **y4** end position of the arc drawing

Example :

```
0 value HPEN_RED
0 value HPEN_BLUE

: setPens  ( -- )
   PS_SOLID    5 $FF $00 $00 RGB CreatePen to HPEN_RED
   PS_SOLID    5 $00 $00 $FF RGB CreatePen to HPEN_BLUE
 ;

: drawArcs ( -- )
   setPens
   hdc HPEN_RED    SelectObject drop
   hdc 10 10 400 200 10 105 400 105 Arc drop
   hdc HPEN_BLUE   SelectObject drop
   hdc 10 10 400 200 400 105 10 105 Arc drop
 ;
```

Result :
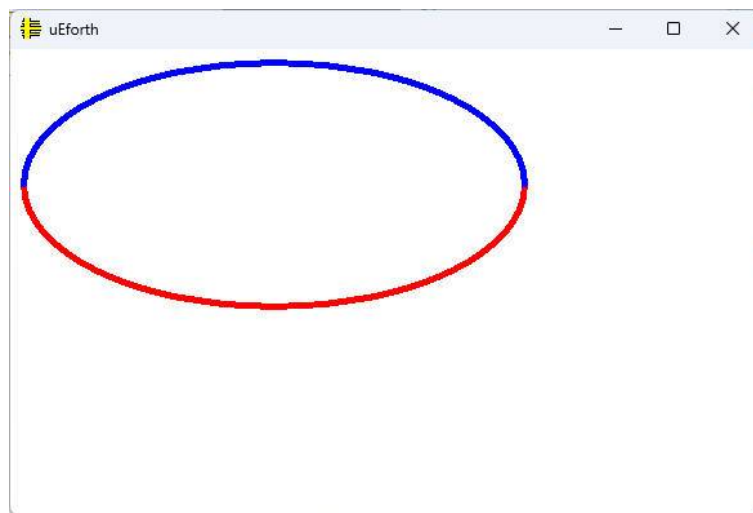
Figure 27: arc tracing

Here we see two half arcs following each other changing color.

To summarize, we have covered the essential drawing words for creating graphic backgrounds: grids, boxes, frames, etc.

Display text in the graphical environment

When eForth Windows is launched, the texts that are displayed are managed in a Windows terminal console. You can do a lot of things in this console, except change fonts, position text to the nearest pixel, frame text, integrate graphical paths...

By displaying text in a Windows window, we will enter a complex but rich universe full of possibilities.

DrawTextA

The word `DrawTextA` is not defined in the `graphics` vocabulary. Here is its definition:

```
only forth
windows definitions

\ draws formatted text in the specified rectangle.
z" DrawTextA"   5 User32 DrawTextA
```

This new word requires five parameters:

- **hdc** handle of the current window;

- **lpchText** pointer to the string to display;

- **cchText** length of the string to display. If this value is -1, the string must be terminated by the code zero;

- **LPRECT** lprc pointer to a structure of type RECT;

- **format** text formatting.

Let's get to the heart of the matter with a first example:

```
only
windows also
graphics internals

: RECT! { left top right bottom addr -- }
    left   addr ->left   L!
    top    addr ->top    L!
    right  addr ->right  L!
    bottom addr ->bottom L!
  ;

create LPRECT
    RECT allot

: STR01 ( -- addr len )
    s" This is my first example." ;

: DRAWtext  ( -- )
    10 10 300 80 LPRECT RECT!
```

```
    hdc STR01 LPRECT DT_TOP DrawTextA
  ;

: run04
    600 400 window 100 ms
    DRAWtext
    key drop
  ;
```

Result :
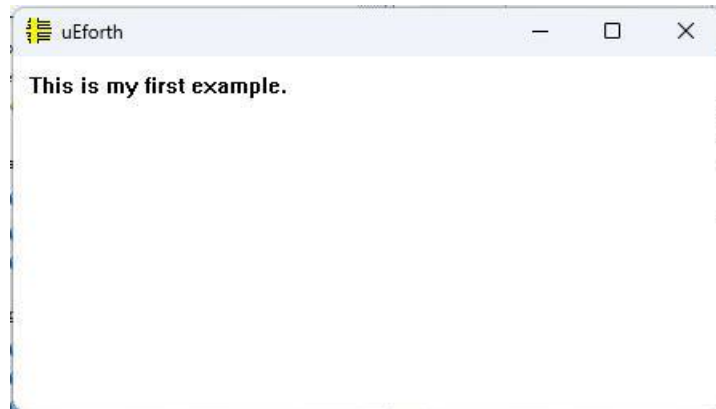
Figure 28: first example of text display

It looks bloated, but this example includes the initialization of various parameters.

Defining the text plot area

Text is displayed in a rectangular area. This rectangular area is defined in a **RECT structure** . If you are not familiar with structures, read the chapter *Data Structures for eForth* .

RECT structure , use **SetRect** which is responsible for assigning the dimensions of the rectangle:

```
create LPRECT
    RECT allot

LPRECT 10 10 300 80 SetRect
```

The sequence **LPRECT 10 10 300 80 SetRect** stores the rectangle parameters in **LPRECT**.

Here the rectangle is 290 pixels wide. What happens if we try to display text that is too long?

```
: STR01 ( -- addr len )
    s" This is my first example.. I try to draw a very long text in this graphic
window." ;

: DRAWtext  ( -- )
    10 10 300 80 LPRECT RECT!
    hdc STR01 LPRECT DT_TOP DrawTextA
```

```
    ;
```
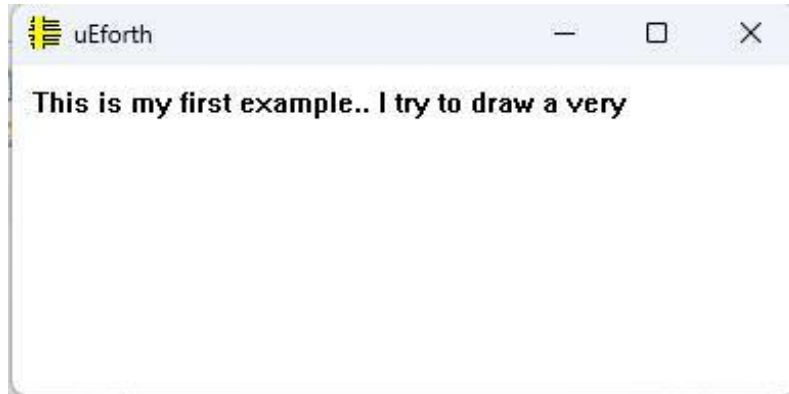

Figure 29: Text display too long

When text extends beyond the edge of the rectangle, it is truncated.

Text Formatting

The formatting of the text drawn by `DrawText` is controlled by its last parameter. In our code, this is defined via the constant `DT_TOP` .

Here are the main formats:

- `DT_TOP` Display from the top of the rectangle

- `DT_LEFT` Aligns text from the left of the rectangle

- `DT_CENTER` Centers the text horizontally within the rectangle

- `DT_RIGHT` Aligns text to the right of the rectangle

- `DT_VCENTER` Centers text vertically

- `DT_BOTTOM` Justifies text from the bottom of the rectangle

- `DT_WORDBREAK` Hyphenate words

There are still other formats, but their study is beyond the scope of this chapter.

These formats can be combined with the `OR operator` , provided that incompatible formats are not combined.

```
: STR01 ( -- addr len )
    s" This is my first example.. I try to draw a very long text in this graphic
window." ;

: FORMATTING ( -- n )
    DT_TOP          \ draw frop top
    DT_WORDBREAK OR \ break words
  ;

: DRAWtext  ( -- )
```

```
    10 10 300 80 LPRECT RECT!
    hdc STR01 LPRECT FORMATTING DrawTextA
  ;
```
Result :
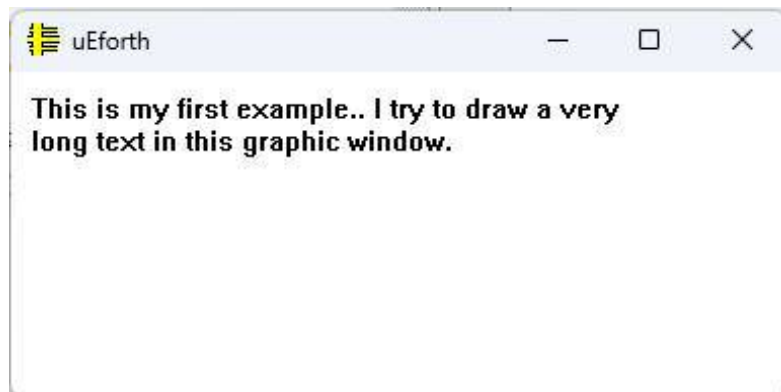
Figure 30: Text formatting

Note that text hyphenation is performed between two words. It is therefore not necessary to worry about the length of the text to be displayed. If a word is likely to overflow, it is automatically carried over to the next line in the rectangle. Let's add text centering and in a narrower rectangle:

```
: FORMATTING ( -- n )
    DT_TOP            \ draw frop top
    DT_WORDBREAK OR \ break words
    DT_CENTER    OR \ center text
  ;

: DRAWtext  ( -- )
    10 10 200 120 LPRECT RECT!
    hdc STR01 LPRECT FORMATTING DrawTextA
  ;
```
Result :
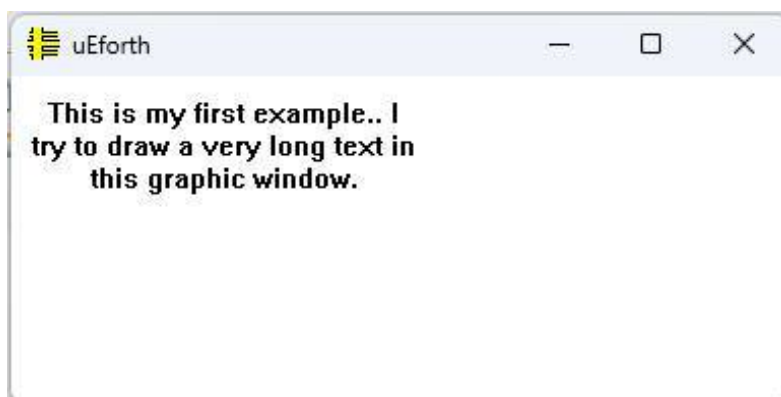
Figure 31: Display in a narrower rectangle

Each time, the word **DrawTextA** places a value on top of the stack. In the last example, it places the value 48. This value corresponds to the vertical dimension, in pixels, of the text that was displayed.

If you don't use this parameter, remove it with **DROP** .

Change text color

To change the text color, you must first define the word **SetTextColor** :

```
only forth
windows also
graphics definitions

\ Set text color
z" SetTextColor"          2 Gdi32 SetTextColor  ( hdc color -- fl )
```

SetTextColor word needs two parameters:

- **hdc** handle of the current context;

- **color** text color .

The color can be encoded as a single 32-bit value, **$00FF00** for example.

Otherwise, to be sure to select a valid color, you can use the word **RGB** :

```
windows
$FF $00 $00 RGB    \ push 32 bits color code on stack
```

Each value processed by **RGB** must be in the range [0..255] in decimal, or [$00..$FF] in hexadecimal. The first value corresponds to the red component, the second to the green component, the third to the blue component.

```
: DRAWtext  ( -- )
    LPRECT 10 10 200 120 SetRect drop
    hdc $FF $00 $00 RGB SetTextColor drop
    hdc STR01 LPRECT FORMATTING DrawTextA drop
  ;
```

Result :
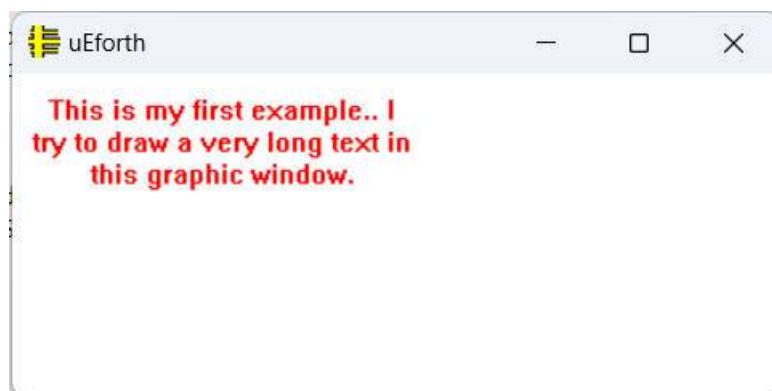
Figure 32: Displaying colored text

TextOutA

This word is defined like this:

```
\ write text
```

```
z" TextOutA"    5 Gdi32 TextOutA  ( hdc x y lpString c -- fl )
```
List of parameters:

- **hdc** context handle;

- **x** x coordinate of the text display position;

- **y y** coordinate of the text display position;

- **lpString** address of the text string to draw. The string does not need to be zero-terminated, because c specifies the length of the string;

- **c** length of the chain.

Use :

```
: String01 s" This is a first test string..."  ;
: String02 s" This is a second test string..."  ;

: TXTout  ( -- )
    hdc 10 10 String01 TextOutA  drop
    hdc 10 30 String02 TextOutA  drop  ;
```
Result :
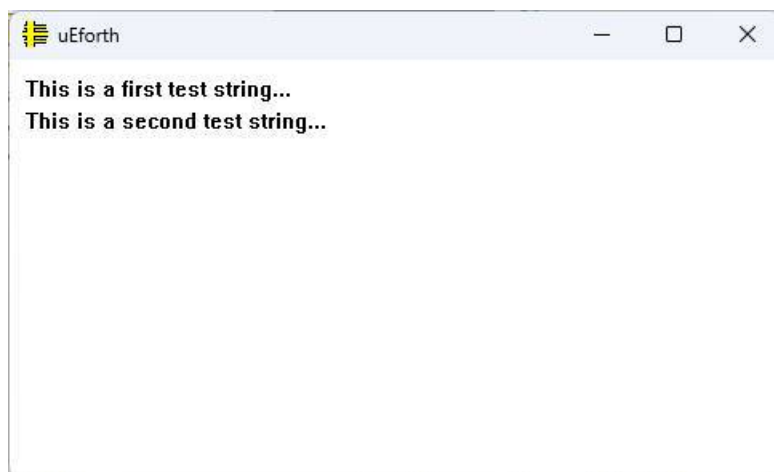
Figure 33: Display with TextOutA

With eForth Windows, it seems that the `SetTextColor` word has no effect on the text color.

Version v 7.0.7.21

FORTH

-	-rot	,	;	:	:noname
!	?	?do	?dup	.	."
.s	'	(local)	[	[']	[char]
[ELSE]	[IF]	[THEN]	]	{	}transfer
@	*	*/	*/MOD	/	/mod
#	#!	#>	#fs	#s	#tib
+	+!	+loop	+to	<	<#
<=	<>	=	>	>=	>BODY
>flags	>flags&	>in	>link	>link&	>name
>params	>R	>size	0<	0<>	0=
1-	1/F	1+	2!	2@	2*
2/	2drop	2dup	3dup	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	bq
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?	default-type	default-use
defer	DEFINED?	definitions	DELETE-FILE	depth	do
DOES>	DROP	dump	dump-file	DUP	echo
editor	else	emit	empty-buffers	ENDCASE	ENDOF
erase	evaluate	EXECUTE	EXIT	extract	F-
f.	f.s	F*	F**	F/	F+
F<	F<=	F<>	F=	F>	F>=
F>S	F0<	F0=	FABS	FATAN2	fconstant
FCOS	fdepth	FDROP	FDUP	FEXP	fq
file-exists?	FILE-POSITION	FILE-SIZE	fill	FIND	fliteral
FLN	FLOOR	flush	FLUSH-FILE	FMAX	FMIN
FNEGATE	FNIP	for	forget	FORTH	forth-builtins
FOVER	FP!	FP@	fp0	free	FROT
FSIN	FSINCOS	FSQRT	FSWAP	fvariable	graphics
here	hex	hld	hold	I	if
IMMEDIATE	include	included	included?	internals	invert
is	J	K	key	key?	L!
latestxt	leave	list	literal	load	loop
LSHIFT	max	min	mod	ms	ms-ticks
mv	n.	needs	negate	next	nip
nl	NON-BLOCK	normal	octal	OF	ok
only	open-blocks	OPEN-FILE	OR	order	OVER
pad	page	PARSE	pause	pause?	PI
postpone	postpone,	precision	previous	prompt	quit
r"	R@	R/O	R/W	R>	r|
r~	rdrop	READ-FILE	recognizers	recurse	refill
remaining	remember	RENAME-FILE	repeat	REPOSITION-FILE	required
reset	resize	RESIZE-FILE	restore	revive	rm
rot	RP!	RP@	rp0	RSHIFT	s"
S>F	s>z	save	save-buffers	scr	sealed
see	set-precision	set-title	sf,	SF!	SF@
SFLOAT	SFLOAT+	SFLOATS	sign	SL@	SP!
SP@	sp0	space	spaces	start-task	startswith?
startup:	state	str	str=	streams	structures
SW@	SWAP	task	tasks	terminate	then
throw	thru	tib	to	touch	transfer
transfer{	type	u.	U/MOD	U<	UL@
UNLOOP	until	update	use	used	UW@
value	VARIABLE	visual	vlist	vocabulary	W!
W/O	while	windows	words	WRITE-FILE	XOR

windows

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_CHANGECBCHAIN 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_MDIICONARRANGE WM_MDICASCADE WM_MDITILE WM_MDIMAXIMIZE WM_MDINEXT WM_MDIRESTORE 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_MOUSEHWHEEL WM_XBUTTONDBLCLK WM_XBUTTONUP WM_XBUTTONDOWN WM_MOUSEWHEEL WM_MOUSELAST WM_MBUTTONDBLCLK WM_MBUTTONUP WM_MBUTTONDOWN WM_RBUTTONDBLCLK WM_RBUTTONUP WM_RBUTTONDOWN WM_LBUTTONDBLCLK 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_MENURBUTTONUP 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_KEYLAST WM_UNICHAR WM_SYSDEADCHAR WM_SYSCHAR WM_SYSKEYUP WM_SYSKEYDOWN WM_DEADCHAR WM_CHAR WM_KEYUP WM_KEYDOWN WM_KEYFIRST 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_NCXBUTTONDBLCLK WM_NCXBUTTONUP WM_NCXBUTTONDOWN WM_NCMBUTTONDBLCLK WM_NCMBUTTONUP WM_NCMBUTTONDOWN WM_NCRBUTTONDBLCLK WM_NCRBUTTONUP WM_NCRBUTTONDOWN WM_NCLBUTTONDBLCLK WM_NCLBUTTONUP WM_NCLBUTTONDOWN WM_NCMOUSEMOVE WM_SYNCPAINT WM_GETDLGCODE WM_NCACTIVATE WM_NCPAINT WM_NCHITTEST 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_VKEYTOITEM WM_DELETEITEM WM_MEASUREITEM WM_DRAWITEM WM_SPOOLERSTATUS WM_NEXTDLGCTL WM_ICONERASEBKGND 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_ERASEBKGND 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_SIZENS 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_SHWOMINNOACTIVE SW_MINIMIZE SW_SHOW SW_SHOWNOACTIVE
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 process-heap HeapReAlloc HeapFree HeapAlloc GetProcessHeap
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 d0<ior 0=ior ior FILE_END FILE_CURRENT FILE_BEGIN FILE_ATTRIBUTE_NORMAL
OPEN_EXISTING CREATE_ALWAYS FILE_SHARE_WRITE FILE_SHARE_READ GetFileSize
SetEndOfFile SetFilePointer MoveFileA DeleteFileA FlushFileBuffers CloseHandle
WriteFile ReadFile CreateFileA NULL wargs-convert wz>sz wargv wargc CommandLineToArgvW
Shell32 GetModuleHandleA GetCommandLineW GetLastError WaitForSingleObject
GetTickCount Sleep ExitProcess Kernel32 contains? dll sofunc GetProcAddress
LoadLibraryA WindowProcShim SetupCtrlBreakHandler boot_extra 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