

The manual for Z79FORTH

version 1.4 - 22 October 2024



Author

- François LAAGEL
- Marc PETREMANN

Collaborators

- ...

Contents

Author.....	1
Collaborators.....	1
Introduction.....	5
Background and Objective.....	5
Object.....	5
Variants.....	6
The Z79Forth board.....	7
Power supply for the Z79Forth card.....	9
The USB cable.....	9
Communicating with the Z79Forth card.....	10
Using a terminal.....	11
Install and use Tera Term.....	11
Tera Term Setup.....	11
First communication with Tera Term.....	13
Compiling FORTH source code to Z79Forth board.....	13
Editing and managing source files for Z79Forth.....	15
Text File Editors.....	15
Using an IDE.....	15
Storage on GitHub.....	17
Some good practices.....	18
Using Numbers with Z79Forth.....	20
Numbers with the FORTH interpreter.....	20
Entering numbers with different number bases.....	20
Change of digital base.....	21
Binary and Hexadecimal.....	22
Size of numbers on FORTH data stack.....	24
Memory access and logical operations.....	25
A real 16-bit FORTH with Z79Forth.....	27
Values on the data stack.....	27
Values in memory.....	27
Word processing depending on data size or type.....	28
Conclusion.....	29
Displaying numbers and strings.....	31
Change of digital base.....	31
Prefixing whole numbers.....	32
Defining new display formats.....	32
Displaying characters and strings.....	35
Comments and debugging.....	37
Write readable FORTH code.....	37
Use UNICODE characters in names.....	38

Source code indentation.....	39
Comments.....	39
Stack Comments.....	40
Meaning of stack parameters in comments.....	40
Word Definition Words Comments.....	41
Text comments.....	42
Comment at the beginning of the source code.....	42
Stack monitor.....	43
The decompiler.....	44
Dictionary / Stack / Variables / Constants.....	45
Expand the dictionary.....	45
Stacks and Reverse Polish Notation.....	45
Handling the parameter stack.....	46
The Return Stack and Its Uses.....	47
Memory Usage.....	47
Variables.....	48
Constants.....	48
Pseudo-constant values.....	48
Basic tools for memory allocation.....	49
Word Creation Words.....	50
Using does>.....	50
Example of color management.....	51
Example, writing in pinyin.....	52
Marking functional layers with MARKER.....	53
How the word FORGET works.....	53
Marking with the word marker.....	54
Data structures for Z79Forth.....	56
Preamble.....	56
Tables in FORTH.....	56
One-dimensional 16-bit data array.....	56
Table definition words.....	57
Reading and writing in a table.....	57
Management of complex structures.....	58
Definition of struct and field.....	58
Examples of structures.....	60
Unicode characters with emit and key.....	62
Unicode encoding.....	62
Retrieve a Unicode character from the terminal.....	63
Displaying Unicode Characters.....	64
Unicode characters in Forth words.....	66
Using binary code.....	68
Retrocomputing and back to basics.....	68
FORTH: compact and elegant.....	69
A FORTH version with direct chaining.....	70
Decompile Z79Forth definitions.....	70

Coding definitions in machine language.....	71
Ressources.....	74
GitHub.....	74
Youtube.....	74

Introduction

Z79Forth is a platform designed as a basis for self-education and further hardware development. It is an Hitachi HD6309 based single board computer running FORTH as its operating system, runtime and development environment. Both 79-STANDARD and ANS94 Forth variants are available as EEPROM images. Applications include an Hexadoku solver and a Pacman implementation dedicated to authentic DEC text terminals.

Background and Objective

Once, on this very planet, there was a time when one could understand the inner workings of a computer almost down to the bit level. The hardware was supplied with schematics and the software source code was provided. Are those days behind us and forever lost in the memory of *old timers*?

Z79Forth is an effort to prove that it is not the case and that the KISS (Keep it Simple Stupid) principle can still produce a fully understandable computer. It is offered to folks who are not afraid of electronics assembly and are willing to learn the Forth programming language.

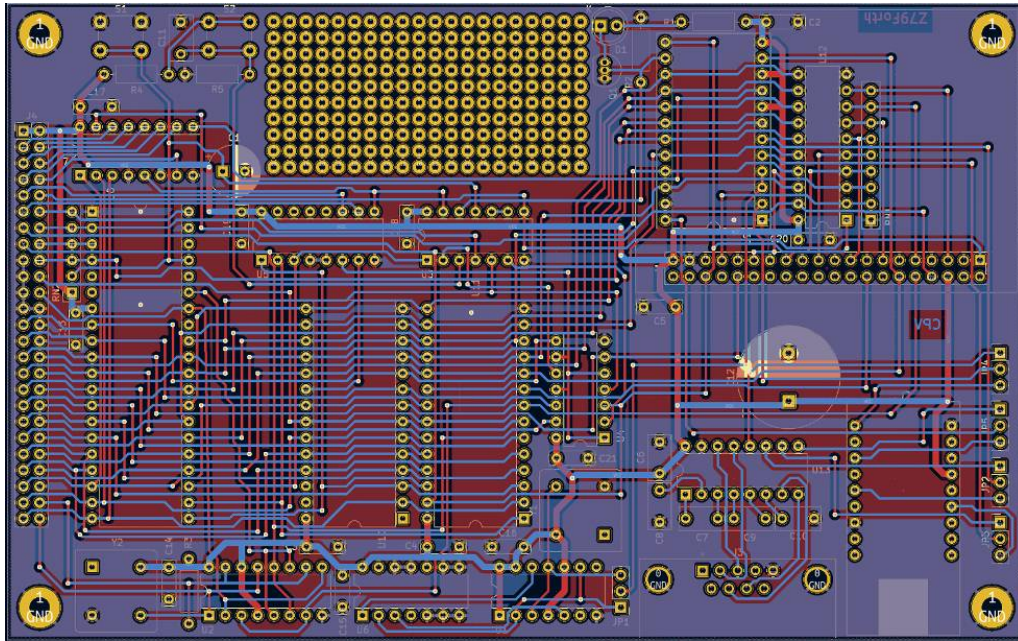
Object

The deliverable of this project is an electronic kit that buyers will have to put together by themselves. It is a single board computer that keeps open the door to further hardware developments. Its main features are:

- 4 megahertz Hitachi HD63C09E processor (Motorola MC6809 compatible).
- 8 kilobyte EEPROM **Forth** resident firmware.
- 32 kilobytes of static RAM of which 25 remain available to applications.
- 64 megabytes of CompactFlash mass storage.
- serial line communication over USB or RS232 (115200 or 38400 bits per second).
- a connector for off-board extended functionality.
- USB phone charger (supplied) powered via a mini-B plug.
- very low power consumption: less than 1 watt in most cases.

Some of the components used in this kit are **collector's items**, only available on ebay. Overall **quality** is a primary concern. This is reflected by the selection of screw machine IC sockets. The supplied firmware is **Open Source**; it is the result of 4 years of off and on development work.

Once the kit is assembled, buyers of this platform will be able to think of themselves of owners of a computer that is only available as a **limited edition**.



Variants

Depending on your willingness to go down the memory lane, you may select either of the following standard compliant implementations of the Forth language:

- the 79-STANDARD specification which might be considered by some as being only of historic value.
- the ANS94 specification which allows contemporary code to be ported easily.

The Z79Forth board

Forth like it's 1979 all over again!

This platform is designed as a basis for self-education and further hardware development. The target Forth variant is the 79-STANDARD, an historic reference. The whole design is based on the Hitachi HD63C09E--a much improved implementation of the Motorola MC6809.

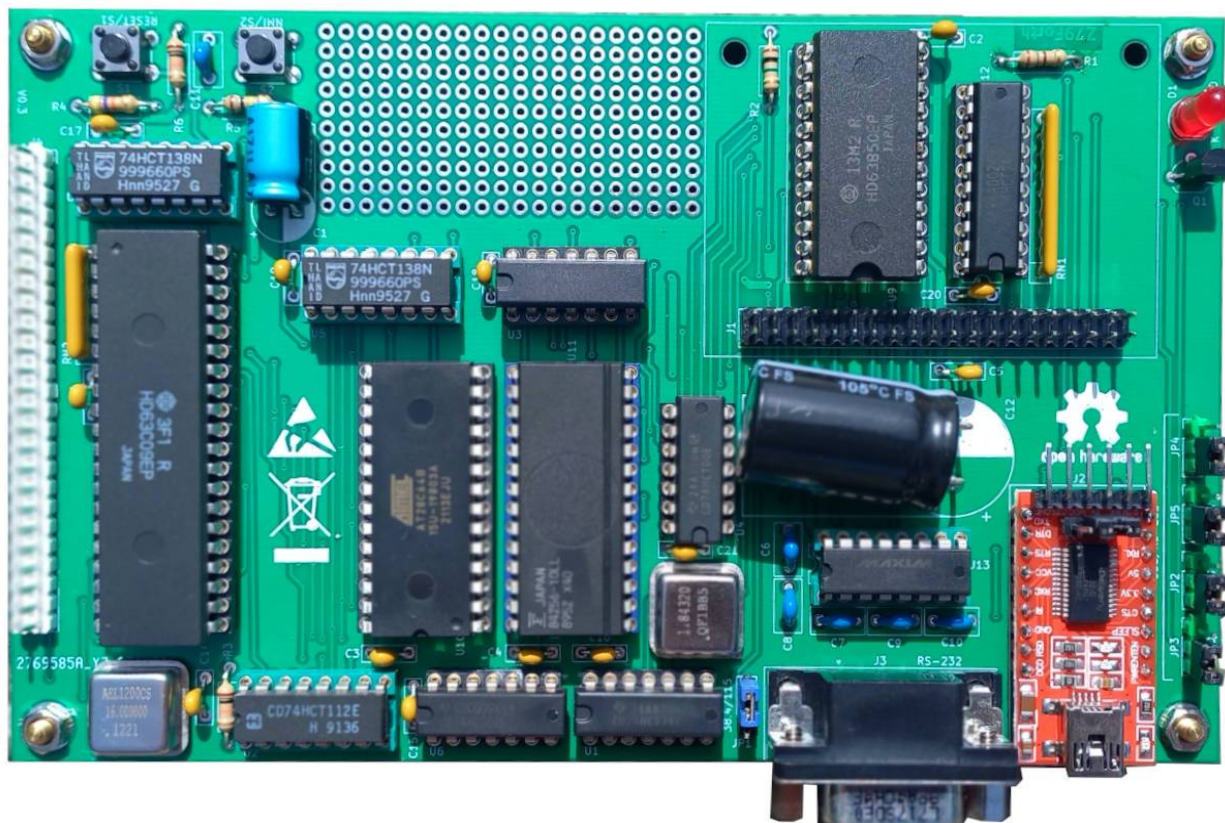


Figure 1: The 79Forth board

Main features are :

- 5 MHz CPU operation.
- 32 KB static RAM. Conceivably expandable to 48 KB.
- 8 KB EEPROM running a native 79-STANDARD Forth sub-set implementation.
- 6 spare IO device lines are decoded and available for further developments.
- USB powered. The current consumption is somewhere between 56 and 150 mA.
- Serial line console operating at 115200 bps.
- Mass storage support on SanDisk CompactFlash (up to 64 MB).

- Interrupt free design.

The software is licensed under the GNU General Public License version 3 and is available at <https://github.com/frenchie68/Z79Forth>

Kicad schematics are also provided over there.

Project status: working wire wrapped prototype. The software itself is believed to perform according to specifications. There are no known bugs at this time.

Power supply for the Z79Forth card

The Z79Forth card can be powered in two ways:

- with a 230V-5V power supply and a USB-A 2.0 to Mini USB male cable;
- from a computer via the USB-A 2.0 to Mini USB male cable.

The USB cable

In both cases, you must use a USB-A 2.0 to Mini USB male cable:



Figure 2: USB 2.0 cable and 230V->5V power supply

You can power the Z79Forth board from the serial port of a computer. In this case, only the USB cable is needed. This solution should only be used if you have few USB devices connected simultaneously to the computer.

If you need to test components on the Z79Forth board, use a USB hub. In the event of an incident, this solution will prevent damage to the computer's USB port.

If you want to communicate with the Z79Forth board and power it at the same time via the USB port, you will need to set the connectors **JP1** to **JP5** on the board. See chapter Communicating with the Z79Forth board.

Communicating with the Z79Forth card

There are two ways to communicate with the Z79Forth board:

- by serial ↔ USB link, via a USB-A 2.0 to Mini USB male cable, see *Powering the Z79Forth board* .
- by USB ↔ RS232 DB9 cable.

If you are using the USB-A 2.0 to Mini USB Male cable, check the position of jumpers JP2 to JP5 :

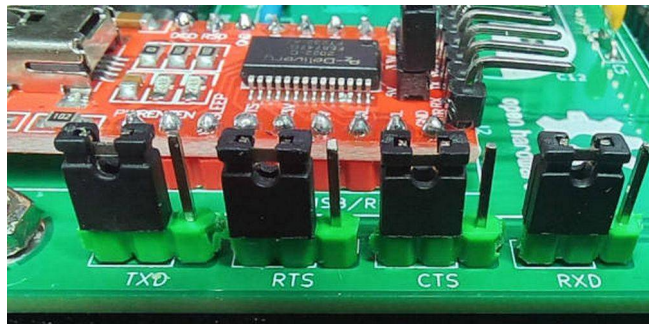


Figure 3: position of jumpers JP2 to JP5

For connection via a USB-A 2.0 male to Mini USB cable, the jumpers must be in the position shown in the photo above.

Jumper JP1 is located to the left of the RS232 DB9 connector. This jumper sets the serial transmission speed :

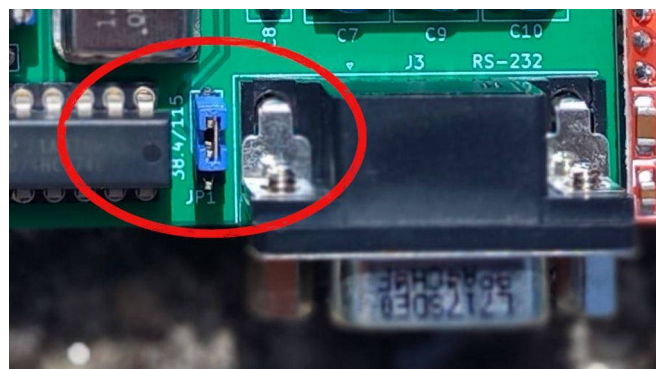


Figure 4: position of jumper JP1

In position 2-3 of jumper JP1, as seen in the photo above, the serial link is set to 115200 bps. If you are using a terminal that does not communicate at this speed, put this jumper in position 1-2. In position JP1 1-2, the serial transmission speed will be reduced to 38400 bps.

Using a terminal

There are many software programs that can emulate a terminal. Here are the most popular ones:

- **Putty** on Windows or Linux;
- **Tera Term** on Windows;
- **Minicom** on Linux

The terminal allows communication with the FORTH interpreter installed on the Z9Forth card.

Install and use Tera Term

For Windows programmers, I recommend the Tera Term terminal. You can download it from this site:

<https://teratermproject.github.io/index-en.html>

Version 5.2 is not large. It takes 8.8MB before installation. Download the *teraterm-5.2.exe* file . Once downloaded, open this file and follow the instructions until the installation is complete.

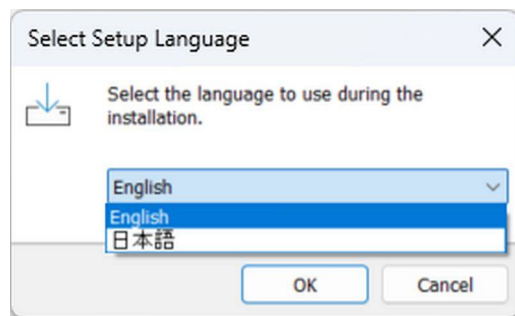


Figure 5: Tera

Term Language Selection

Select English as the language you want to use for Tera Term. Installation is quick and easy.

Tera Term Setup

To communicate with Z9Forth, some parameters need to be set.

Click on *Setup* and select *Serial Port*

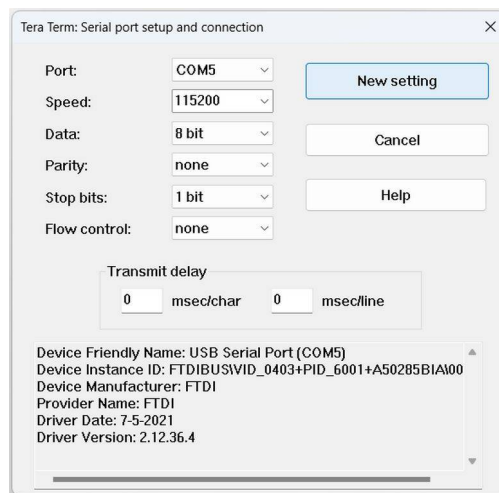


Figure 6: serial link settings

Do not select the serial port. Enter the other parameters as described in the figure above.

Click on *Setup* and select *Windows*. This window allows you to set the text color in the terminal and the background color. Do according to your preferences.

Click on *Setup* and select *Terminal*

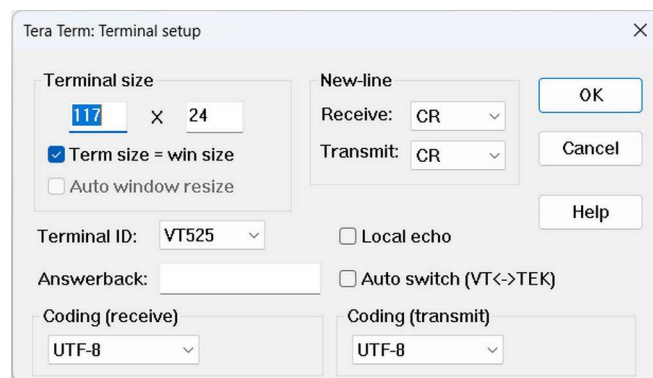


Figure 7: Sélection du terminal

Here, we have selected the **VT525 terminal**. This terminal allows you to use all existing characters: standard ASCII, extended ASCII, accented characters in many languages, Chinese, Japanese, Korean ideograms, etc. This aspect is the subject of the chapter *Unicode characters with emit and key*.

At the dawn of computing, the first computers were very large. And to use them, we communicated with these machines via a terminal. A VT (for Video Terminal), is a computing device that allows a user to interact with a remote computer. It is essentially a screen and a keyboard that sends commands to a main computer and transmits the results to a local screen. VT terminals played a crucial role in the development of computing, especially in Unix and mainframe environments.

Tera Term is a terminal emulator. To communicate with our Z79Forth board, we will only use the **COMx serial port**, where **x** is the serial port number.

First communication with Tera Term

Once Tera Term is set up, click *Setup* and select *Save setup* . *Save the settings under the default file name TERATERM.INI* .

Close Tera Term. Restart Tera Term. Your microcomputer must be connected to the Z79Forth card via USB port. If all goes well, the connection is immediate. If this is not the case:

- Click on *File* and select *New Connection*
- Click on *Serial* . The port offered is usually the one connected to the Z79Forth card.

If the connection is successful, press the RET key on the keyboard several times. This is what you get:

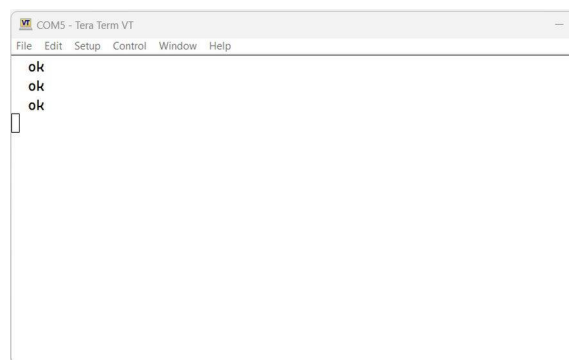


Figure 8: Communication between Z79Forth and Tera Term is established

The settings are roughly similar with other terminal emulation programs.

Compiling FORTH source code to Z79Forth board

First of all, let's remember that the FORTH language is on the Z79Forth card! FORTH is not on your PC. So, you can't compile the source code of a program in FORTH language on the PC.

To compile a program in FORTH language, you must first open a source file on the PC with the editor of your choice.

Next, we copy the source code to compile. Here, a source code opened with Wordpad:

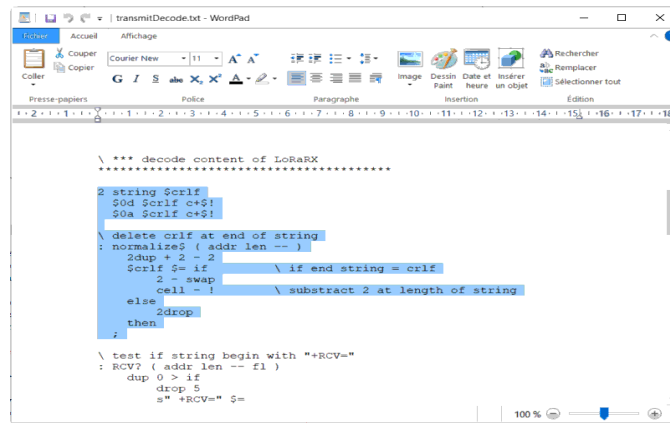


Figure 9: Edit FORTH source code with Wordpad

Select the source code or portion of code you are interested in. Then click *copy*. The selected code is in the PC's editing buffer.

Click on the Tera Term terminal window. Click *Paste*.

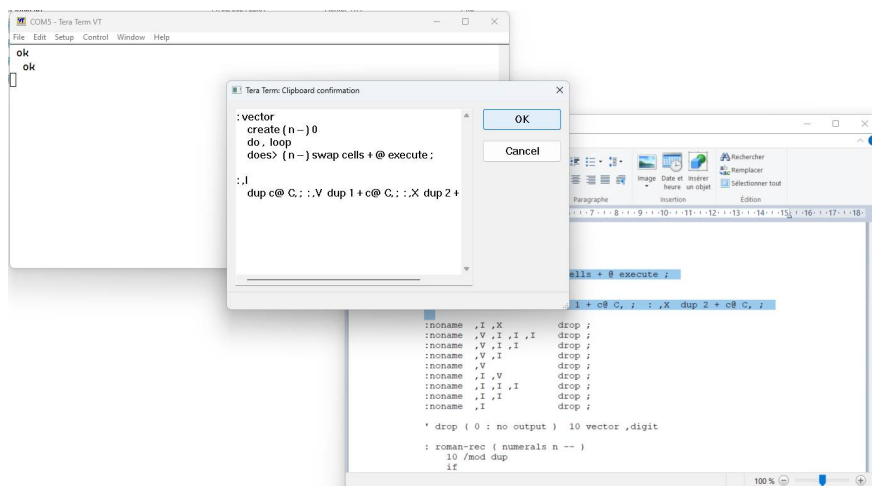


Figure 10: Copy/pasted a portion of code from Wordpad to Z79Forth via Tera Term

The FORTH code thus copied/pasted will be immediately interpreted or compiled by Z79Forth.

To run a compiled code, simply type the word FORTH to launch, from the Tera Term terminal.

Editing and managing source files for Z79Forth

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";

Text File Editors

edit file editor is the simplest:

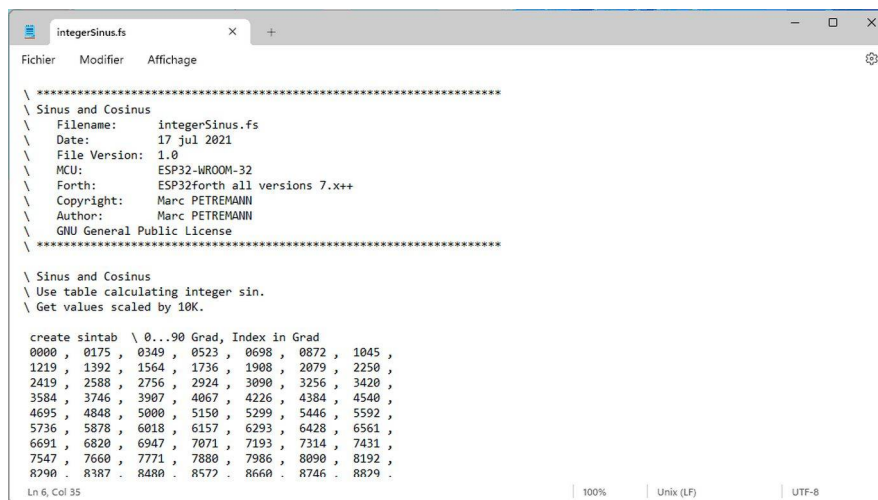


Figure 11: editing with edit under windows 11

Other editors, such as **WordPad**, are not recommended, as you risk saving the FORTH source code in a file format that is not compatible with Z79Forth.

On Linux, the equivalent is called **gEdit**. MacOS also has a simple text editor.

If you use a custom file extension, such as **4th**, 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** :

1 Integrated Development Environment

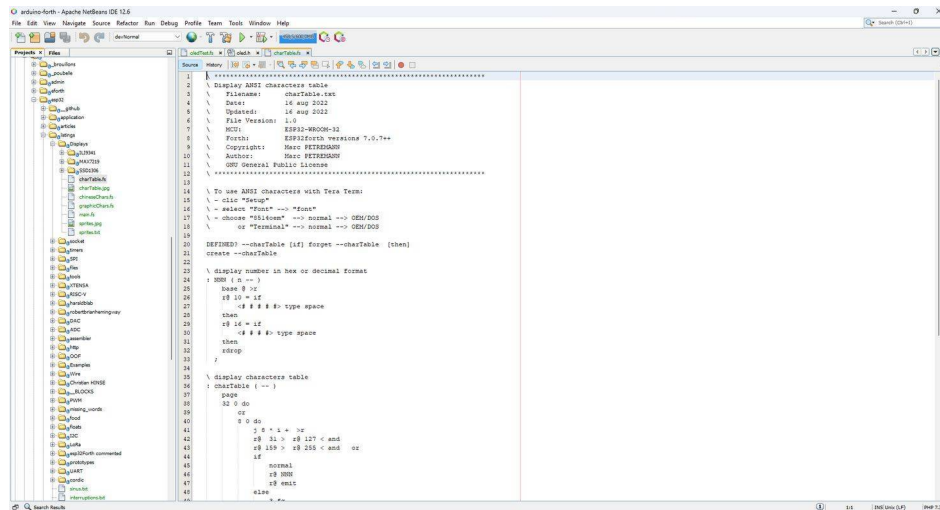


Figure 12: editing with Netbeans

Netbeans offers several interesting features:

- version control with **GIT** ;
- recovery of previous versions of modified files;
- file comparison with **Diff** ;
- one-click **FTP upload** 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.

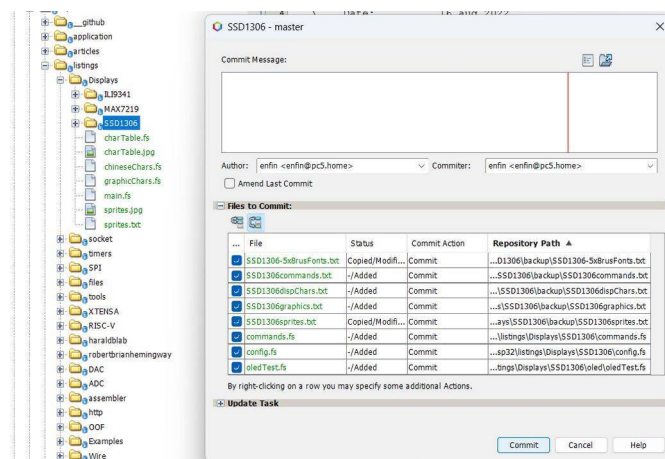


Figure 13: GIT operation in Netbeans

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

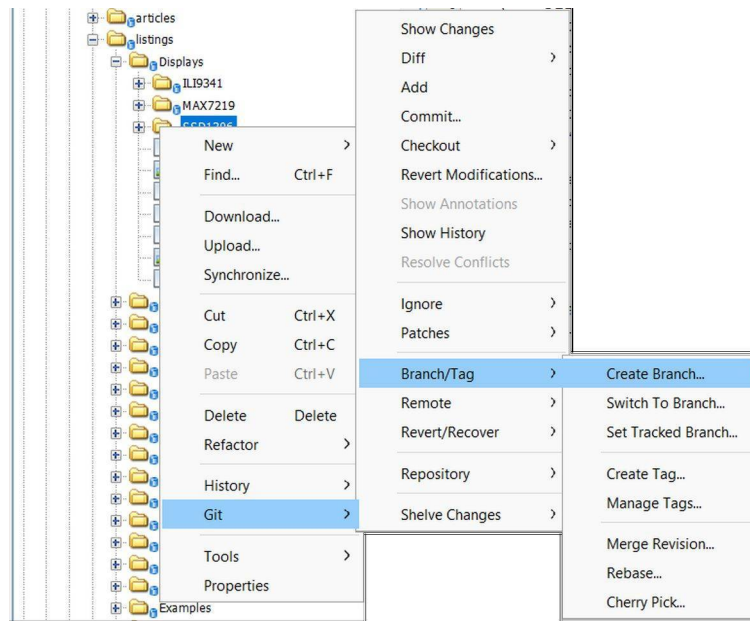


Figure 14: creating a branch on a project

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

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

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

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

Storage on GitHub

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

² <https://github.com/>

³ <https://sourceforge.net/>

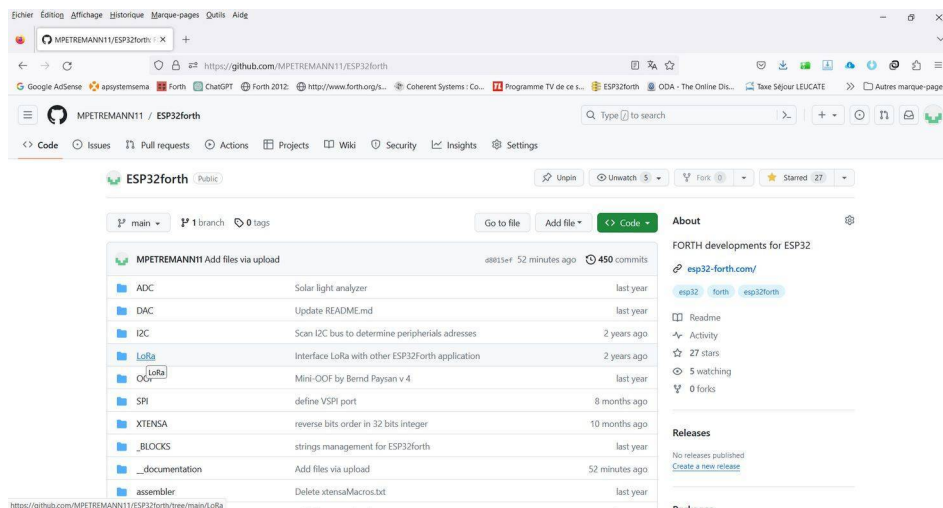


Figure 15: storing files on Github

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

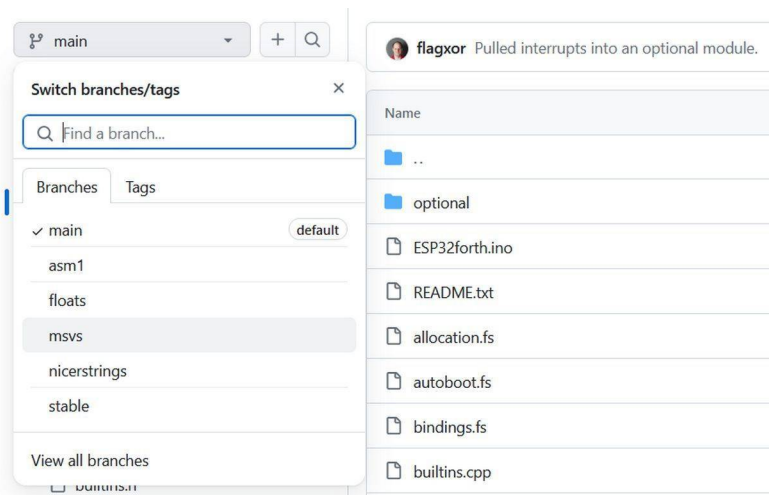


Figure 16: access to a project branch

Some good practices

The first best practice is to name your working files and folders well. You are developing for Z79Forth, so create a folder named **Z79Forth** .

sandbox subfolder in this folder .

For well-built projects, create one folder per project. For example, if you are programming a game, create a subfolder called **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.4th** to store project settings;

- **documentation** directory to store files in the format of your choice, related to the project documentation;
- **myApp.4th** for your project definitions. Choose a file name that is fairly descriptive. For example, to manage a maze game, take the name **game-maze.4th** .

Once the project is in its final state, the ideal is to create the blocks in Z79Forth's block management system, allowing near-instant compilation of the application code.

Using Numbers with Z79Forth

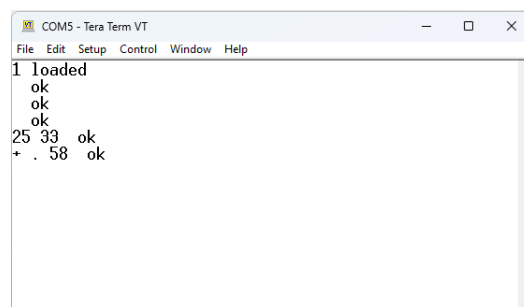
We've started Z79Forth without any issues. Now we'll dive deeper into some number crunching to understand how to master 79Forth.

We will begin by addressing these basic concepts by inviting you to carry out simple manipulations.

Numbers with the FORTH interpreter

When Z79Forth starts, the TERA TERM terminal window (or any other terminal program of your choice) should indicate that Z79Forth is available. Press the *ENTER* key on the keyboard once or twice. Z79Forth responds with the confirmation of successful execution **ok** .

We will test the entry of two numbers, here **25** and **33** . Type these numbers, then *ENTER* on the keyboard. Z79Forth always responds with **ok** . You have just stacked two numbers on the FORTH language stack. Now enter **+** . then the *ENTER* key . Z79Forth displays the result:



```
COM5 - Tera Term VT
File Edit Setup Control Window Help
1 loaded
ok
ok
ok
25 33 ok
+ . 58 ok
```

Figure 17: first operation with Z79Forth

This operation was handled by the FORTH interpreter.

Z79Forth, like all versions of the FORTH language, has two states:

- **interpreter** : the state you just tested by performing a simple sum of two numbers;
- **compiler** : a state that allows new words to be defined. This aspect will be explored further later.

Entering numbers with different number bases

In order to fully understand the explanations, you are invited to test all the examples via the TERA TERM terminal window or any other terminal program of your choice.

Numbers can be entered naturally. In decimal, it will ALWAYS be a sequence of numbers, for example:

```
-1234 5678 + .
```

The result of this example will display **4444**. FORTH numbers and words must be separated by at least one *space character*. The example works perfectly if you type one number or word per line :

```
-1234
5678
+
.
```

Numbers can be prefixed if you want to enter values other than in decimal form:

- **\$** sign to indicate that the number is a hexadecimal value;
- **%** sign to indicate that the number is a binary value;

Example :

```
255 . \ display 255
$ff . \ display 255
```

The purpose of these prefixes is to avoid any misinterpretation in the case of similar values:

```
$0305
0305
```

are not **equal numbers** if the hexadecimal numeric base is not explicitly defined!

Change of digital base

Z79Forth has words to change the number base:

- **hex** to select the hexadecimal number base;
- **decimal** to select the decimal numeric base.

Any number entered into a numeric base must respect the syntax of numbers in that base:

```
3E7F
```

will cause an error if you are in decimal base.

```
hex 3e7f
```

will work perfectly in hexadecimal base. The new numeric base remains valid as long as you do not select another numeric base:

```
hex
$0305
0305
```

are equal numbers !

Once a number is placed on the data stack in a numeric base, its value does not change. For example, if you place the value **\$ff** on the data stack, this value, which is **255** in decimal, or **11111111** in binary, will not change if you switch back to decimal:

```
hex ff decimal. \display: 255
```

At the risk of insisting, **255** in decimal is **the same value** as **\$ff** in hexadecimal!

In the example given at the beginning of the chapter, we define a constant in hexadecimal :

```
25 constant myScore
```

If we type:

```
hex myScore .
```

This will display the contents of this constant in its hexadecimal form. The change of base has **no consequence** on the final operation of the FORTH program.

Binary and Hexadecimal

The modern binary numbering system, the basis of the binary code, was invented by Gottfried Leibniz in 1689 and appears in his paper Explanation of Binary Arithmetic in 1703.

In his article, LEIBNITZ uses only the characters **0** and **1** to describe all numbers:

```
: bin0to15 ( -- )
  2 base !
  $10 0 do
    cr i .
  loop
  cr decimal ;
bin0to15 \ display:
0
1
10
11
100
101
110
111
1000
1001
1010
1011
1100
1101
1110
1111
```

Is it necessary to understand binary coding ? I would say yes and no. **No** for everyday uses. **Yes** to understand microcontroller programming and mastery of logical operators.

It was George Boole who formally described logic. His work was forgotten until the advent of the first computers. It was Claude Shannon who realized that this algebra could be applied in the design and analysis of electrical circuits.

Boolean algebra deals exclusively with **0s** and **1s** .

The fundamental components of all our computers and digital memories use binary coding and Boolean algebra.

The smallest unit of storage is the byte. It is a space made up of 8 bits. A bit can have only two states: **0** or **1** . The smallest value that can be stored in a byte is **00000000** , the largest being **11111111** . If we cut a byte in two, we will have:

- four least significant bits, which can take the values **0000** to **1111** ;
- four most significant bits that can take one of these same values.

If we number all the combinations between 0000 and 1111, starting from 0, we arrive at 15:

```
: binary ( -- )
  2 base !
;
: bin0to15 ( -- )
  binary
  $10 0 do
    cr i .
    i hex . binary
  loop
  cr decimal ;
bin0to15 \ display:
0 0
1 1
10 2
11 3
100 4
101 5
110 6
111 7
1000 8
1001 9
1010 A
1011 B
1100 C
1101 D
1110 E
1111 F
```

On the right side of each line, we display the same value as on the left side, but in hexadecimal: **1101** and **D** are the same values!

Hexadecimal representation was chosen to represent numbers in computing for practical reasons. For the most significant or least significant part of a byte, on 4 bits, the only combinations of hexadecimal representation will be between **0** and **F**. Here, the letters A to F **are hexadecimal digits** !

```
$3E      \ is more readable as 00111110
```

Hexadecimal representation therefore offers the advantage of representing the contents of a byte in a fixed format, from **00** to **FF** . In decimal, it would have been necessary to use 0 to 255.

Size of numbers on FORTH data stack

Z79Forth uses a 16-bit data stack of memory size, or 2 bytes (8 bits x 2 = 16 bits). The smallest hexadecimal value that can be pushed onto the FORTH stack will be **0000** , the largest will be **FFFF**. Any attempt to push a value of greater size results in an error:

```
abcdefabcdefabcdef \ display:  
OoR error (0000/0000)  
F58F >NUMBER+0081
```

Let's stack the largest possible value in 16-bit (2-byte) hexadecimal format:

```
decimal  
$ffff . \ display: -1
```

I see you surprised, but this result is **normal** ! The word **.** displays the value that is at the top of the data stack in its signed form. To display the same unsigned value, you must use the word **u.** :

```
$ffff u . \ display: 65535
```

This is because of the 16 bits used by FORTH to represent an integer, the most significant bit is used as the sign:

- if the most significant bit is **0** , the number is positive;
- if the most significant bit is **1** , the number is negative.

So, if you followed along, our decimal values 1 and -1 are represented on the stack, in binary format in this form:

```
2 base!  
0000000000000001 \ push 1 on stack  
1111111111111111 \ push -1 on stack
```

And this is where we will call on our mathematician, Mr. LEIBNITZ, to add these two numbers in binary. If we do as in school, starting from the right, we will simply have to respect this rule: 1 + 1 = 10 in binary. We put the results on a third line:

```
0000000000000000 1  
1111111111111111 1  
                  1 0
```

Next step:

```
00000000000000001
1111111111111111 1 1
      1 0
      1 00
```

When we reach the end, we will have the following result:

```
00000000000000001
1111111111111111
10000000000000000
```

But since this result has a 17th most significant bit of 1, knowing that the integer format is strictly limited to 16 bits, the final result is 0. Is this surprising? Yet this is what any digital clock does. Hide the hours. When you get to 59, add 1, the clock will display 0.

The rules of decimal arithmetic, namely $-1 + 1 = 0$ have been perfectly respected in binary logic!

Memory access and logical operations

The data stack is not a data storage space in any case. Its size is very limited. And the stack is shared by many words. The order of the parameters is fundamental. An error can generate malfunctions:

```
variable score
```

Let's store any value in **score** :

```
decimal
1900 score!
```

Let's get the contents of **score** :

```
score @ .      \ display 1900
```

To isolate the low-order byte. Several solutions are available to us. One solution exploits binary masking with the logical operator **and** :

```
hex
score @ .      \ display: 76C
score @
$00 FF and .   \ display: 6C
```

To isolate the second byte from the right:

```
score @
$ FF 00 and .   \ display: 700
```

Here we had some fun with the contents of a variable.

To conclude this chapter, there is still much to learn about binary logic and the different possible digital encodings. If you have tested the few examples given here, you certainly understand that FORTH is an interesting language:

- thanks to its interpreter which allows numerous tests to be carried out interactively without requiring recompilation in code transmission;
- a dictionary with most words accessible from the interpreter;
- a compiler that allows you to add new words *on the fly* , then test them immediately.

Finally, what does not spoil anything, the FORTH code, once compiled, is certainly as efficient as its equivalent in C language.

A real 16-bit FORTH with Z79Forth

Z79Forth is a real 16-bit FORTH. What does it mean ?

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

Values on the data stack

When starting Z79Forth, the FORTH interpreter is available. If you enter any number, it will be dropped onto the stack as a 16-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 16-bit integer values are added together and the result is dropped 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 +. \ displays 80
```

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

With Z79Forth, an ASCII character will be designated by a 16-bit integer, but whose value will be bounded [32..256[. Example :

```
decimal  
67 emit \display C
```

With Z79Forth, a signed 16-bit integer will be defined in the range -32768 to 32767.

Sometimes we talk about half-heartedness. A digital halfword is the high or low 8-bit portion of a 16-bit integer. A half word will be defined in the range 0 to 256.

Values in memory

Z79Forth allows you to define constants and variables. Their content will always be in 16-bit format. But there are situations where that doesn't necessarily suit us. Let's take a simple example, defining a Morse code alphabet. We only need a few bytes:

- one to define 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, several bytes are stored. The question is: how will we retrieve the information in these words?

The execution of one of these words deposits a 16-bit value, a value which corresponds to the memory address where we stored our Morse code information. It is the word **c@** that we will use to extract the Morse code from each letter:

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

The first byte extracted like this 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
;
my .morse \ displays .-
mB .morse \ displays -...
mC .morse \ displays -.-.
```

There are plenty of certainly more elegant examples. Here, it's to show a way of manipulating 8-bit values, our bytes, while we use these bytes on a 16-bit stack.

Word processing depending on data size or type

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

```
$bread = "Baked bread";
$price = 2.30;
echo $bread . " : " . $price;
// 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 "Cooked bread" ;
: price s "2.30";
bread type    s" : " type    price type
\ display    Baked bread: 2.30
```

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

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

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

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

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

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

Conclusion

FORTH has no data typing. All data passes through a data stack. Each position in the Z79Forth stack is ALWAYS a 16-bit integer!

That's all there is to know.

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

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

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

```
: morse: ( comp: c -- | exec -- )
  create
  ,
  does>
    dup 1 cells + swap @ 0 do
      dup i + c@ emit
    loop
    drop space
  ;
2 morse: mA      char . c,   char - c,
4 morse: mB      char - c,   char . c,   char . c,   char . c,
4 morse: mC      char - c,   char . c,   char - c,   char . c,
mA mB mC        \ display   .- -... -.-.
```

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

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

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. The range of definition of signed 16-bit integers is -32768 to 32767. Example:

```
32767 .          \ displays 32767
32767 1+ .       \ displays -32768
-1 u.           \ displays 65535
```

These numbers can be processed in any numeric base, all numeric 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.

Z79Forth has two predefined words to select different numeric bases:

- **DECIMAL** to select the decimal numeric base. This is the numeric base taken by default when starting Z79Forth;
- **HEX** to select the hexadecimal number 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 (256 in decimal)
```

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

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

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

```
VARIABLE RANGE_BASE \ RANGE_BASE variable definition
BASE @ RANGE_BASE ! \ storage 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.

Prefixing whole numbers

Z79Forth allows the entry of numeric values in one of the following bases by prefixing these values as follows:

- prefix **\$** to mark an integer in hexadecimal base;
- **%** prefix to mark an integer in binary base;
- prefix **&** or **#** to mark an integer in decimal base;
- **@** prefix to mark an integer in octal base.

Example :

```
hex F7 2 base! 00001111 and decimal.
```

Can be rewritten with numeric prefixes:

```
$F7 %00001111 and .
```

These prefixes offer the advantage of not causing an error if you accidentally select the wrong numeric base:

```
#101 . \ displays 101
%101 . \ displays 5
$101 . \ displays 257
```

Defining new display formats

Forth has primitives that allow you to adapt the display of a number to any format. With Z79Forth, 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 -- )
  s>d
  <# # # [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 **s>d** converts a 16-bit integer to a 32-bit integer,
- 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 following **,**
- 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 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
s>d <#:00:00 #S #> SPACETYPE ;

```

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 numeral 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
  s>d
  <#
  4 0 DO            \ Format Loop
    FOUR-DIGITS
  LOOP
  #> TYPE SPACE     \ Binary display
  R> BASE ! ;       \ Initial digital base restoration

```

Example of execution:

```

#12 AFB          \ displays 0000 0000 0000 0110
$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 ;
: DOHERTY ( -- )
  618051254. .TEL ;
doherty          \ displays: 06.18.05.12.54

```

This telephone directory, which can be compiled from a source file, is easily editable, and although the names are not classified, 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:

```
#out variable
: #out+! ( n -- )
  #out +! \ increments #out
;
: (.) ( n -- al )
  DUP ABS s>d <# #S SWAP SIGN #>
;
: .R ( nl -- )
  >R (.) R> OVER - SPACES TYPE
;
: ASCII-CHARS (---)
  cr 0 #out !
  128 32
  DO
    I 3 .R SPACE      \ displays character code
  4 #out+!
  I EMIT 2 SPACES      \ display character
  3 #out+!
  #out @ 77 =
  IF
    CR 0 #out!
  THEN
  LOOP ;
```

Running **ASCII-CHARS** 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 ascii-chars
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:

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


Comments and debugging

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

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

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

```
: cycle.stop -1 +to MAX_LIGHT_TIME MAX_LIGHT_TIME 0 = if
0 myLIGHTS ! then ;
```

This code will be perfectly compiled by Z79Forth. 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 **cycle.stop**. Z79Forth allows to write word names of limited size. The size of the words defined within this limit 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: **cycle.stop**
- in CamelCoding style **cycleStop way**
- for programmers wanting very understandable code **cycle-stop-lights**
- programmer who likes concise code **cs1**

Z79Forth will transform all characters in a word to uppercase: **CYCLE.STOP**, **CYCLESTOP**, **CYCLE-STOP-LIGHT**, **CSL**.

There are no naming rules. The main thing is that you can easily reread your FORTH code. However, FORTH programmers have certain habits:

- **MAX_LIGHT_TIME** uppercase constants
- definition word of other words **defColor:**, that is to say word followed by a colon;

4 Integrated Development Environment = Integrated Development Environment

- address transformation word **->date** , here the parameter will be 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** !

Attention :

- If the terminal does not support UNICODE characters, Z79Forth does not accept names written in alphabets other than the Latin alphabet.
- If the terminal supports UNICODE characters, you will be able to use all possible alphabets.

You can use these alphabets for comments:

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

Or

```
: .date \ 海报今天的日期
...code... ;
```

Characters from a language encoding and used in comments are only usable in your source code.

Use UNICODE characters in names

The fascinating thing about FORTH is that it is a computer language that can quickly diverge from the norm established in other programming languages.

In all programming languages, function names can only be written in Latin characters and are limited to the ASCII-US set. Only HTML and its extension CSS can deviate by accepting class names in other alphabets. Example of a CSS class:

```
.signature {
  margin-top: 16px;
  margin-bottom: 50px;
  text-align: right;
}
```

It is quite possible to write **.signature** in Russian:

```
.сигнатура {
  margin-top: 16px;
  margin-bottom: 50px;
  text-align: right;
}
```

Similarly, with Z79Forth, we can define a word with this alphabet:

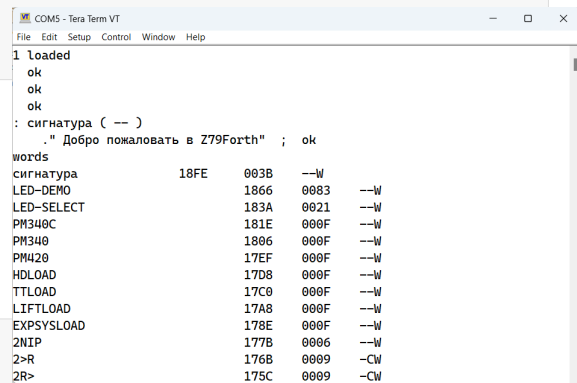
```
: сигнатура ( -- )
  ." Добро пожаловать в Z79Forth" ;
```

Let's run **words** :

The word **сигнатура** appears well in the FORTH dictionary.

AND we can execute this word

```
сигнатура \ displays:
Добро пожаловать в Z79Forth ok
```



It works just as well with Chinese, Arabic, Hebrew, Hindi, etc. characters. But don't abuse this facility!

Source code indentation

Whether the code is written 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 :

```
variable MAX_LIGHT_TIME
: cycle.stop
  -1 MAX_LIGHT_TIME +!
  MAX_LIGHT_TIME 0 =
  if
    0 myLIGHTS !
  then
;
;
```

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

```
: cycle.stop
  -1 MAX_LIGHT_TIME +!
  MAX_LIGHT_TIME 0 =
  if 0 myLIGHTS ! then
;
;
```

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

Stack Comments

As we have just seen, they are marked by `(` and `)` . Their content has no effect on the FORTH code during compilation or execution. 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 Z79Forth, the stack data takes up two bytes. They are therefore integers in 16-bit format. However, some words process data in eight-bit format. So what do we put on the data stack? With Z79Forth, it will **ALWAYS be 16-BIT DATA** ! An example with the word `c` !:

```
variable myDelimiter
64 myDelimiter c!    ( c addr -- )
```

`c` parameter indicates that we are stacking an integer value in 16-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:

```
variable myDelimiter
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.
With Z79Forth, this is in 32-bit format;
- **fl** indicates a boolean value, 0 or non-zero;
- **n** indicates an integer. 16-bit signed integer for Z79Forth;
- **str** indicates a string. Equivalent to **addr len --**
- **u** indicates an unsigned integer

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

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

Word Definition Words Comments

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

```
\ define a command or data stream for SSD1306
: streamCreate: ( comp: <name> | exec: -- addr len )
  create
    here    \ leave current dictionary pointer on stack
    0 c,    \ initial lenght data is 0
  does>
    dup 1+ swap c@
    \ send a data array
;
```

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:

```
\ *****
\ Manage commands for display
\   Filename:      commands.fs
\   Date:          21 may 2023
\   Updated:       21 may 2023
\   File Version:  1.0
\   Forth:         Z79Forth all versions 7.x++
\   Copyright:     Marc PETREMANNN
\   Author:        Marc PETREMANNN
\   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.

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.

Once the word run perfectly, we can remove our tracing word.

```
: myTEST
  128 32 do i emit
  loop
;
```

The decompiler

In a conventional compiler, the source code is transformed into object 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 have access to the executable code contained in his libraries with the compiler's resources alone.

With Z79Forth, the developer can decompile his routines, but also visualize the predefined primitives and therefore understand the functioning of his program or of FORTH in its entirety.

To see the definition of a word, simply type **SEE** followed by the word to be decompiled.

Example of compilation:

```
: C>F ( °C --- °F) \ Conversion Celsius in Fahrenheit
9 5 */ 32 +
;
see c>f
\ display:
1AF6 8E0009      lxi      ???
1AF9 BDE7DB      jsr      NPUSH
1AFC 8E0005      lxi      ???
1AFF BDE7DB      jsr      NPUSH
1B02 BDFA70      jsr      */
1B05 8E0020      lxi      ???
1B08 BDE7DB      jsr      NPUSH
1B0B 7EF899      jmp      + ok
```

Let's test **SEE** on a predefined word from the vocabulary:

```
see dup \ display :
FC85 BDE9D7      jsr      CKDPTRA
FC88 AE          fcb      $AE
FC89 C4          fcb      $C4
FC8A 7EE7DB      jmp      NPUSH ok
```

Here we have just decompiled **dup** . We can see that **dup** is written in assembly language...

Dictionary / Stack / Variables / Constants

Expand the dictionary

Forth belongs to the class of threaded 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, Z79Forth constructs the number in the dictionary. 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. Z79Forth incorporates 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

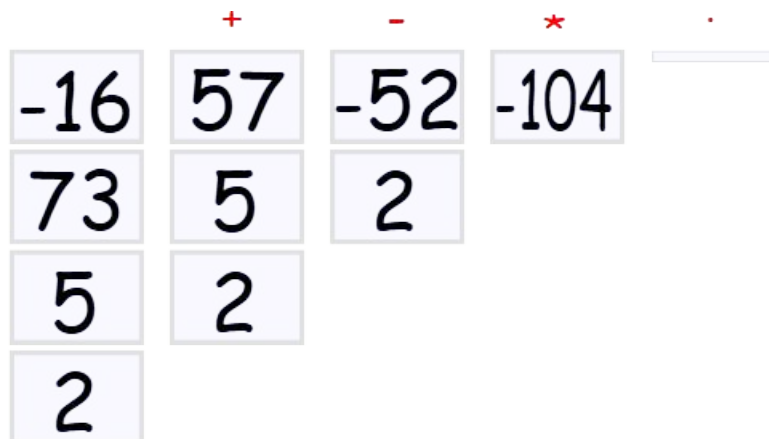
Cell	content	comment
0	-16	(TOS) Top of stack
1	73	(NOS) Next in the stack
2	5	
3	2	

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: -104
```

Note that Z79Forth conveniently displays the stack elements as it interprets each line, and the value of -16 is displayed as a 16-bit unsigned integer. 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 **Data stack underflow** stack pointer 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, Z79Forth 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 make numbers part of 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 other numbers down. **rot** rotates the first 3 numbers. These actions are shown below.



Figure 18: data stack manipulation

The Return Stack and Its Uses

The user can store and retrieve from the return stack. If you use the return stack for temporary storage, you must return it to its original state, or you will likely crash the Z79Forth 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. The word **r@** copies the top of the return stack to the parameter stack.

The words **>r** and **r>** can only be used in a FORTH definition.

Memory Usage

In Z79Forth, 16-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 in. It places the number in the memory location referenced by the address, consuming both parameters in the process.

Unsigned numbers that represent 8-bit (byte) values can be placed in character-sized characters, 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 of the contents of x
```

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 an XY matrix
64 constant matrix-width
16 constant matrix-height

create my-matrix
matrix-width matrix-height * allot
```

Pseudo-constant values

A value defined with **value** is a hybrid type of variable and constant. We define and initialize a value. This value 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** .

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** in 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
```

Word 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 Z79Forth, 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.

create has also been 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 ,
```

temperatures word parameter area with the word **,** .

With Z79Forth, we will see how to customize the execution of words defined by **create** .

Using does>

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

Here's how it generally works in Forth:

- **CREATE** : this keyword is used to create a new data space in the Z79Forth 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:

```
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.

```
\ define a color, similar as constant
: defCOLOR:
  create ( addr1 -- <name> )
  ,
  does> ( -- regAddr )
  @
```

```
;
```

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

```
$0000FF constant BLUE
```

Or

```
$0000FF defCOLOR: BLUE
```

are similar. However, by creating our colors with **defCOLOR:** we have the following advantages:

- a more readable Z79Forth source code. We can easily detect all the constants naming a color;
- we leave ourselves the possibility of modifying the **does>** part of **defCOLOR:** without having to then rewrite the lines of code which would not use **defCOLOR:**

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
: color: ( n -- <name> )
  create
  ,
  does>
    @ to currentCOLOR
;

$00 color: setBLACK
```

```
$ff color: setWHITE
```

Running the **setBLACK** or **setWHITE** word greatly simplifies the Z79Forth code. Without this mechanism, one of these lines would have had to be repeated regularly:

```
$00 currentCOLOR !
```

Or

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

Example, writing in pinyin

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

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

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

In this very simplified code, we can program pinyin words to write in Mandarin. The following code only works with the PuTTY terminal:

```
\ Work with PuTTY terminal or Tera Term in VT525 emulation  
: chinese:  
  create ( c1 c2 c3 -- )  
    c, c, c,  
  does>  
    3 type  
;
```

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

```
Good Morning --> 早安 (Zao an)
```

Copy 早 and go to PuTTY terminal and type:

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

paste the character 早. Z79Forth should display the following codes:

```
230 151 169
```

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

```
169 151 230 chinese: Zao  
137 174 229 Chinese: An
```

Use :

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

Marking functional layers with MARKER

Z79Forth does not have the word **FORGET** .

The word **FORGET** appears from the very beginning of the FORTH language in almost all standards: 79-Standard, 83-Standard...

So why isn't it in Z79Forth?

In fact, the most recent standard, forth200x, has declared the word **FORGET** obsolete. It may still remain in recent versions of FORTH as a concession for existing implementations to the older standards.

How the word FORGET works

This is how the word FORGET is used on versions of the FORTH language that include this word.

Let's take a series of definitions:

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

: gray> ( n -- n )
  0 1 16 lshift ( -- g b mask )
  begin
    >r          \ save mask on stack return
    2dup 2/ xor
    r@ and or
    r> 1 rshift
    dup 0=
  until
  drop nip ;    \ clears data stack leaving the result

: test
  2 base !      \ select binary base
  32 0 do
    cr I dup 5 .r ." ==> " \ displays values (binary)
    >gray dup 5 .r ." ==> " right justified on 5 characters
    gray>      5 .r
  loop
  decimal ;    \ puts back to decimal
```

How to remove the word **test** ?

With **FORGET** , just type:

```
FORGET test
```

If we then type **WORDS** , we will only find the definitions **>gray** and **gray>** .

The word **FORGET** therefore allows us to remove part of the definitions:

```
: w1 ; ok
: w2 ; ok
: w3 ; ok
: w4 ; ok
: w5 ; ok
words
w5 w4 w3 w2 w1 .....
forget w3
words
w2 w1 .....
```

forget w3 sequence removes from the dictionary the word **w3** and all words defined after it, i.e. **w4** and **w5** .

While the **FORGET** word is convenient in the development phase, it does present a risk of confusion by causing accidental deletion in sensitive dictionary areas. A vectorized execution word can have its vector destroyed. Execution of this word can then profoundly disrupt FORTH.

When developing in FORTH language, we first design sets of words by functional layers. This will be for example:

- a display management layer for a game;
- an application layer describing the game and using the previous layer;
- a final layer locking all application layers.

There is an elegant way to mark these different layers using the word **marker** .

Marking with the word marker

Forth allows you to forget words and everything that has been assigned in the dictionary after them. You have to use a marker created with **marker** :

```
-gray
marker -gray
: >gray ( n -- n' )
  dup 2/ xor ;
: gray> ( n -- n )
  0 1 31 lshift ( -- g b mask )
  begin
    >r 2dup 2/ xor r@ and or r> 1 rshift dup 0=
  until
  drop nip ;
```

We find these words in the dictionary:

```
words
gray> >gray -gray
```

It may seem surprising to invoke this word **-gray** before it is defined. Moreover, at the first compilation, this word causes an error message but does not prevent compilation:

```
-gray
'-gray' ? (0000/0000)
E44E NUMCVRA
E0D0 NMCVIRA
E0BD MINTLRA
marker -gray ok
\ ...etc.....
```

One can manually type **-gray** and all words defined after **-gray** will be removed from the dictionary, including **-gray** .

But it's even more interesting when we recompile our code. Because the first word found by the FORTH interpreter will be **-gray** :

```
-gray ok
marker -gray ok
```

At first compilation we had **'-gray'? (0000/0000)** .

On the next compilation, we have **-gray ok** .

We can recompile our FORTH code as many times as necessary. Each time the interpreter encounters the **-gray** marker , the memory space taken by all the words previously defined after **-gray** will be reclaimed.

This freed space is available through the new definitions.

Here we have a considerable advantage over other programming languages: the ability to handle definitions compiled by functional layers!

Each layer can be compiled after another software layer, itself followed by other software layers.

When the application development is finalized, we can delete all the lines of code exploiting **marker** .

Data structures for Z79Forth

Preamble

Z79Forth is a 16-bit version of the FORTH language. 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 **cells** . Executing this word for Z79Forth:

```
1 cells . \ displays 2
```

The value 2 means that the size of the elements placed on the data stack is 2 bytes, or 2x8 bits = 16 bits.

With a 32-bit FORTH version, **cells** will stack the value 4. Similarly, if you are using a 64-bit version, **cells** will stack the value 8.

Tables in FORTH

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

One-dimensional 16-bit data array

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 cells \ 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 )
  cells 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 cells +
;
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 half the memory space as using 16-bit cells (2 bytes).

Reading and writing in a table

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

```

arrayC myCTemps
  6 allot           \ reserve 6 bytes
  0 myCTemps 6 0 fill \ fill these 6 bytes with value 0
32 0 myCTemps c!    \ stores 32 in myCTemps[0]
25 5 myCTemps c!    \ stores 25 in myCTemps[5]
0 myCTemps c@ .     \ displays 32

```

In our example, the array contains 6 elements.

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          \ reserve 63 * 16 bytes
  mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill \ fill this space 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 ( xy -- addr )
  SCR_WIDTH * +
  mySCREEN +
  ;
: SCR@ ( xy -- c )
  xySCRaddr c@
  ;
: SCR! ( cxy -- )
  xySCRaddr c!
  ;
char X 15 5 SCR!      \ stores char X at col 15 line 5
15 5 SCR@ emit        \ display X

```

Management of complex structures

The C language has instructions for defining complex structures. These instructions do not exist in Z79Forth.

Definition of struct and field

In the previous examples, we saw how to define one- and two-dimensional data arrays. Let's take the case of a two-dimensional array again:

```

5 constant spriteWidth
7 constant spriteHeight
create mySprite
  spriteWidth spriteHeight * allot

```

If I want to define a sprite with other dimensions, I have to define two other constants, specific to this new sprite.

The idea would be to embed the dimensions of each sprite in the data of this sprite:

```

create mySprite
  5 , 7 ,
  5 7 * allot

```

The first two cells contain the width and height of the sprite respectively. To access our sprite's data, let's define three accessors:

```

: ->sprite.width ( addr - addr' )
  0 + ;

```

```

: ->sprite.height ( addr - addr' )
  2 + ;
: ->sprite.content ( addr - addr' )
  4 + ;

```

To access the first address of our sprite's data:

```
mySprite ->sprite.content
```

The first idea that comes to mind will be to factorize the creation of these accessors through a generic **field** definition :

```

i  int field ->sprite.width
   int field ->sprite.height
   int field ->sprite.content

```

The word **int** is a kind of constant defined as follows:

```

: typer ( len -- )
  constant
;

1 typer char
1 typer byte
2 typer int
4 typer double
1 typer i8
2 typer i16
4 typer i32

```

The idea, using these constants, will be to increment a counter each time a new field is declared with **field** . This is what we will achieve in the definition of **field** :

```

\ Define a field in data structure
: field ( comp: c -- | exec: addr -- addr' )
  create
    dup
      last-struct @ ,      \ get and compile latest position in structure
      last-struct +!      \ increment latest structure
    does>
      @ +                  \ get real position of data in structure
;

```

last-struct value points to the last defined structure. This structure will remember each new position of a field in this structure. A structure will be initiated by some kind of variable:

```

\ store parameter address of latest defined structure
0 value last-struct

\ Define a new structure
: struct ( comp: -- <name> | -- n )

```

```

create
    0 ,                \ initial value
    last count + 2 +
    >body to last-struct \ store param. addr in last-struct
does>
    @
;

```

Information : the syntax and use of the words **struct** , **field** , **typer** are inspired by those defined in the eForth and ESP32Forth versions created by Brad NELSON.

The full source code for these words adapted to Z79Forth is available here:

<https://github.com/MPETREMANN11/Z79Forth/blob/master/SW/MP%20extensions/structures.4th>

Examples of structures

To define a sprite structure:

```

struct sprite
    int field ->sprite.width
    int field ->sprite.height
    int field ->sprite.content

create bigSmiley
    5 , 7 , 5 7 * allot

bigSmiley ->sprite.width @ . \ displays sprite width
bigSmiley ->sprite.height @ \ displays sprite height

```

Here is another trivial example of structure:

```

struct YMDHMS
    int field ->year
    byte field ->month
    byte field ->day
    byte field ->hour
    byte field ->min
    byte field ->sec

```

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

YMDHMS word has the sole purpose of initializing and associating accessors to the complex structure. Here is how these accessors are used:

```

create DateTime
    YMDHMS allot

2022 DateTime ->year !

```



```

03 DateTime ->month c!
21 DateTime ->day   c!
22 DateTime ->hour  c!
36 DateTime ->min   c!
15 DateTime ->sec   c!

: .date ( date -- )
  >r
  ." YEAR: " r@ ->year   @ . cr
  ." MONTH: " r@ ->month c@ . cr
  ." DAY: " r@ ->day     c@ . cr
  ." HH: " r@ ->hour    c@ . cr
  ." MM: " r@ ->min     c@ . cr
  ." SS: " r@ ->sec     c@ . cr
  r> drop
;

DateTime .date

```

DateTime word is defined as an array. Access to each field in this array is done through its accessor.

In **YMDHMS** structure , the year is in 16-bit format, all other fields are reduced to 8-bit integers. In the **.date** code, the use of accessors allows easy access to each element of our complex structure.

Unicode characters with emit and key

As a preamble, you have installed a VT compatible terminal.

On Windows, I recommend the **TeraTerm** terminal , which is VT100 to VT525 compatible. **TeraTerm** does not fully emulate all the features of VT terminals, but for our purposes, we will analyze how to handle Unicode characters.

Let's start with the displayable ASCII characters, those whose code is between 32 and 127. Here's how to display the table of available 7-bit characters:

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

tableChars

```
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 ¯
```

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

If we try to display a character with a code greater than 127, there will be a display error.

```
132 emit \ display ❖
215 emit \ display ❖
```

Unicode encoding

Unicode is a standardized encoding system that assigns a unique number to each character, allowing computers to recognize and display them correctly, regardless of the language or platform used.

Each Unicode character is associated with a code point, a number that uniquely identifies it. This code point is then converted into a sequence of bytes to be stored in a computer.

There are different Unicode encodings, the most common being UTF-8 and UTF-16. These encodings determine how code points are converted into bytes.

UTF-8 is the most widely used encoding on the Internet. It is compatible with older systems that do not support Unicode and is very effective for texts in English and Western European languages.

Each Unicode character is represented by a sequence of one to four bytes. The number of bytes required depends on the Unicode code point value of the character. The first bits of each byte in a sequence indicate the total number of bytes in the sequence. This lets the decoder know immediately how many bytes it needs to read to reconstruct the character.

- **initial byte** : always starts with a bit sequence '110' for a 2-byte sequence, '1110' for a 3-byte sequence, or '11110' for a 4-byte sequence. The following bits contain part of the Unicode code point.
- **Next bytes** : Always start with the bit sequence '10'. The following bits also contain part of the Unicode code point.

Let's take the character "é" (acute accent, used in France). Its Unicode code point is U+00E9. It will be encoded on two bytes in UTF-8 as follows:

- **first byte**: **11000011** (the '110' indicates a 2-byte sequence), \$C3 in hexadecimal
- **second byte** : **10111001** , \$A9 in hexadecimal

```
$c3 emit $a7 emit \ displays é
```


Retrieve a Unicode character from the terminal

A Unicode UTF8 character cannot be retrieved simply by the word **key** . Executing **key** only works for characters with an ASCII code between 32 and 127. To retrieve the values constituting a Unicode character with a code greater than 127, **key** must be executed several times, typically 2 to 4 times:

```
hex key key . . \ press 'é' on keyboard, display A9 C3
```


Here is the definition of the word **ukey** which will test the character entered at the terminal and stack one to four bytes corresponding to the Unicode sequence of the character

```
\ get an UNICODE character
: ukey ( -- )
  key dup 1 lshift >r
  begin
    r@ $80 and
  while
    r> 1 lshift >r
    key
  repeat
  r> drop
;
```


For all other characters, UTF8 (Unicode) encoding will be used. Here is the encoding of the character :

```
$88 $96 $E2 emit emit emit \ displays 
```

We can simplify the coding by using a three-byte array:

```
create blackChar
$E2 c, $96 c, $88 c,
blackChar 3 type \ displays 
```


































Let's create a character definition word:

```
: Unicode: ( comp: c1 c2 c3 -- <mot> | exec: --- )
  create
    c, c, c,
  does>
    3 type ;
$88 $96 $E2 Unicode: blackChar
blackChar \ displays 
```

The character  is taken from the Unicode table documented here:


https://en.wikipedia.org/wiki/Block_Elements

Excerpt from this table:

Blocks																	
		0	1	2	3	4	5	6	7	8	9	H AS	B	C	D	E	F
U+258x		Do															
U+259x																	

By testing the first character  of this table with **ukey** , we obtain these codes:

```
hex ukey . . . \ displays 80 96 E2
```


Then we test the last character  with **ukey** :

```
hex ukey . . . \ displays 0F 96 E2
```

Note: To test a Unicode character with **ukey** , you must copy/paste the Unicode character after launching **ukey** . Unicode characters are not available directly on the keyboard, unless you know the ALT-NNNNNN key combination. These combinations may vary depending on the system used (Windows, Linux, MacOS, etc.).

Displaying Unicode Characters

To display Unicode characters, there are several solutions. The simplest is to use a string:

```
: upperHalfBlock ( -- )
  ."  " ;
```

Or via the **Unicode** definition:

```
$80 $96 $E2 Unicode:  upperHalfBlock
```

Running **upperHalfBlock** displays our **■** character. The **Unicode:** word exploits a very small three-byte array. This array stores the three successive codes of the **■** character. We could have just as easily used **emit** like this:

```
: upperHalfBlock ( -- )
    $E2 emit $96 emit $80 emit ;
```

The solution using **type** in the runtime part of **Unicode:** is the most compact.

What would happen if we used a four-byte array and just put the ASCII code of a single character in it?

```
create charA
    char A c,    0 c,    0 c,    0 c,
charA 4 type    \ displays A
```

Whether this table contains 1 or 2 or 3 or 4 valid bytes, if the sequence matches a valid Unicode character, that character will be displayed by **type** . The question will be: "how to display a Unicode character that is in a Unicode table?"

We will first define an empty character emission buffer:

```
\ buffer for Unicode char
create ubuffer
    4 allot
```

Then we define the display of the contents of this buffer:

```
\ display buffer content
: .ubuffer ( -- )
    ubuffer 4 type
;
```

We initialize the contents of this buffer by storing the Unicode sequence, on 4 bytes, of the first character of the character table that interests us. Here, it will be the value **\$E2968000** which corresponds to the character **■**:

```
\select 1st code of Unicode blocks set
: uBlocksSelect ( -- )
    ubuffer 4 erase
    $E2968000. ubuffer 2!
;
```

.uBlock word retrieves an index and calculates the value of the 3rd byte to be modified in the **ubuffer** buffer :

```
\display one char from Unicode blocks table
: .uBlock (i --) \i must be in interval [0..31]
    $80 + ubuffer 2 + c!
    .ubuffer
;
```

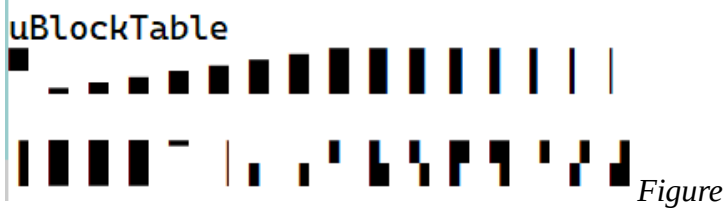
We check that all these definitions work well:

```
\ display table of Unicode blocks set
: uBlockTable ( -- )
```

```

cr
2 0 do
  16 0 do
    j $10 * i + .uBlock
    space
  loop
  cr cr
loop
;
uBlockSelect
uBlockTable

```



19: *uBlockTable* execution

Unicode characters in Forth words

The name structure in Z79Forth allows the use of Unicode characters. Example:

```

: été ." Summer" ;
été \ display été

```

We can also use our block Unicode characters:

```

: ■ 6 .uBlock ;

```

The character ■ becomes a FORTH word! Let's use it in a definition:

```

: bigLine ( n -- )
  0 do ■ loop
;

```

The word **bigLine** will generate a horizontal bar of n characters in length. This allows us to draw a graph:

```

: graph ( -- )
  cr
  06 bigLine cr

```

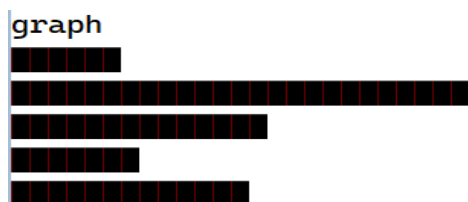


Figure 20: *graph* execution

```

25 bigLine cr
14 bigLine cr
07 bigLine cr
13 bigLine cr

```



This manipulation can work with many versions of the Forth language, especially any version that accesses Unicode characters, either directly or through a Unicode-aware terminal.

But using Unicode characters in names is discouraged, as it is outside of all programming standards.

Using binary code

With the exception of rare versions of the FORTH language written in certain programming languages, such as JavaScript, almost all versions have a kernel written in machine code (binary code).

This code can be generated by an assembler or by a compiler, in C language for example.

For this reason, a FORTH version always has these constraints:

- dependence on a certain type of processor (Intel, Motorola, Z80, 6809, Xtensa, etc.);
- dependence on a software environment (DOS, OSx, Windows, Linux, Android);

The eForth and derivative versions are built around a virtual FORTH machine described in C language. They are only linked to the processor via the binary libraries allowing the generation of the usable FORTH version.

When the first personal computers were originally designed, memory space was reduced to a bare minimum. Due to the very architecture of the processor, which had a 16-bit addressing mode, the usable programming environment was as follows:

- a programming and hardware control language in ROM or EPROM;
- a RAM space for user programs.

On these early personal computers, the programming language and RAM shared 64 kilobits of addressable memory space. Unless you used hardware tricks, you were limited to this space.

Retrocomputing and back to basics

You can use a modern computer without trying to understand how the first consumer personal computers worked. Let's briefly summarize the architecture of a modern computer:

- the hardware layer: a master card and its processor, control and communication devices (serial link, parallel, screen, keyboard, mouse, joystick)...
- BIOS: a fairly small program that gives the first instructions to the processor, often non-existent on home microcomputers;
- the operating system loaded by the BIOS which initializes the high-level hardware and software environment;
- applications started by the operating system or the user....

On a personal computer from the early days of computing, with assembly language, a little time and good documentation, it was possible, alone in one's corner, to set up an application. It was in the years 1977-1990 that the FORTH language experienced a certain glory.

FORTH: compact and elegant

The FORTH language is one of those languages that has attracted many developers. It is an atypical language, because there are no real applications: **the application extends the language !**

Features of FORTH versions for personal microcomputers:

- a 1KB to 2KB machine code kernel
- a set of primitives extending this core to 8KB
- some development assistance applications: editor, graphics management, etc....

This extraordinary compactness has as a corollary an unrivaled speed of execution, compared to other programming languages of the time: Basic, LISP, Logo...

Some versions of the FORTH language included a multi-tasking engine.

So why has FORTH almost disappeared in 2024? One answer would be: because of LINUX. LINUX is essentially written in C language. To unite thousands of programmers, it has built a development ecosystem, confining FORTH to a niche closer to hardware.

A C compiler has a very complex operation. To summarize, a source code, in C language is first pre-analyzed, then it is linked with the required libraries, followed by a pre-compilation and ending with the generation of the executable code.

The FORTH language uses two states:

- an interpreter: it reads instructions coming from the keyboard or from a source file;
- a compiler: it transforms the source code by extending the dictionary. This transformation is done in a single pass.

By using Z79Forth, you are therefore taking a real journey through time by finding a minimalist, but very efficient environment. The Z79Forth card accesses a very high capacity recording medium:

Compared to the old digital media available in the days of personal computers, being able to access such storage space for Z79Forth is simply luxurious!

A FORTH version with direct chaining

There are only two types of FORTH language architecture:

- **indirect chaining** : each word has an execution address and the FORTH compiler records the execution address of the words to be compiled. A FORTH engine will interpret each execution address;
- **direct chaining** : the execution address of a FORTH word is written in machine code. The FORTH compiler records a succession of subroutine calls to each compiled word in a definition.

Z79Forth uses direct chaining.

Decompile Z79Forth definitions

Z79Forth has a decompiler/disassembler. To decompile a definition, you need to use the **see word** :

```
see dup
FC85 BDE9D7    jsr    CKDPTRA
FC88 AE        fcb    $AE
FC89 C4        fcb    $C4
FC8A 7EE7DB    jmp     NPUSH
```

Here we have decompiled the word **dup** .

In this decompilation, what does the address **FC85 correspond to** ?

```
hex
' dup u.      \ displays: FC85
```

the address **FC85** is the address of the **dup** execution code .

In the decompilation of **dup** , we will distinguish three parts:

```
FC85 BDE9D7    jsr    CKDPTRA
FC88 AE        fcb     $AE
FC89 C4        fcb     $C4
FC8A 7EE7DB    jmp     NPUSH
```

- in green the address at which the executable code is physically located;
- in red the executable code;
- in blue the assembly translation of the executable code...

In the assembler translation, two labels appear, **CKDPTRA** and **NPUSH** . These two labels do not appear in the FORTH vocabulary. In the Z79Forth sources:

- **CKDPTRA** Check data stack minimum depth
- **NPUSH** Push n on data stack

Let's see what decompiling a definition compiled on top of the Z79Forth kernel gives:

```
: square dup * ;
3 square .      \ displays: 9
4 square .      \ displays: 16
```

The word **square** simply squares a number. Let's see its decompilation:

```
see square
18F2 BDFC85    jsr     DUP
18F5 7EF926    jmp     *
```

Using direct chaining, Z79Forth transformed **square** 's source code into machine code!

With Z79Forth, there is no need to code in machine language anymore, as the compiler produces perfectly optimized executable code.

Where is the address of the execution field of our square word?

```
' square hex u.      \ displays: 18F2
```

The address **18F2** is the same as where the compiled code for the definition of our word **square** starts.

What happens if we compile an empty definition?

```
: emptyDef ;
see emptyDef      \ displays:
1903 39           rts
```

The executable code of **emptyDef** contains a single-byte instruction! This information will be very useful later when considering machine language coding.

Coding definitions in machine language

As explained before, coding in machine language, with Z79Forth, is useless.

However, in some situations, coding in machine language allows you to define words that will execute faster than in high-level FORTH language. In the following example, we will swap the bytes of a 16-bit value placed on the data stack:

```
: bswap ; -1 allot
$ec c, $c4 c, \ ldd ,u
$1e c, $89 c, \ exg a,b
$ed c, $c4 c, \ std ,u
$39 c, \ rts
```

The binary code was generated using an online assembler:

<http://6809.uk/>

The binary code is retrieved from the *disassembly output column* for use by Z79Forth.

bswap word test :

```
hex
1234 bswap . \ displays: 3412
5472 bswap . \ displays: 7254
```

Here is what the decompilation of our **bswap** word gives:

```
see bswap \ displays:
190C EC fcb $EC
190D C4 fcb $C4
190E 1E fcb $1E
190F 89 fcb $89
1910 ED fcb $ED
1911 C4 fcb $C4
1912 39 rts
```

The way our **bswap** word has been coded must remain very punctual and respond exclusively to performance and compactness criteria:

- the use of machine code reduces the portability of the source code;
- you need to learn 6809/6309 assembler to use machine code;
- you need to have some tools outside of Z79Forth, here an online assembler.

Could **bswap** have been written entirely in FORTH? Yes:

```
: bswap ( n1 - n2 )
```

```
$100 /mod
swap $100 * + ;
```

This code is simply longer in terms of memory space occupied in the dictionary, but also slower in execution than the machine code version.

If you need to define a word in machine code, it is strongly recommended to put both versions in your source files, in this way:

```
\ invert bytes of n
\ : bswap ( n1 - n2 )
\      $100 /mod
\      swap $100 * + ;
: bswap ; -1 allot
$ec c, $c4 c, \ ldd ,u
$1e c, $89 c, \ exg a,b
$ed c, $c4 c, \ std ,u
$39 c, \ rts
```

Here, the version defined in FORTH is commented out.

This approach will make it easier to reuse the source code if you need to port the application to a version of the FORTH language other than Z79Forth.

Ressources

- **Z79FORTH / Facebook**
<https://www.facebook.com/groups/505661250539263>
- **Z79Forth Blog**
<https://z79forth.blogspot.com/>
- **FORTH code analyzer**
Enter your FORTH code. Select Z79Forth. Find your code with syntax highlighting and hyperlinks to known words
<https://analyzer.arduino-forth.com/>

GitHub

- **Z79forth** is a journey into retro computing that takes advantage of modern technologies where appropriate (CMOS, USB and CompactFlash)
<https://github.com/frenchie68/Z79Forth>

Youtube

- CPE1704TKS: Engineering Field Notes from a Debugging Session
<https://www.youtube.com/watch?v=OFIxfCywh0Q>

Index

and.....	25	GIT.....	16	space.....	36
base.....	31	hex.....	31	struct.....	60
binary code.....	68	hex.....	21	u.....	24
c!.....	47	HH:MM:SS format.....	29	value.....	48
c@.....	47	Hitachi HD6309.....	5	variable.....	48
constant.....	48	hold.....	33	45
create.....	28, 50	jumper JP1.....	10	45
decimal.....	21, 31	jumpers JP2 to JP5.....	10	45
direct chaining.....	70	marker.....	54	36
does>.....	50	memory.....	47	.s.....	43
drop.....	47	Netbeans.....	15	@.....	47
dup.....	47	prefix.....	32	#.....	33
emit.....	35	return stack.....	47	#>.....	33
features.....	7	s".....	36	#s.....	33
forget.....	53	serial transmission speed.....	10	<#.....	32