The manual for Z79FORTH

version 1.2.1 - 6 October 2024

Author

- François LAAGEL
- Marc PETREMANN

Collaborators

- …

Contents

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.

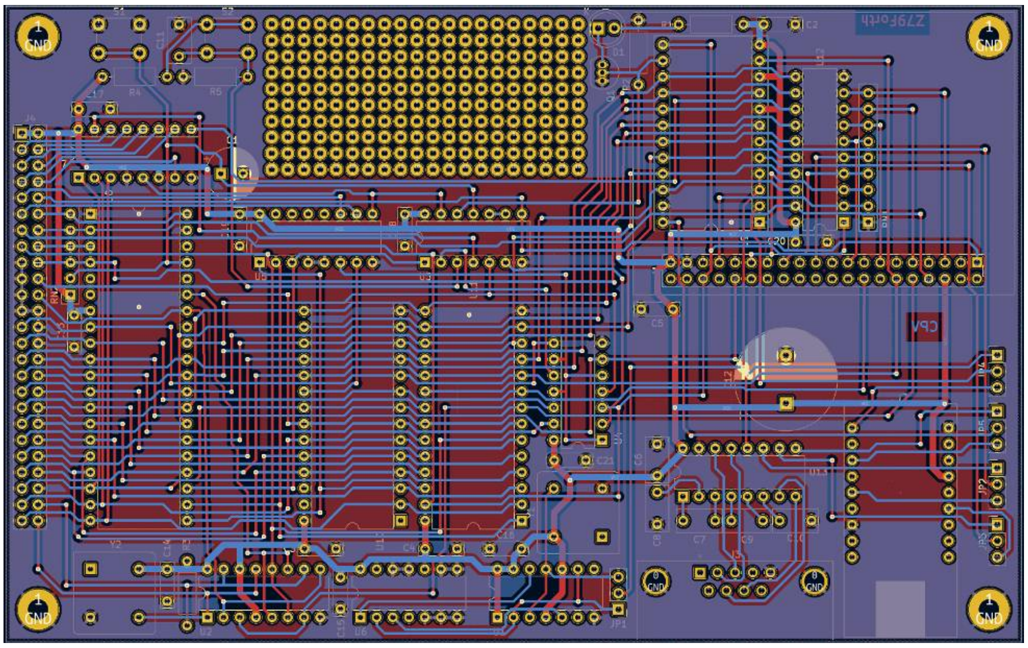
Object

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.

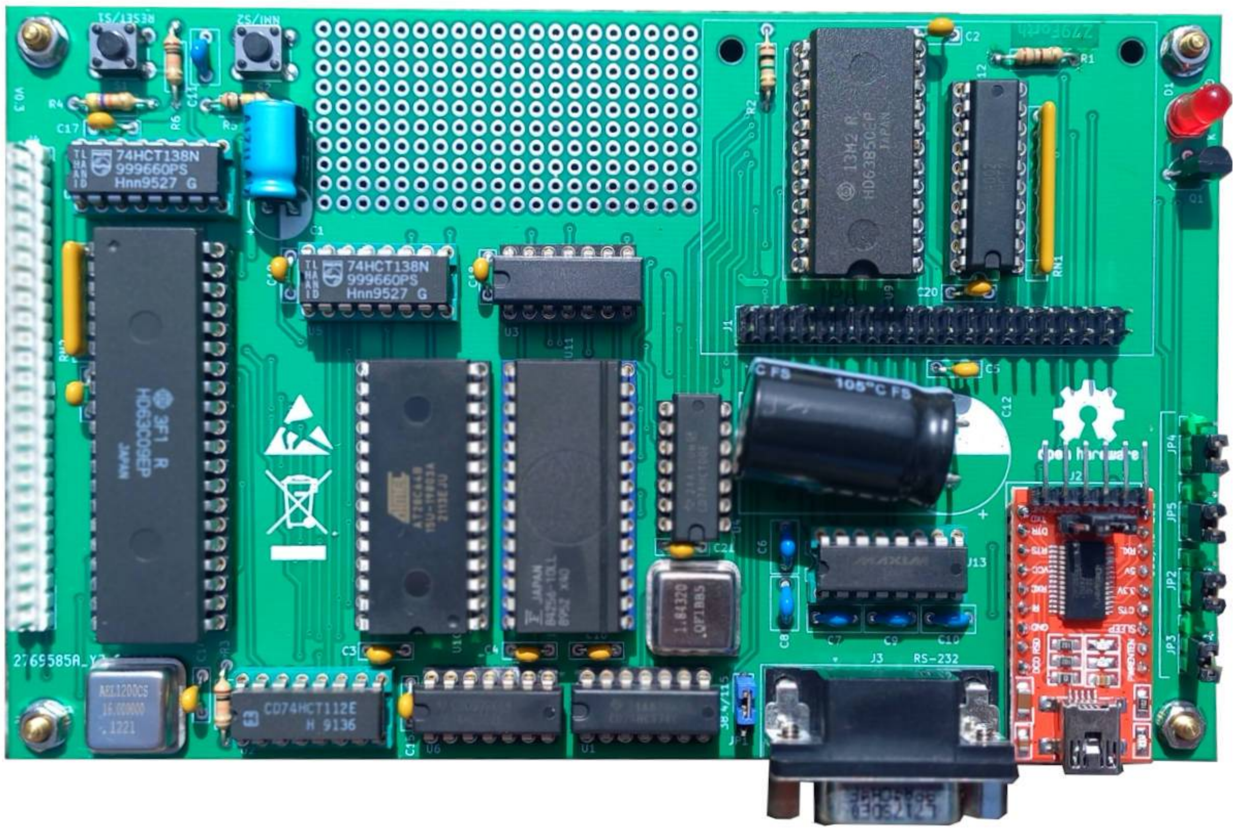


Figure 1: The 79Forth board

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 279Forth 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:

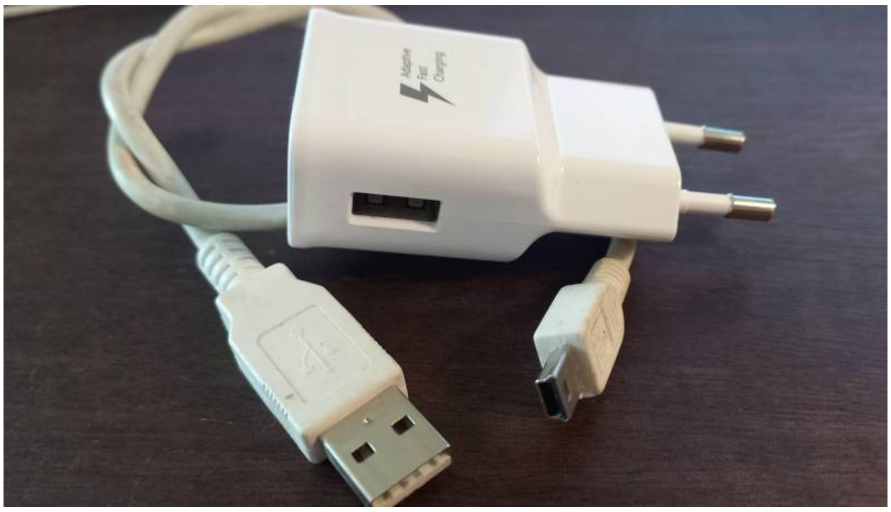


Figure 2: USB 2.0 cable and 230V->5V power supply

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.

Communication with Z79Forth board

...writing in progress....

Using Numbers with Z79Forth

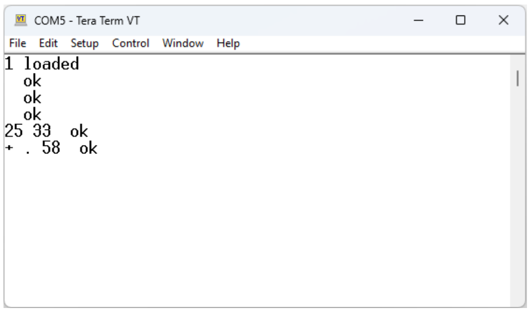
We've started Z79Forth without any issues. Now we'll dive deeper into some number crunching to understand how to master 79Forth.

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:



```
1 loaded
  ok
  ok
  ok
25 33  ok
+ . 58  ok
```

Figure 3: 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:

```
000000000000000 1
111111111111111 1
              1 0
```

Next step:

```
0000000000000001
11111111111111 1 1
              1 0
             1 00
```

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

```
 0000000000000001
 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...

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